Ejercicios de “Informática de 1ºde Matemáticas” (2009–10)

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 2010

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 2009–10.

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-09/temas/2009-10-IM-temas-PF.pdf

Relación 1

Introducción a la programación funcional

```
-- ---------------------------------------------------------------------
-- 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 con Haskell

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

import Test.QuickCheck

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

```haskell
--             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,
```

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

-- 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.

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

```haskell
-- 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

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

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

-- ---------------------------------------------------------------------
-- 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,
--    *Main> mitades [1,2,3,4,5,6]
```

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

```haskell
-- 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
disj_1 True  True  = True
```

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

-- La propiedad es
```

```haskell
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
```

```haskell
--    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
```

```haskell
-- 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 ==> múltiplo (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
```

```haskell
-- 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

Condicionales y guardas (1)

```haskell
-- ---------------------------------------------------------------------
-- 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
-- ax² + bx + c = 0. Por ejemplo,
--    raices 1 (-2) 1 => [1.0,1.0]
--    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
-- 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.
```

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

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
```

```haskell
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

Condicionales y guardas (2)

```haskell
module G1_Rel_4_sol where

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función media3 que reciba como argumentos
-- tres números y devuelva la media aritmética de dichos números. Por
-- ejemplo,
--    media3 1 3 8     =>  4.0
--    media3 (-1) 0 7  =>  2.0
--    media3 (-3) 0 3  =>  0.0
-- ---------------------------------------------------------------------

media3 :: Float -> Float -> Float -> Float
media3 x y z = (x+y+z)/3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función sumaMonedas que reciba los
-- siguientes argumentos: euros, dos-euros, cinco-euros, diez-euros y
-- veinte-eros y devuelva la cantidad total de 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 :: Int -> Int -> Int -> Int -> Int -> Int
sumaMonedas a b c d e = 1*a+2*b+5*c+10*d+20*e

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función areaDeCoronaCircular que calcule el
-- área de una corona circular a partir de los radios interior y
-- exterior de la misma.
--    areaDeCoronaCircular 1 2 => 9.42477796076938
--    areaDeCoronaCircular 2 5 => 65.9734457253857
--    areaDeCoronaCircular 3 5 => 50.2654824574367
-- Indicación: Usar la constante pi.
-- ---------------------------------------------------------------------

areaDeCoronaCircular :: Float -> Float -> Float
areaDeCoronaCircular r1 r2 = pi*(r2^2 -r1^2)

-- ---------------------------------------------------------------------
-- Ejercicio 4. 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 :: [a] -> [a] -> [a]
intercala [x1,x2] [y1,y2] = [x1,y1,x2,y2]

-- ---------------------------------------------------------------------
-- Ejercicio 5. 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 :: ((a,b,c),(d,e,f),(h,i,j)) -> ((a,d,h),(b,e,i),(c,f,j))
reagrupa ((x1,x2,x3),(y1,y2,y3),(z1,z2,z3)) =
    ((x1,y1,z1),(x2,y2,z2),(x3,y3,z3))
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función maxMin que reciba cuatro números, y
-- devuelva una lista formada por dos pares de la forma (”min”,x) y
-- (”max”. y) donde x e y son el mínimo y el máximo de los cuatro
-- números.
--    maxMin 2 5 (-1) 7  => [(”min”,-1),(”max”,7)]
--    maxMin 0 5 9 4     => [(”min”,0),(”max”,9)]
-- Indicación: Usar las funciones minimum y maximum.
-- ---------------------------------------------------------------------

maxMin :: (Num a, Ord a) => a -> a -> a -> a -> [(String,a)]
maxMin a b c d =
    [(”min”, minimum [a,b,c,d]),(”max”, maximum [a,b,c,d])]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función tresIguales tal que
-- (tresIguales x y z) se verifica si los elementos x, y y z son
-- iguales. Por ejemplo,
--    tresIguales 2 3 2  =>  False
--    tresIguales 2 2 2  =>  True
-- ---------------------------------------------------------------------

tresIguales :: Eq a => a -> a -> a -> Bool
tresIguales x y z = (x == y) && (y == z)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función tresDiferentes tal que
-- (tresDiferentes x y z) se verifica si los elementos x, y y z son
-- distintos. Por ejemplo,
--    tresDiferentes 2 3 5  =>  True
--    tresDiferentes 2 3 2  =>  False
-- ---------------------------------------------------------------------

tresDiferentes :: Eq a => a -> a -> a -> Bool
tresDiferentes x y z = (x /= y) && (y /= z) && (x /= z)

-- ---------------------------------------------------------------------
-- Ejercicio 9. 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 3 3 3 3  =>  True
--    cuatroIguales 3 3 2 3  =>  False
-- Indicación: Usar la función tresIguales.
-- ---------------------------------------------------------------------

cuatroIguales :: Eq a => a -> a -> a -> a -> Bool
cuatroIguales x y z u = (x == y) && tresIguales y z u

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función maxTres tal que (maxTres x y z) es
-- el máximo de x, y y z. Por ejemplo,
--    maxTres 7 3  5   =>  5
--    maxTres 7 12 5   =>  12
--    maxTres 7 12 50  => 50
-- ---------------------------------------------------------------------

maxTres :: Ord a => a -> a -> a -> a
maxTres x y z
    | x >= y && y >= z = x
    | y >= z           = y
    | otherwise        = z
```

Relación 5

Definiciones por comprensión

```haskell
module G1_Rel_5_sol where

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

import Test.QuickCheck
import Data.Char

-- ---------------------------------------------------------------------
-- Ejercicio 1. Usando una lista por comprensión, definir la constante
-- sumaDeCuadrados cuyo valor sea la suma 1^2 + 2^2 + ... 100^2 y
-- calculr su valor
-- ---------------------------------------------------------------------

-- La defición es
sumaDeCuadrados = sum [x^2 | x <- [1..100]]

-- Su evaluación es
--    *Main> sumaDeCuadrados
--    338350

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Redefinir por comprensión la función
--    replicate :: Int -> a -> [a]
-- tal que (replicate n x) es la lista formada por n copias del elemento
-- x. Por ejemplo,
```

```
--    *Main> replicate 3 True
--    [True, True, True]
-- Llamar la nueva definición replicate'.
-- ---------------------------------------------------------------------

-- La definición
replicate' :: Int -> a -> [a]
replicate' 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
-- (replicate n x) es n.
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- 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,
--    *Main> 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]
```

```haskell
-- ---------------------------------------------------------------------
-- 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]

-- ---------------------------------------------------------------------
-- 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.
--
```

```haskell
-- 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. Usando una lista por
-- comprensión y la función factores (del tema), definir la función
--    perfectos :: Int -> [Int]
-- tal que (perfectos n) es la lista de todos los números perfectos
-- menores que n. Por ejemplo:
--    *Main> perfectos 500
--    [6,28,496]
-- ---------------------------------------------------------------------

-- 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]

-- ---------------------------------------------------------------------
-- 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)]
--
-- Demostrar cómo la función pares puede redefinirse usando dos
-- listas por comprensión con un generador cada una. Comprobar con
-- QuickCheck que las dos definiciones son equivalentes.
--
-- Indicación: Utilizar la función predefinida concat y encajar una
```

```haskell
-- 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
--    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]
--
-- Redefinir la función posiciones usando la función busca (definida en
-- el tema). Llamar a la nueva función 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]
```

```haskell
-- 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. 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. Modificar el programa de cifrado César para que pueda
-- utilizar también letras mayúsculas. Por ejemplo,
-- ---------------------------------------------------------------------

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

```haskell
--    minuscula2int 'd'  =>  3
--    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
```

```haskell
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']
```

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

Relación 6

Definiciones por recursión (1)

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

import Test.QuickCheck
import Data.List (sort)

-- ------------------------------------------------------------------
-- Ejercicio 1 (Máximo común divisor)
-- ------------------------------------------------------------------
-- 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
-- ------------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función mcd.
-- ------------------------------------------------------------------------

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

-- ------------------------------------------------------------------------
-- Ejercicio 1.2. 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
```

```haskell
-- 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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. 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.

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

```haskell
-- Ejercicio 2 (Suma de cuadrados)
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir por recursión la función sumaCuadrados tal que
-- (sumaCuadrados n) es la suma de los cuadrados de los números de 1 a n. Por
-- ejemplo,
--    sumaCuadrados 4 => 30
-- ---------------------------------------------------------------------

-- La definición es
sumaCuadrados :: Integer -> Integer
sumaCuadrados 0         = 0
sumaCuadrados n | n > 0 = sumaCuadrados (n-1) + n*n

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. 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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Medidas de centralización)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    media :: [Double] -> Double
-- que calcule la media aritmética de una lista numérica. Por ejemplo,
--    media [1,2,3]       ==>  2.0
--    media [1,-2,3.5,4]  ==>  1.625
-- Nota: La expresión (fromIntegral x) es el número real correspondiente
-- al número entero x.
-- ---------------------------------------------------------------------

media :: [Double] -> Double
```

```haskell
media xs = (sum xs) / fromIntegral (length xs)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    mediana :: [Double] -> Double
-- que calcule la mediana de una lista numérica; esto es, el valor que deja el
-- mismo número de datos menores y mayores que él (si la lista contiene un
-- número par de elementos, la mediana será la media aritmética de los dos
-- valores centrales). Por ejemplo,
--    mediana [3,2,5,1,4]  ==>  3.0
--    mediana [-1,7,8,1]   ==>  4.0
-- Notas:
-- 1. (sort xs) es el resultado de ordenar la lista xs.
-- 2. (xs !! n) es el n-ésimo elemento de xs.
-- ---------------------------------------------------------------------

mediana :: [Double] -> Double
mediana xs | odd a     = ys !! ((a-1) `div` 2)
           | otherwise = (ys !! (a `div`2) + ys !! (a `div` 2 - 1) ) / 2
    where a  = length xs
          ys = sort xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. La función
--    divideMedia :: [Double] -> [[Double]]
-- dada una lista numérica, xs, calcula la lista [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 comprensión.
-- ---------------------------------------------------------------------

divideMedia :: [Double] -> [[Double]]
divideMedia xs = [[x | x <- xs, x < m], [x | x <- xs, x > m]]
    where m = media xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Dada una lista, xs, sus valores externos son aquellos valores
-- menores que media xs-(máximo xs-mínimo xs)/2) o mayores que
```

```haskell
-- media xs+(máximo xs-mínimo xs)/2).
--
-- Definir, por comprensión, la función
--    valoresExternos:: [Double] -> [Double]
-- que calcula los valores externos de una lista numérica. Por ejemplo,
--   valoresExternos [1,2,5,5,5,5,5,5]      ==>  [1.0,2.0]
--   valoresExternos [5,5,5,5,5,5,8,9]      ==>  [8.0,9.0]
--   valoresExternos [1,2,5,5,5,5,5,5,8,9]  ==>  []
-- ---------------------------------------------------------------------

valoresExternos :: [Double] -> [Double]
valoresExternos xs = [x | x <- xs, p x]
    where p x = x<m-(b-a)/2 || x>m+(b-a)/2
          m   = media xs
          a   = minimum xs
          b   = maximum xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Comprobar con QuickCheck si se verifica alguna de las
-- siguientes propiedades:
-- * La media de una lista no vacía es mayor o igual que la mediana.
-- * La media de una lista no vacía es menor o igual que la mediana.
-- * La media de una lista no vacía es menor o igual que (minimo + maximo)/2
-- ---------------------------------------------------------------------

-- Las propiedades son:
prop1, prop2, prop3 :: [Double] -> Property
prop1 xs = not (null xs) ==> media xs >= mediana xs
prop2 xs = not (null xs) ==> media xs <= mediana xs
prop3 xs = not (null xs) ==> media xs <= (minimum xs + maximum xs)/2

-- La comprobación con QuickCheck es
--    Main> quickCheck prop1
--    Falsifiable, after 3 tests:
--    [3.0,3.0,2.66666666666667,-0.8]
--
--    Main> quickCheck prop2
--    Falsifiable, after 10 tests:
--    [-4.33333333333333,-3.75,-2.0,4.0]
--
```

```
--    Main> quickCheck prop3
--    Falsifiable, after 2 tests:
--    [1.75,-4.33333333333333,2.66666666666667,-3.2]
-- Por tanto, no se verifica ninguna de las propiedades.

-- ---------------------------------------------------------------------
-- Ejercicio 4 (Extracciones e inserciones)
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función extrae tal que (extrae xs n) es la lista
-- resultado de eliminar el elemento n-ésimo de la lista xs. Si n es negativo
-- o mayor que la longitud de la lista el resultado debe ser la misma
-- lista. Por ejemplo,
--    extrae [1,2,3,4,5] 2     ==>  [1,2,4,5]
--    extrae [1,2,3,4,5] 4     ==>  [1,2,3,4]
--    extrae [1,2,3,4,5] (-1)  ==>  [1,2,3,4,5]
--    extrae [1,2,3,4,5] 7     ==>  [1,2,3,4,5]
-- Nota: (drop n xs) es la lista obtenida eliminando los n primeros
-- elementos de xs.
-- ---------------------------------------------------------------------

extrae :: [a] -> Int -> [a]
extrae xs n = (take n xs) ++ (drop (n+1) xs)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función inserta tal que (inserta xs e n) debe
-- introducir el elemento n en la posición n de la lista xs. Si (n<=0) el
-- elemento se insertará al principio de la lista, y si n es igual o mayor que
-- la longitud de la lista, el elemento e se colocará al final de la lista. Por
-- ejemplo,
--    inserta [1,3,5] 7 1     ==>  [1,7,3,5]
--    inserta [1,3,5] 7 3     ==>  [1,3,5,7]
--    inserta [1,3,5] 7 (-1)  ==>  [7,1,3,5]
--    inserta [1,3,5] 7 9     ==>  [1,3,5,7]
-- ---------------------------------------------------------------------

inserta :: [a] -> a -> Int -> [a]
inserta xs e n = (take n xs) ++ [e] ++ (drop n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que si se inserta el elemento
```

```
-- n-ésimo de una lista xs en el resultado de extraer el elemento n-ésimo de xs
-- se obtiene la lista xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_InsertaExtrae :: [Int] -> Int -> Property
prop_InsertaExtrae xs n =
    0<=n && n < length xs ==>
    inserta (extrae xs n) (xs!!n) n == xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. (Reconocimiento de permutaciones)
-- ---------------------------------------------------------------------
-- Una permutación de una lista es otra lista con los mismos elementos,
-- pero posiblemente en distinto orden. Por ejemplo, [1,2,1] es una
-- permutación de [2,1,1] pero no de [1,2,2]. En este ejercicio vamos a
-- estudiar las permutaciones.
-- ---------------------------------------------------------------------
-- Ejercicio 5.1. 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

-- Nota: la función borra es la función delete de la librería List.

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función esPermutación tal que
-- (esPermutación xs ys) se verifique si xs es una permutación de
```

```haskell
-- ys. Por ejemplo,
--    esPermutación [1,2,1] [2,1,1]  ==>  True
--    esPermutación [1,2,1] [1,2,2]  ==>  False
-- ---------------------------------------------------------------------

-- La definición es
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 5.3. Comprobar con QuickCheck que si una lista es una permutación
-- de otra, las dos tienen el mismo número de elementos.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_PemutacionConservaLongitud :: [Int] -> [Int] -> Property
prop_PemutacionConservaLongitud xs ys =
    esPermutacion xs ys ==> length xs == length ys

-- y la comprobación es
--    Main> quickCheck prop_PemutacionConservaLongitud
--    Arguments exhausted after 86 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. 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

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

Relación 7

Definiciones por recursión (2)

```haskell
-- ---------------------------------------------------------------------
-- Importaci□n de librer□as auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir por recursi□n la funci□n
--    potencia :: Integer -> Integer -> Integer
-- tal que (potencia x n) es x elevado al n□mero 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 c□mo se eval□a (potencia 2 3).
-- ---------------------------------------------------------------------

-- El c□lculo 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 funci□n 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

-- La comprobaci□n es
--    *Main> quickCheck prop_potencia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Mostrar c□mo a partir de la definici□n
--    length :: [a] -> Int
--    length []     = 0               -- length.1
--    length (_:xs) = 1 + length xs   -- length.2
-- se calcula
--    length [1,2,3]
-- ---------------------------------------------------------------------

-- El c□lculo es
--      length [1,2,3]
--    = 1 + length [2,3]           { por def. de length.2 }
--    = 1 + (1 + length [3])       { por def. de length.2 }
--    = 1 + (1 + (1 + length []))  { por def. de length.2 }
--    = 1 + (1 + (1 + 0)           { por def. de length.1 }
--    = 3                          { por def. de + }

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Mostrar c□mo a partir de la definici□n
--    drop :: Int -> [a] -> [a]
--    drop 0 xs         = xs           -- drop.1
--    drop (n+1) []     = []           -- drop.2
--    drop (n+1) (x:xs) = drop n xs    -- drop.3
```

```
-- se calcula
--    drop 3 [1,2,3,4,5]
-- ----------------------------------------------------------------------

-- El c□lculo es
--      drop 3 [1,2,3,4,5]
--    = drop 2 [2,3,4,5]       { por def. drop.3 }
--    = drop 1 [3,4,5]         { por def. drop.3 }
--    = drop 0 [4,5]           { por def. drop.3 }
--    = [4,5]                  { por def. drop.1 }

-- ----------------------------------------------------------------------
-- Ejercicio 2.3. Mostrar c□mo a partir de la definici□n
--    init :: [a] -> [a]
--    init [_]    = []             -- init.1
--    init (x:xs) = x : init xs    -- init.2
-- se calcula
--    init [1,2,3]
-- ----------------------------------------------------------------------

-- El c□lculo es
--      init [1,2,3]
--    = 1 : (init [2,3])         { por def. init.2 }
--    = 1 : (2 : (init [3]))     { por def. init.2 }
--    = 1 : (2 : [])             { por def. init.1 }
--    = 1 : [2]                  { por def. (:) }
--    = [1,2]                    { por def. (:) }

-- ----------------------------------------------------------------------
-- Ejercicio 3.1.1. Definir por recursi□n la funci□n
--    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
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.1.2. Comprobar con quickCheck que and' es equivalente a
-- and.
-- ---------------------------------------------------------------------

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

-- La comprobaci□n es
--    *Main> quickCheck prop_and
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.2.1. Definir por recursi□n la funci□n
--    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]
concat' []       = []
concat' (xs:xss) = xs ++ concat' xss

-- ---------------------------------------------------------------------
-- Ejercicio 3.2.2. Comprobar con quickCheck que conca' es equivalente a
-- concat.
-- ---------------------------------------------------------------------

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

-- La comprobaci□n es
--    *Main> quickCheck prop_concat
--    +++ OK, passed 100 tests.

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

```haskell
-- Ejercicio 3.3.1. Definir por recursi□n la funci□n
--    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 3.3.2. 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 comprobaci□n es
--    *Main> quickCheck prop_replicate
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.4.1. Definir por recursi□n la funci□n
--    selecciona :: [a] -> Int -> a
-- tal que (selecciona xs n) es el n-□simo 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 3.4.2. Comprobar con quickCheck que selecciona es
-- equivalente a (!!).
```

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

-- 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 s�lo 12 de las pruebas cumplen las condiciones.

-- 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 comprobaci�n es
--    *Main> quickCheck prop_selecciona_3
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.5.1. Definir por recursi�n la funci�n
--    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 3.5.2. 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 comprobaci□n es
--    *Main> quickCheck prop_elem
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por recursi□n la funci□n
--    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 4.2. Definir la funci□n
--    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 4.3. Comprobar con quickCheck que la mezcla de dos listas
-- ordenadas es una lista ordenada.
-- ---------------------------------------------------------------------

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

-- La comprobaci□n es
--    *Main> quickCheck prop_mezcla_ordenada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4,4. Definir la funci□n
--    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

-- Nota: la funci□n borra es la funci□n delete de la librer□a List.

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

```
-- Ejercicio 4.5. Definir la funci□n
--    esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutaci□n de
-- ys. Por ejemplo,
--    esPermutacion [1,2,1] [2,1,1]  ==>  True
--    esPermutacion [1,2,1] [1,2,2]  ==>  False
-- ---------------------------------------------------------------------

-- La definici□n es
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 4.6. Comprobar con QuickCheck que la inversa de una lista
-- es una permutaci□n de la lista.
-- ---------------------------------------------------------------------

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

-- La  comprobaci□n es
--    Main> quickCheck prop_InversaEsPermutacion
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.7. Comprobar con QuickCheck que la mezcla de dos listas
-- es una permutaci□n de su uni□n.
-- ---------------------------------------------------------------------

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

-- La comprobaci□n es
--    quickCheck prop_mezcla_permutacion
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.8. Definir la funci�n
--    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 m�ximo 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

-- ---------------------------------------------------------------------
-- Ejercicio 4.9. Definir la funci�n
--    ordMezcla :: Ord a => [a] -> [a]
-- tal que (ordMezcla xs) es la lista obtenida ordenado xs por mezcla
-- (es decir, considerando que la lista vac�a y las listas unitarias
-- est�n 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 4.10. Comprobar con QuickCheck que la ordenaci�n por mezcla
-- de una lista es una lista ordenada.
-- ---------------------------------------------------------------------

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

-- La comprobaci�n es
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.11. Comprobar con QuickCheck que la ordenaci□n por mezcla
-- de una lista es una permutaci□n de la lista.
-- ---------------------------------------------------------------------

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

-- La comprobaci□n es
--    *Main> quickCheck prop_ordMezcla_permutacion
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, usando el m□todo de los 5 pasos, la funci□n
--    sum' :: [Int] -> Int
-- tal que (sum' xs) es la suma de los n□meros de xs. Por ejemplo,
--    sum' [2,3,5]  =>  10
-- ---------------------------------------------------------------------

-- Paso 1: Definici□n del tipo
-- sum' :: [Int] -> Int

-- Paso 2: Enumeraci□n de los casos
-- sum' :: [Int] -> Int
-- sum' []     =
-- sum' (x:xs) =

-- Paso 3: Definici□n de los casos sencillos
-- sum' :: [Int] -> Int
-- sum' []     = 0
-- sum' (x:xs) =

-- Paso 4: Definici□n del resto de casos
-- sum' :: [Int] -> Int
-- sum' []     = 0
-- sum' (x:xs) = x + sum' xs
```

```haskell
-- Paso 5: Generalizaci□n y simplificaci□n
sum' :: Num a => [a] -> a
sum' = foldr (+) 0

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir, usando el m□todo de los 5 pasos, la funci□n
--    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]
-- ---------------------------------------------------------------------

-- Paso 1: Definici□n del tipo
-- take' :: Int -> [a] -> [a]

-- Paso 2: Enumeraci□n de los casos
-- take' :: Int -> [a] -> [a]
-- take' 0 []         =
-- take' 0 (x:xs)     =
-- take' (n+1) []     =
-- take' (n+1) (x:xs) =

-- Paso 3: Definici□n de los casos sencillos
-- take' :: Int -> [a] -> [a]
-- take' 0 []         = []
-- take' 0 (x:xs)     = []
-- take' (n+1) []     = []
-- take' (n+1) (x:xs) =

-- Paso 4: Definici□n del resto de casos
-- take' :: Int -> [a] -> [a]
-- take' 0 []         = []
-- take' 0 (x:xs)     = []
-- take' (n+1) []     = []
-- take' (n+1) (x:xs) = x : take' n xs

-- Paso 5: Generalizaci□n y simplificaci□n
take' :: Int -> [a] -> [a]
take' 0 _          = []
```

```
take' (n+1) []     = []
take' (n+1) (x:xs) = x : take' n xs

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir, usando el m?todo de los 5 pasos, la funci?n
--    last' :: [a] -> a
-- tal que (last xs) es el ?ltimo elemento de xs. Por ejemplo,
--    last' [2,3,5]  =>  5
-- ---------------------------------------------------------------------

-- Paso 1: Definici?n del tipo
-- last' :: [a] -> a

-- Paso 2: Enumeraci?n de los casos
-- last' :: [a] -> a
-- last' (x:xs) =

-- Paso 3: Definici?n de los casos sencillos
-- last' :: [a] -> a
-- last' (x:xs) | null xs   = x
--              | otherwise =

-- Paso 4: Definici?n del resto de casos
-- last' :: [a] -> a
-- last' (x:xs) | null xs   = x
--              | otherwise = last' xs

-- Paso 5: Generalizaci?n y simplificaci?n
last' :: [a] -> a
last' [x]    = x
last' (_:xs) = last' xs
```

Relación 8

Definiciones por recursión y por comprensión

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

import Data.Char
import Data.List
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]
```

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

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]
```

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

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]
-}

-- ---------------------------------------------------------------------
-- 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,
--    imparesCudradosC [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,
--    imparesCudradosR [1,2,3]  ==>  [1,9]
-- ---------------------------------------------------------------------
```

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

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

{-
     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]
-}

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

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

entreR :: Integer -> Integer -> [Integer]
entreR m n | m > n    = []
           | m <= n   = 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]
```

```haskell
-- ---------------------------------------------------------------------
-- 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
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 menores 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]
```

```haskell
-- ---------------------------------------------------------------------
-- 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 menores 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.
-- ---------------------------------------------------------------------

-- 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
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 23. 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
```

```haskell
agarrado ps = sum [p * 0.9 | p <- ps, p * 0.9 <= 199]

-- ---------------------------------------------------------------------
-- Ejercicio 24. 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, 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 25. 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 26. 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.
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 27. Definir, por recursión, la función
--    sumaDigitosRec :: String -> Int
-- tal que (sumaDigitosRec xs) es la suma de los dígitos de la cadena
-- 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 28. 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 29. 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"
```

```haskell
-- Nota: Usar las funciones toLower y toUpper.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 30. Definir, por recursión, la función
--    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 31. 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 32. 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
```

```haskell
--      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"]
-- ---------------------------------------------------------------------

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 33. 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 34. Comprobar con QuickCheck que ambas definiciones son
```

```haskell
-- 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 35. 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 tienn longitud lon y posen 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 36. Definir, por comprensión, 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", "casa", "ocupado"]
--    ["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 37. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- 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 38. 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 39. 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 40. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- 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 41. 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 42. Definir, por comprensión, la función
--    contiene :: String -> String -> Bool
-- tal que (contiene xs ys) se verifica si ys es una subcadena de
```

```haskell
-- xs. Por ejemplo,
--    contiene "escasamente" "casa"   ==>  True
--    contiene "escasamente" "cante"  ==>  False
--    contiene "" ""                  ==>  True
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica
-- si ys es un prefijo de xs.
-- ---------------------------------------------------------------------

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 43. 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.
```

Relación 9

Funciones de orden superior

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

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

-- ---------------------------------------------------------------------------
-- Ejercicio 1. La función
--    divideMedia :: [Double] -> ([Double],[Double])
-- dada una lista numérica, xs, calcula la lista [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 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
```

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

-- La propiedad es
prop_divideMedia :: [Double] -> Bool
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 3. Comprobar con QuickCheck que si (ys,zs) es el par
```

```haskell
-- 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 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.

-- ---------------------------------------------------------------------
-- Ejercicio 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
```

```haskell
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.

-- ---------------------------------------------------------------------
-- Ejercicio 6. 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]

-- 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

-- ---------------------------------------------------------------------
-- Ejercicio 7. 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]
-- ---------------------------------------------------------------------

-- La definición es
takeWhile' :: (a -> Bool) -> [a] -> [a]
takeWhile' _ [] = []
takeWhile' p (x:xs)
    | p x       = x : takeWhile' p xs
    | otherwise = []

-- ---------------------------------------------------------------------
-- Ejercicio 8. 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]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 9. 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 es
concat' :: [[a]] -> [a]
concat' = foldr (++) []
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 10. Redefinir, usando foldr, la función map. Por ejemplo,
--    map' (+2) [1,7,3]  =>  [3,9,5]
-- ---------------------------------------------------------------------

-- La definición por recursión

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

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


-- ---------------------------------------------------------------------
-- Ejercicio 11. Redefinir, usando foldr, la función filter. Por
-- ejemplo,
--    filter' (<4) [1,7,3,2]  =>  [1,3,2]
-- ---------------------------------------------------------------------

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

Relación 10

Plegados

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

import Data.Char
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Redefinir, usando foldr, la función maximum.
-- ---------------------------------------------------------------------

maximum' :: (Ord a) => [a] -> a
maximum' (x:xs) = foldr max x xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Redefinir, usando foldr, la función minimum.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- 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]
-- ---------------------------------------------------------------------
```

```haskell
inversaFR :: [a] -> [a]
inversaFR = foldr f []
            where f x xs = xs ++ [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]
-- ---------------------------------------------------------------------

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 (\x y -> 10*x+y) 0

dec2entF' :: [Int] -> Int
dec2entF' = foldl f 0
            where f x y = 10*x+y
-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir, mediante plegado, la función
```

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

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 xs y) es la lista obtenida borrando la primera
-- ocurrencia de y en xs. Por ejemplo,
--    borra [2,3,5,6] 5    =>  [2,3,6]
--    borra [2,3,5,6,5] 5  =>  [2,3,6,5]
--    borra [2,3,5,6,5] 7  =>  [2,3,5,6,5]
-- ---------------------------------------------------------------------

borra :: Eq a => [a] -> a -> [a]
borra xs y = foldr f [] xs
             where f z zs | z == y    = zs
                          | otherwise = z:zs

-- ---------------------------------------------------------------------
-- 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]
-- ---------------------------------------------------------------------

diferencia :: Eq a => [a] -> [a] -> [a]
diferencia xs ys = foldl borra xs ys

-- La definición anterior puede simplificarse a
diferencia' :: Eq a => [a] -> [a] -> [a]
diferencia' = foldl 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
--    int2bin = unfold (== 0) (`mod` 2) (`div` 2)
```

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

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

```haskell
map'' f = unfold null (f . head) tail

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

```haskell
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
-- 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]]
-- ---------------------------------------------------------------------
```

```haskell
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 cao 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
```

```haskell
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
```

Relación 11

Modelización de un juego de cartas

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

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

-- ------------------------------------------------------------------------
-- Ejercicio 1 (Modelización de un juego de cartas)
-- ------------------------------------------------------------------------
-- 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.
-- ------------------------------------------------------------------------

-- 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 1.1. 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 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
```

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

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 1.2. 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,
--    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
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.3. 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.

-- ---------------------------------------------------------------------
-- 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)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.4. 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 1.5. 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 1.6. 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 1.7. 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 1.8. 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 1.9. 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.
```

```haskell
-- * 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

-- ---------------------------------------------------------------------
-- Ejercicio 1.10. 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 12

Tipos de datos algebraicos

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

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- 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)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. 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. Usando esta función, redefinir la función
--    ocurre :: Int -> Arbol -> Bool
-- definida en las transparencias del tema 9. ¿Porqué esta definición es
-- más eficiente que la original?
-- ---------------------------------------------------------------------

data Arbol = Hoja Int | Nodo Arbol Int Arbol

ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))

-- La definición en las transparencias es
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d

-- La nueva definición es
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

-- 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.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Se considera el siguiente tipo de árboles binarios
--    type ArbolB = Hoja Int | Nodo Arbol Arbol deriving Show
-- Definir la función
--    nHojas :: ArbolB -> Int
```

```haskell
-- tal que (nHojas a) es el número de hojas del árbol a. Por ejemplo,
--    nHojas (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7)))  =>  3
-- ---------------------------------------------------------------------

data ArbolB = HojaB Int
            | NodoB ArbolB ArbolB
            deriving Show

nHojas :: ArbolB -> Int
nHojas (HojaB _)     = 1
nHojas (NodoB a1 a2) = nHojas a1 + nHojas a2

-- ---------------------------------------------------------------------
-- Ejercicio 3.2, 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,
--    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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. 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])
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. 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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. 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
```

```haskell
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
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]

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

```haskell
-- Ejercicio 5.2. 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

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. 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)

-- ---------------------------------------------------------------------
-- Ejercicio 6. 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,
--    sumaMultiplosMenores 10  =>  23
```

```haskell
-- 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)
```

Relación 13

Listas infinitas (1)

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, usando takeWhile y map, la función
--    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..])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir por recursión 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.
-- ---------------------------------------------------------------------
```

```haskell
repite :: a -> [a]
repite x = x : repite x

-- Por comprensión:
repite' x = [x| _ <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir por recursión 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.
-- ---------------------------------------------------------------------

repiteFinita :: Int -> a -> [a]
repiteFinita 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
--    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 consecutivas 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])
--    ==> [[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.

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

```haskell
--    siguiente 40  =>  20
-- ---------------------------------------------------------------------

siguiente n | even n    = n `div` 2
            | otherwise = 3*n+1

-- ---------------------------------------------------------------------
-- 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]
```

```haskell
-- ---------------------------------------------------------------------
-- 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,
--    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]

-- ---------------------------------------------------------------------
-- 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 13. 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 14. La función
--    repeat :: a -> [a]
-- está definida de forma que (repeat x) es la lista formada por
-- infinitos elementos x. Por ejemplo,
--    repeat 3  ==>  [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 15. 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
```

```haskell
-- 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)
-- ------------------------------------------------------------------------

replicateArbol :: Int -> a -> Arbol a
replicateArbol n = takeArbol n . repeatArbol
```

Relación 14

Listas infinitas (2)

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. 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
--    euler1 :: Integer -> Integer
-- (euler1 n) es la suma de todos los múltiplos de 3 ó 5 menores que
-- n. Por ejemplo,
--    euler1 10  =>  23
--
-- Escribir distintas definiciones y comparar su eficiencia al calcular
-- (euler1 1000).
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- 1ª solución                                                         --
-- ---------------------------------------------------------------------

-- Definición:
euler1a :: Integer -> Integer
euler1a n = sum [x | x <- [1..n-1], multiplo x 3 || multiplo x 5]
```

```
    where multiplo x y = mod x y == 0

-- Cálculo:
--    *Main> euler1a 1000
--    233168

-- ---------------------------------------------------------------------
-- 2ª solución                                                         --
-- ---------------------------------------------------------------------

-- Definición con gcd
euler1b :: Integer -> Integer
euler1b n = sum [x | x <- [1..n-1], gcd x 15 > 1]

-- ---------------------------------------------------------------------
-- 3ª solución                                                         --
-- ---------------------------------------------------------------------

-- Definición con progresiones:
euler1c :: Integer -> Integer
euler1c n = sum [3,6..n-1] + sum [5,10..n-1] - sum [15,30..n-1]

-- ---------------------------------------------------------------------
-- 4ª solución                                                         --
-- ---------------------------------------------------------------------

-- Definición con progresiones y nub:
euler1d :: Integer -> Integer
euler1d n = sum (nub ([3,6..n-1] ++ [5,10..n-1]))

-- ---------------------------------------------------------------------
-- 5ª solución                                                         --
-- ---------------------------------------------------------------------

-- Definición con progresiones y union:
euler1e :: Integer -> Integer
euler1e n = sum ([3,6..n-1] `union` [5,10..n-1])

-- ---------------------------------------------------------------------
-- Comparación de soluciones                                           --
```

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

--    +------------+------+-----------+
--    | Definición | segs | bytes     |
--    +------------+------+-----------+
--    | a          | 0.02 |   526.116 |
--    | b          | 0.04 | 3.657.176 |
--    | c          | 0.03 | 3.080.384 |
--    | d          | 0.03 | 3.080.384 |
--    | e          | 0.01 | 1.606.708 |
--    +------------+------+-----------+

-- ---------------------------------------------------------------------
-- Ejercicio 2. Cada término de la sucesión de Fibonacci se obtiene
-- sumando los dos anteriores. Comenzando con 1 y 2, los 10 primeros
-- términos son
--    1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
--
-- Definir la función
--    euler2 :: Integer -> Integer
-- tal que (euler2 n) es la suma de los términos pares (2, 8, 34, ...)
-- de la sucesión de Fibonacci que sean menores que n. Por ejemplo,
--    euler2 35  =>  44
--
-- Escribir distintas definiciones y comparar su eficiencia al calcular
-- (euler2 4000000).
-- ---------------------------------------------------------------------

euler2 :: Integer -> Integer
euler2 n = sum [x | x <- takeWhile (< n) fibs, even x]

-- fibs es la sucesión de los números de Fibonacci. Por ejemplo,
--    take 10 fibs  =>  [1,2,3,5,8,13,21,34,55,89]
fibs = 1 : 2 : [a+b | (a,b) <- zip fibs (tail fibs)]

-- Eficiencia:
--    *Main> :s +s
--    *Main> euler2 4000000
--    4613732
--    (0.02 secs, 524952 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3. Los factores primos de 13195 son 5, 7, 13 y 29. Definir la
-- función
--    euler3 :: Integer -> Integer
-- tal que (euler3 n) es el mayor factor primo de n. Por ejemplo,
--    euler3 13195  =>  29
--
-- Escribir distintas definiciones y comparar su eficiencia al calcular
-- (euler3 600851475143).
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Primera definición                                                  --
-- ---------------------------------------------------------------------

-- (euler3a n) es el mayor factor primo de n. Por ejemplo,
--    euler3a 13195  =>  29
euler3a :: Integer -> Integer
euler3a n = last (factores n)

-- (factores n) es la lista de los factores primos de n. Por ejemplo,
--    factores 13195  =>  [5,7,13,29]
factores :: Integer -> [Integer]
factores 0 = []
factores 1 = []
factores n = m : factores (n `div` m)
    where m = menorFactor n

-- (menorFactor n) es el menor factor de n. Por ejemplo,
--    menorFactor 13195  =>  5
menorFactor :: Integer -> Integer
menorFactor x = menorFactor' x 2
    where menorFactor' x y | x `mod` y == 0 = y
                           | y^2 > x        = x
                           | otherwise      = menorFactor' x (y+1)

-- ---------------------------------------------------------------------
-- Segunda solución                                                    --
-- ---------------------------------------------------------------------
```

```haskell
-- (euler3b n) es el mayor factor primo de n usando en el
-- cálculo la lista de los número primos obtenida mediante la criba de
-- Erastótenes. Por ejemplo,
--    euler3b 35  =>  7
euler3b :: Integer -> Integer
euler3b n = euler3b' n primos
    where euler3b' n (x:xs)
              | x^2 > n      = n
              | mod n x == 0 = euler3b' (div n x) (x:xs)
              | otherwise    = euler3b' n xs

-- 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]

-- ---------------------------------------------------------------------
-- Comparación de las definiciones                                     --
-- ---------------------------------------------------------------------

-- Las definiciones son equivalentes
prop_equivalentes n =
    n > 1 ==> euler3a n == euler3b n

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

-- Resumen:
--    +------------+------+-----------+
--    | Definición | segs | bytes     |
--    +------------+------+-----------+
--    | a          | 0.03 |   746.916 |
--    | b          | 0.09 | 3.526.856 |
--    +------------+------+-----------+

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

```haskell
-- Ejercicio 4. Un capicúa es un número que es igual leído de izquierda
-- a derecha que de derecha a izquierda. El mayor capicúa formado por el
-- producto de dos números de 2 cifras es 9009 = 91 × 99.
--
-- Definir la función
--    euler4 :: Integer -> Integer
-- tal que (euler4 n) es mayor capicúa que es el producto de dos números
-- de n cifras. Por ejemplo,
--    euler4 2  =>  9009
--
-- Escribir distintas definiciones y comparar su eficiencia al calcular
-- (euler4 3).
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Primera solución                                                    --
-- ---------------------------------------------------------------------

euler4a :: Integer -> Integer
euler4a n = head (problemaCapicuas n)

-- (problemaCapicuas n) es la lista de las capicúas de 2*n cifras que
-- pueden escribirse como productos de dos números de n cifras. Por
-- ejemplo, Por ejemplo,
--    *Main> problemaCapicuas 2
--    [9009,8448,8118,8008,7227,7007,6776,6336,6006,5775,5445,5335,
--     5225,5115,5005,4884,4774,4664,4554,4224,4004,3773,3663,3003,
--     2992,2772,2552,2442,2332,2112,2002,1881,1771,1551,1221,1001]
problemaCapicuas n = [x | x <- capicuas n,
                          not (null (productosDosNumerosCifras n x))]

-- (capicua n) es la capicúa formada anadiendo el inverso de n a
--  continuación de n. Por ejemplo,
--    capicua 93  =>  9339
capicua :: Integer -> Integer
capicua n = read (xs ++ (reverse xs))
    where xs = show n

-- (capicuas n) es la lista de las capicúas de 2*n cifras de mayor a
-- menor. Por ejemplo,
```

```
--    *Main> capicuas 1
--    [99,88,77,66,55,44,33,22,11]
--    *Main> capicuas 2
--    [9999,9889,9779,9669,9559,9449,9339,9229,9119,9009,8998,...
capicuas :: Integer -> [Integer]
capicuas n = [capicua x | x <- numerosCifras n]

-- (numerosCifras n) es la lista de los números de n cifras de mayor a
-- menor. Por ejemplo,
--    *Main> numerosCifras 1
--    [9,8,7,6,5,4,3,2,1]
--    *Main> numerosCifras 2
--    [99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,...
numerosCifras :: Integer -> [Integer]
numerosCifras n = [a,a-1..b]
    where a = 10^n-1
          b = 10^(n-1)

-- (productosDosNumerosCifras n x) es la lista de los números y de n
-- cifras tales que existe un z de n cifras y x es el producto de y por
-- z. Por ejemplo,
--    productosDosNumerosCifras 2 9009  =>  [99,91]
productosDosNumerosCifras n x = [y | y <- numeros,
                                     mod x y == 0,
                                     elem (div x y) numeros]
    where numeros = numerosCifras n

-- ---------------------------------------------------------------------
-- Segunda solución                                                    --
-- ---------------------------------------------------------------------

euler4b n = maximum [x*y | x <- [a,a-1..b],
                           y <- [a,a-1..b],
                           esCapicua (x*y)]
            where a = 10^n-1
                  b = 10^(n-1)

-- (esCapicua x) se verifica si x es capicua. Por ejemplo,
--    esCapicua 353  =>  True
--    esCapicua 357  =>  False
```

```haskell
esCapicua :: Integer -> Bool
esCapicua n = xs == reverse xs
    where xs = show n

-- ---------------------------------------------------------------------
-- Tercera solución                                                    --
-- ---------------------------------------------------------------------

euler4c :: Integer -> Integer
euler4c n = maximum [x*y | (x,y) <- pares a b,
                           esCapicua (x*y)]
            where a = 10^n-1
                  b = 10^(n-1)

-- (pares a b) es la lista de los pares de números entre a y b de forma
-- que su suma es decreciente. Por ejemplo,
--    pares 9 7  =>  [(9,9),(8,9),(8,8),(7,9),(7,8),(7,7)]
pares a b = [(x,z-x) | z <- [a',a'-1..b'],
                       x <- [a,a-1..b],
                       x <= z-x, z-x <= a]
            where a' = 2*a
                  b' = 2*b

-- ---------------------------------------------------------------------
-- Cuarta solución                                                     --
-- ---------------------------------------------------------------------

euler4d :: Integer -> Integer
euler4d n = maximum [x | y <- [b..a],
                         z <- [y..a],
                         let x = y * z,
                         let s = show x,
                         s == reverse s]
            where a = 10^n-1
                  b = 10^(n-1)

-- ---------------------------------------------------------------------
-- Quinta solución                                                     --
-- ---------------------------------------------------------------------
```

```haskell
euler4e :: Integer -> Integer
euler4e n = head [x*y | (x,y) <- pares a b, esCapicua (x*y)]
            where a = 10^n-1
                  b = 10^(n-1)

-- (pares' a b) es la lista de los pares de números entre a y b de forma
-- que su suma es decreciente. Por ejemplo,
--    pares' 9 7  =>  [(9,9),(8,9),(8,8),(7,9),(7,8),(7,7)]
pares' a b = [(x,y) | x <- [a,a-1..b], y <- [a,a-1..x]]

-- ---------------------------------------------------------------------
-- Sexta solución                                                      --
-- ---------------------------------------------------------------------

euler4f :: Integer -> Integer
euler4f n = maximum [ x*y | x<-[b..a],
                            y<-[x..a] ,
                            esCapicua' (x*y)]
            where a = 10^n-1
                  b = 10^(n-1)

-- (esCapicua' x) se verifica si x es capicua. Por ejemplo,
--    esCapicua 353  =>  True
--    esCapicua 357  =>  False
esCapicua' n = x == reverse x
    where x = cifras n

-- (cifras n) es la lista de las cifras de n en orden inverso. Por
-- ejemplo,
--    cifras 325  => [5,2,3]
cifras :: Integer -> [Integer]
cifras n
    | n < 10    = [n]
    | otherwise = (ultima n) : (cifras (quitarUltima n))

-- (ultima n) es la última cifra de n. Por ejemplo,
--    ultima 325  =>  5
ultima  :: Integer -> Integer
ultima n =  n - (n `div` 10)*10
```

```haskell
-- (quitarUltima n) es el número obtenido al quitarle a n su última
-- cifra. Por ejemplo,
--    quitarUltima 325  =>  32
quitarUltima :: Integer -> Integer
quitarUltima n = (n - (ultima n)) `div` 10

-- ---------------------------------------------------------------------
-- Comparación de soluciones                                           --
-- ---------------------------------------------------------------------

--    +------------+-------+-------------+
--    | Definición | segs  | bytes       |
--    +------------+-------+-------------+
--    | a          |  0.21 |  12.270.928 |
--    | b          |  3.20 | 286.109.728 |
--    | c          |  6.20 | 423.331.068 |
--    | d          |  1.52 | 135.002.928 |
--    | e          |  0.30 |  17.216.760 |
--    | f          | 19.73 | 991.403.840 |
--    +------------+-------+-------------+

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    euler5 :: Integer -> Integer -> Integer
-- tal que (euler5 a b) es el menor número divisible por los
-- números desde a hasta b. Por ejemplo,
--    euler5 1 10  =>  2520
--
-- Escribir distintas definiciones y comparar su eficiencia al calcular
-- (euler5 1 20).
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Primera solución                                                    --
-- ---------------------------------------------------------------------

-- (euler5a a b) es el menor número divisible por los números del
-- a al b. Por ejemplo,
--    euler5a 2 5  =>  60
-- (euler5a a b) es el mínimo común múltiplo de los números de a
```

```haskell
-- a b.
euler5a :: Integer -> Integer -> Integer
euler5a a b
    | b == 1    = a
    | otherwise = euler5a (lcm a b) (b-1)

-- ---------------------------------------------------------------------
-- Segunda definición                                                  --
-- ---------------------------------------------------------------------

euler5b :: Integer -> Integer -> Integer
euler5b a b = foldl lcm 1 [a..b]

euler5b':: Integer -> Integer -> Integer
euler5b' a b = foldr (lcm) 1 [a..b]

-- ---------------------------------------------------------------------
-- Comparación de soluciones                                           --
-- ---------------------------------------------------------------------

--    +------------+------+-----------+
--    | Definición | segs | bytes     |
--    +------------+------+-----------+
--    | a          | 0.02 | 3.082.204 |
--    | b          | 0.02 | 3.077.052 |
--    +------------+------+-----------+
```

Relación 15

Razonamiento sobre programas (1)

```
-- ---------------------------------------------------------------------
-- Importaci□n de librer□as auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1a. Definir por recursi□n la funci□n
--    sumaImpares :: Int -> Int
-- tal que (sumaImpares n) es la suma de los n primeros n□meros
-- impares. Por ejemplo,
--    *Main> sumaImpares 5  =>  25
-- ---------------------------------------------------------------------

sumaImpares :: Int -> Int
sumaImpares 0     = 0
sumaImpares (n+1) = (sumaImpares n) + (2*n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 1b. Definir, sin usar recursi□n, la funci□n
--    sumaImpares' :: Int -> Int
-- tal que (sumaImpares' n) es la suma de los n primeros n□meros
-- impares. Por ejemplo,
--    *Main> sumaImpares' 5  =>  25
-- ---------------------------------------------------------------------
```

```haskell
sumaImpares' :: Int -> Int
sumaImpares' n = sum [1,3..(2*n-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 1c. Definir la funci□n
--    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 n□meros x entre 1 y 100.
-- ---------------------------------------------------------------------

-- La definici□n es
sumaImparesIguales :: Int -> Int -> Bool
sumaImparesIguales m n =
    and [sumaImpares x == sumaImpares' x | x <- [m..n]]

-- La comprobaci□n es
--    *Main>  sumaImparesIguales 1 100
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1d. Definir la funci□n
--    grafoSumaImpares :: Int -> Int -> [(Int,Int)]
-- tal que (grafoSumaImpares m n) es la lista formadas por los n□meros x
-- entre m y n y los valores de (sumaImpares x).
--
-- Calcular (grafoSumaImpares 1 9).
-- ---------------------------------------------------------------------

-- La definici□n es
grafoSumaImpares :: Int -> Int -> [(Int,Int)]
grafoSumaImpares m n =
    [(x,sumaImpares x) | x <- [m..n]]

-- El c□lculo 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 inducci□n 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 hip□tesis]
    = 0               [por sumaImpares.1]
    = 0^2             [por aritm□tica]

  Caso inductivo: Se supone la hip□tesis de inducci□n (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 □lgebra]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2a. Definir por recursi□n la funci□n
--    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 comprensi□n la funci□n
--    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 inducci□n 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 aritm□tica]

  Caso inductivo: Se supone la hip□tesis de inducci□n (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 aritm□tica]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3a. Definir por recursi□n la funci□n
--    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]
-- ---------------------------------------------------------------------
```

```haskell
copia :: Int -> a -> [a]
copia 0 _     = []              -- copia.1
copia (n+1) x = x : copia n x   -- copia.2

-- ---------------------------------------------------------------------
-- Ejercicio 3b. Definir por recursi□n la funci□n
--    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
prop_copia :: Eq a => Int -> a -> Bool
prop_copia n x =
    todos (==x) (copia n' x)
    where n' = abs n

-- La comprobaci□n es
--    *Main> quickCheck prop_copia
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3d. Demostrar, por inducci□n 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 hip□tesis de inducci□n (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 funci□n
--    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 xs = foldr ((&&) . p) True xs

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la funci□n
--    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]]
```

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

traspuesta :: [[a]] -> [[a]]
traspuesta []           = []
traspuesta ([]:xss)     = traspuesta xss
traspuesta ((x:xs):xss) =
    (x:[h | (h:_) <- xss]) : traspuesta (xs : [t | (_:t) <- xss])
```

Relación 16

Razonamiento sobre programas (2)

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

import Test.QuickCheck
import Data.List (delete)

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

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

-- 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,
```

```haskell
--    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.

-- ---------------------------------------------------------------------
-- 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]
-}

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

```haskell
-- ---------------------------------------------------------------------
-- 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]
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
```

```haskell
--    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 []            = error "no tiene valor en la tabla"
busca x ((x',y):xys)
    | x == x'         = y
    | otherwise       = busca 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
-- 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 []           = Nothing
busca' x ((x',y):xys)
    | x == x'         = Just y
    | otherwise       = busca' 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.
```

```haskell
-- ---------------------------------------------------------------------
-- 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
-- ceiling (sum [fromIntegral e | (p,e) <- xs] / 2)

-- ---------------------------------------------------------------------
-- 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)]

-- ---------------------------------------------------------------------
-- 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 [e | (p,e) <- a, p `elem` c] >= mayoria a

-- ---------------------------------------------------------------------
-- Ejercicio 3.16. Comprobar con QuickCheck que las coaliciones
-- obtenidas por (mayorias asamblea) son coaliciones mayoritarias en la
-- asamblea.
-- ---------------------------------------------------------------------

-- 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
```

```haskell
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]]
-- ---------------------------------------------------------------------

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]
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)]
```

```haskell
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 17

Razonamiento sobre programas (3)

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

```
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 18

Ecuación con factoriales

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- 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.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- 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]
-- 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
-- ---------------------------------------------------------------------

invFactorial :: Integer -> Maybe Integer
```

```haskell
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
-- única solución de la ecuación
--    a! * b! = a! + b! + c!
```

```haskell
-- 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 19

Razonamiento sobre programas (4)

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

import Test.QuickCheck
import Control.Monad

-- ---------------------------------------------------------------------
-- Ejercicio 1. La definición de la composición de funciones es
--    (.) :: (b -> c) -> (a -> b) -> (a -> c)
--    (f . g) x = f (g x)
-- y la de aplicación de una función a los elementos de una lista es
--    map :: (a -> b) -> [a] -> [b]
--    map _ []     = []
--    map f (x:xs) = f x : map f xs
-- Demostrar que para todo par de funciones f, g se tiene que
--    (map f). (map g) = map (f . g)
-- ---------------------------------------------------------------------

{-
 Por extensionalidad, tenemos que demostrar que para todo lista xs se
 tiene que
    ((map f). (map g)) xs = map (f . g) xs
 Lo demostraremos por inducción en xs.

 Caso base: Hay que demostrar que
```

```
    ((map f). (map g)) [] = map (f . g) []
 En efecto,
    ((map f). (map g)) []
    = map f (map g [])     [por (.)]
    = map f []             [por map.1]
    = []                   [por map.1]
    = map (f . g) []       [por map.1]

 Caso inductivo: Suponemos la hipótesis de inducción
    ((map f). (map g)) xs = map (f . g) xs
 tenemos que demostrar que
    ((map f). (map g)) (x:xs) = map (f . g) (x:xs)
 En efecto,
    ((map f). (map g)) (x:xs)
    = map f (map g (x:xs))                [por (.)]
    = map f (g x : map g xs)              [por map.2]
    = f (g x) : map f (map g xs)          [por map.2]
    = (f . g) x : map f (map g xs)        [por (.)]
    = (f . g) x : ((map f) . (map g)) xs  [por (.)]
    = (f . g) x : (map (f . g)) xs        [por hip. de inducción]
    = map (f . g) (x:xs)                  [por map.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que para todo función f y todo par de lista xs
-- e ys se tiene que
--    map f (xs ++ ys) = (map f xs) ++ (map f ys)
-- ---------------------------------------------------------------------

{-
 Demostración: Por inducción en xs.
 Caso base: Hay que demostrar que
    map f ([] ++ ys) = (map f []) ++ (map f ys)
 En efecto,
    map f ([] ++ ys)
    = map f ys                  [por ++.1]
    = [] ++ (map f ys)          [por ++.1]
    = (map f []) ++ (map f ys)  [por map.1]

 Caso inductivo: Suponemos la hipótesis de inducción
```

```
    map f (xs ++ ys) = (map f xs) ++ (map f ys)
 Hay que demostrar que
    map f ((x:xs) ++ ys) = (map f (x:xs)) ++ (map f ys)
 En efecto,
    map f ((x:xs) ++ ys)
    = map f (x : (xs ++ ys))              [por ++.2]
    = f x : (map f (xs ++ ys))            [por map.2]
    = f x : ((map f xs) ++ (map f ys))    [por hip. de inducción]
    = (f x : (map f xs)) ++ (map f ys)    [por ++.2]
    = (map f (x:xs)) ++ (map f ys)        [por map.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que para todo función f
--    (map f) . reverse = reverse . (map f)
-- ---------------------------------------------------------------------

{-
 Demostración: Hay que demostrar que para toda lista xs se tiene que
    ((map f) . reverse) xs = (reverse . (map f)) xs
 Lo demostraremos por inducción en xs.

 Caso base: Hay que demostrar que
    ((map f) . reverse) [] = (reverse . (map f)) []
 En efecto, reduciendo el término izquierdo
    ((map f) . reverse) []
    = map f (reverse [])      [por .]
    = map f []                [por reverse.1]
    = []                      [por map.1]
 y reduciendo el término derecho,
    (reverse . (map f)) []
    = reverse (map f [])      [por .]
    = reverse []              [por map.1]
    = []                      [por reverse.1]
 Por tanto,
    ((map f) . reverse) [] = (reverse . (map f)) []

 Caso inductivo: Suponemos la hipótesis de inducción
    ((map f) . reverse) xs = (reverse . (map f)) xs
 Hay que demostrar que
```

```
    ((map f) . reverse) (x:xs) = (reverse . (map f)) (x:xs)
 En efecto,
    ((map f) . reverse) (x:xs)
    = map f (reverse (x:xs))                [por .]
    = map f (reverse xs ++ [x])             [por reverse.2]
    = (map f (reverse xs)) ++ (map f [x])   [por Ejercicio 2]
    = (map f (reverse xs)) ++ [f x]         [por map]
    = (((map f) . reverse) xs) ++ [f x]     [por .]
    = ((reverse . (map f)) xs) ++ [f x]     [por hip. de inducción]
    = (reverse (map f xs)) ++ [f x]         [por .]
    = reverse (f x : (map f xs))            [por reverse.2]
    = reverse (map f (x:xs))                [por map.2]
-}

-- ---------------------------------------------------------------------
-- Nota. 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 siguinete 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)
  coarbitrary = undefined

-- ------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    maptree :: (a -> a) -> Arbol a -> Arbol a
-- tal que (maptree f x) es el árbol obtenido aplicándole a cada nodo de
-- x la función f. Por ejemplo,
--    *Main> arbol
--    Nodo 9
--         (Nodo 3
--               (Nodo 2 Hoja Hoja)
--               (Nodo 4 Hoja Hoja))
--         (Nodo 7 Hoja Hoja)
--    *Main> maptree (2+) arbol
--    Nodo 11
--         (Nodo 5
--               (Nodo 4 Hoja Hoja)
--               (Nodo 6 Hoja Hoja))
--         (Nodo 9 Hoja Hoja)
-- ------------------------------------------------------------------

maptree :: (a -> a) -> Arbol a -> Arbol a
maptree f Hoja         = Hoja
maptree f (Nodo x i d) = Nodo (f x) (maptree f i) (maptree f d)

-- ------------------------------------------------------------------
-- Ejercicio 4.2. La función
--    espejo :: Arbol a -> Arbol a
--    espejo Hoja         = Hoja
--    espejo (Nodo x i d) = Nodo x (espejo d) (espejo i)
```

```haskell
-- es 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))
-- Comprobar con QuickCheck que
--     (maptree (1+)) . espejo = espejo . (maptree (1+))
-- ---------------------------------------------------------------------

espejo :: Arbol a -> Arbol a
espejo Hoja         = Hoja
espejo (Nodo x i d) = Nodo x (espejo d) (espejo i)

-- La propiedad es
prop_maptree_espejo :: Arbol Int -> Bool
prop_maptree_espejo x =
    ((maptree (1+)) . espejo) x == (espejo . (maptree (1+))) x

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Demostrar por inducción que
--    (maptree f) . espejo = espejo . (maptree f)
-- ---------------------------------------------------------------------

{-
 Demostración: Hay que demostrar que para todo árbol binario x
    ((maptree f) . espejo) x = (espejo . (maptree f)) x
 Lo demostraremos por inducción en x.

 Caso base: Hay que demostrar que
    ((maptree f) . espejo) Hoja = (espejo . (maptree f)) Hoja
 En efecto, reduciendo el término izquierdo
    ((maptree f) . espejo) Hoja
    = maptree f (espejo Hoja)     [por .]
    = maptree f Hoja              [por espejo.1]
```

```
    = Hoja                          [por maptree.1]
 y reducciendo el término derecho
    (espejo . (maptree f)) Hoja
    = espejo (maptree f Hoja)       [por .]
    = espejo Hoja                   [por maptree.1]
    = Hoja                          [por espejo.1]
 Por tanto,
    ((maptree f) . espejo) Hoja = (espejo . (maptree f)) Hoja

 Caso inductivo: Suponemos la hipótesis de inducción
    ((maptree f) . espejo) i = (espejo . (maptree f)) i
    ((maptree f) . espejo) d = (espejo . (maptree f)) d
 Hay que demostrar que
    ((maptree f) . espejo) (Nodo x i d) =
    (espejo . (maptree f)) (Nodo x i d)
 En efecto,
    ((maptree f) . espejo) (Nodo x i d)
    = maptree f (espejo (Nodo x i d))
         [por .]
    = maptree f (Nodo x (espejo d) (espejo i))
         [por espejo.2]
    = Nodo (f x) (maptree f (espejo d)) (maptree f (espejo i)
         [por maptree.2]
    = Nodo (f x) (((maptree f) . espejo) d) (((maptree f) . espejo) i)
         [por .]
    = Nodo (f x) ((espejo . (maptree f)) d) ((espejo . (maptree f)) i)
         [por hip. de inducción]
    = Nodo (f x) (espejo (maptree f d)) (espejo (maptree f i))
         [por .]
    = espejo (Nodo (f x) (maptree f i) (maptree f d))
         [por espejo.2]
    = espejo (maptree f (Nodo x i d))
         [por maptree.2]
    = (espejo . (maptree f)) (Nodo x i d)
         [por .]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 4.5. La función
--    preorden :: Arbol a -> [a]
```

```
--    preorden Hoja         = []
--    preorden (Nodo x i d) = x : (preorden i ++ preorden d)
-- es 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]
-- Comprobar con QuickCheck que
--    (map (1+)) . preorden = preorden . (maptree (1+))
-- ---------------------------------------------------------------------

preorden :: Arbol a -> [a]
preorden Hoja         = []
preorden (Nodo x i d) = x : (preorden i ++ preorden d)

-- La propiedad es
prop_map_preorden :: Arbol Int -> Bool
prop_map_preorden x =
    ((map (1+)) . preorden) x == (preorden . (maptree (1+))) x

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.6. Demostrar que
--    (map f) . preorden = preorden . (maptree f)
-- ---------------------------------------------------------------------

{-
 Demostración: Hay que demostrar que para todo árbol binario x
    ((map f) . preorden) x = (preorden . (maptree f)) x
 Lo demostraremos por inducción en x.

 Caso base: Hay que demostrar que
    ((map f) . preorden) Hoja = (preorden . (maptree f)) Hoja
 Reduciendo el término izquierdo
```

```
   ((map f) . preorden) Hoja
   = map f (preorden Hoja)      [por .]
   = map f []                   [por preorden.1]
   = []                         [por map.1]
Reduciendo el término derecho,
   (preorden . (maptree f)) Hoja
   = preorden (maptree f Hoja)   [por .]
   = preorden Hoja               [por maptree.1]
   = []                          [por preorden.1]
Por tanto,
   ((map f) . preorden) Hoja = (preorden . (maptree f)) Hoja

Caso inductivo: Suponemos la hipótesis de inducción
   ((map f) . preorden) i = (preorden . (maptree f)) i
   ((map f) . preorden) d = (preorden . (maptree f)) d
Hay que demostrar que
   ((map f) . preorden) (Nodo x i d) =
   (preorden . (maptree f)) (Nodo x i d)
 En efecto,
   ((map f) . preorden) (Nodo x i d)
   = map f (preorden (Nodo x i d))
        [por .]
   = map f (x : (preorden i ++ preorden d))
        [por preorden.2]
   = f x : (map f (preorden i ++ preorden d))
        [por map.2]
   = f x : ((map f (preorden i)) ++ (map f (preorden d)))
        [por Ejercicio 2]
   = f x : ((((map f) . preorden) i) ++ (((map f) . preorden) d))
        [por .]
   = f x : (((preorden . (maptree f)) i) ++ ((preorden . (maptree f)) d))
        [por hip. de inducción]
   = preorden (Nodo (f x) (maptree f i) (maptree f d))
        [por preorden.2]
   = preorden (maptree f (Nodo x i d))
        [por maptree.2]
   = (preorden . (maptree f)) (Nodo x i d)
        [por .]
-}
```

Relación 20

Cálculo numérico

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Diferenciación numérica                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    derivada :: Float -> (Float -> Float) -> Float -> Float
-- 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  ==>  -0.9999273
--    derivada 0.001 cos pi  ==>  0.0004768371
-- ---------------------------------------------------------------------

derivada :: Float -> (Float -> Float) -> Float -> Float
derivada a f x = (f(x+a)-f(x))/a

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir las funciones
--    derivadaBurda :: (Float -> Float) -> Float -> Float
--    derivadaFina  :: (Float -> Float) -> Float -> Float
--    derivadaSuper :: (Float -> Float) -> Float -> Float
-- 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,
--    derivadaFina cos pi  ==>  0.0
-- ---------------------------------------------------------------------

derivadaBurda :: (Float -> Float) -> Float -> Float
derivadaBurda = derivada 0.01

derivadaFina :: (Float -> Float) -> Float -> Float
derivadaFina  = derivada 0.0001

derivadaSuper :: (Float -> Float) -> Float -> Float
derivadaSuper = derivada 0.000001

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función
--    derivadaFinaDelSeno :: Float -> Float
-- tal que (derivadaFinaDelSeno x) es el valor de la derivada fina del
-- seno en x. Por ejemplo,
--    derivadaFinaDelSeno pi  ==>  -0.9989738
-- ---------------------------------------------------------------------

derivadaFinaDelSeno :: Float -> Float
derivadaFinaDelSeno = derivadaFina sin

-- ---------------------------------------------------------------------
-- Cálculo de la raíz cuadrada                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3. 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 cuadradad de x, entonces
--   (y+x/y) es una aproximación mejor.
```

```
--       x_0     = 1
--       x_{n+1} = (x_n+x/x_n)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por iteración con until, la función
--    raiz :: (Fractional a, Ord a) => a -> a
-- 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 :: (Fractional a, Ord a) => a -> a
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
--    (~=) :: Float -> Float -> Bool
-- tal que (x ~= y) si |x-y| < 0.001. Por ejemplo,
--    3.05 ~= 3.07        ==  False
--    3.00005 ~= 3.00007  == True
-- ---------------------------------------------------------------------

infix 5 ~=
(~=) :: Float -> Float -> 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 :: Float -> Property
```

```haskell
prop_raiz x =
    x >= 0 ==> (raiz x)^2 ~= 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 :: (Fractional a, Ord a) => a -> a
-- tal que (raizI x) es la raíz cuadrada de x calculada usando la
-- propiedad anterior. Por ejemplo,
--    raizI 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raizI :: (Fractional a, Ord a) => a -> a
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
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_raizI :: Float -> Property
prop_raizI x =
    x >= 0 ==> (raizI x)^2 ~= 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.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por recursión la función
--    puntoCero :: (Float -> Float) -> Float
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCero cos  ==>  1.570796
-- ---------------------------------------------------------------------

puntoCero :: (Float -> Float) -> Float
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
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por iteración con until, la función
--    puntoCeroI :: (Float -> Float) -> Float
-- tal que (puntoCeroI f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCeroI cos  ==>  1.570796
-- ---------------------------------------------------------------------

puntoCeroI :: (Float -> Float) -> Float
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 :: Float -> Float
-- tal que (raizCuadrada x) es la raíz cuadrada de x. Por ejemplo,
--    raizCuadrada 9  ==>  3.0
-- ---------------------------------------------------------------------

raizCuadrada :: Float -> Float
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
```

```haskell
prop_raizCuadrada :: Float -> Property
prop_raizCuadrada x =
    x >= 0 ==> (raizCuadrada x)^2 ~= x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir, usando puntoCero, la función
--    raizCubica :: Float -> Float
-- tal que (raizCubica x) es la raíz cuadrada de x. Por ejemplo,
--    raizCubica 27  ==>  3.0
-- ---------------------------------------------------------------------

raizCubica :: Float -> Float
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
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizCubica :: Float -> Property
prop_raizCubica x =
    x >= 0 ==> (raizCubica x)^3 ~= x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.5. Definir, usando puntoCero, la función
--    arcoseno :: Float -> Float
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcoseno 1  == 1.566332
-- ---------------------------------------------------------------------
```

```haskell
arcoseno :: Float -> Float
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 :: Float -> Property
prop_arcoseno x =
    0 <= x && x <= 1  ==> sin (arcoseno x) ~= x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--    arcocoseno :: Float -> Float
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcocoseno 0  == 1.5707963
-- ---------------------------------------------------------------------

arcocoseno :: Float -> Float
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 :: Float -> Property
prop_arcocoseno x =
```

```
    0 <= x && x <= 1  ==> cos (arcocoseno x) ~= x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--    inversa :: (Float -> Float) -> Float -> Float
-- tal que (inversa g x) es el valor de la inversa de g en x. Por
-- ejemplo,
--    inversa (^2) 9  ==  3.0
-- ---------------------------------------------------------------------

inversa :: (Float -> Float) -> Float -> Float
inversa g a = puntoCero f
    where f x = g x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.8. Redefinir, usando inversa, las funciones raizCuadrada,
-- raizCubica, arcoseno y arcocoseno.
-- ---------------------------------------------------------------------

raizCuadrada' = inversa (^2)
raizCubica'   = inversa (^3)
arcoseno'     = inversa sin
arcocoseno'   = inversa cos
```

Relación 21

Listas infinitas (3)

```
-- ---------------------------------------------------------------------
-- 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
--    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.
```

```haskell
--    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
--    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
```

```haskell
-- 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 2.1. Definir la función
--    iniciales :: [a] -> [[a]]
-- tal que (iniciales xs) es la lista de los segmentos iniciales de la
-- lista xs. Por ejemplo,
--    iniciales [2,3,4]    ==  [[],[2],[2,3],[2,3,4]]
--    iniciales [1,2,3,4]  ==  [[],[1],[1,2],[1,2,3],[1,2,3,4]]
-- ---------------------------------------------------------------------

iniciales :: [a] -> [[a]]
iniciales []     = [[]]
iniciales (x:xs) = [] : [x:ys | ys <- iniciales xs]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que el número de los
-- segmentos iniciales es el número de los elementos de la lista más
-- uno.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_iniciales :: [Int] -> Bool
prop_iniciales xs =
    length(iniciales xs) == 1 + length xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    finales :: [a] -> [[a]]
-- tal que (finales xs) es la lista de los segmentos finales de la lista
-- xs. Por ejemplo,
--    finales [2,3,4]    ==  [[2,3,4],[3,4],[4],[]]
--    finales [1,2,3,4]  ==  [[1,2,3,4],[2,3,4],[3,4],[4],[]]
-- ---------------------------------------------------------------------

finales :: [a] -> [[a]]
finales []     = [[]]
finales (x:xs) = (x:xs) : finales xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que el número de los
-- segmentos finales es el número de los elementos de la lista más uno.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_finales :: [Int] -> Bool
prop_finales xs =
    length(finales xs) == 1 + length xs

-- La comprobación es
```

```
--    Main> quickCheck prop_finales
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    segmentos :: [a] -> [[a]]
-- tal que (segmentos xs) es la lista de los segmentos de la lista
-- xs. Por ejemplo,
--    Main> segmentos [2,3,4]
--    [[],[4],[3],[3,4],[2],[2,3],[2,3,4]]
--    Main> segmentos [1,2,3,4]
--    [[],[4],[3],[3,4],[2],[2,3],[2,3,4],[1],[1,2],[1,2,3],[1,2,3,4]]
-- ---------------------------------------------------------------------

segmentos :: [a] -> [[a]]
segmentos []     = [[]]
segmentos (x:xs) = segmentos xs ++ [x:ys | ys <- iniciales xs]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que todos los segmentos
-- iniciales de xs son segmentos de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_iniciales_segmentos :: [Int] -> Bool
prop_iniciales_segmentos xs =
    and [elem ys segmentosDexs | ys <- inicialesDexs]
    where segmentosDexs = segmentos xs
          inicialesDexs = iniciales xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que si para todo lista xs se
-- verifica que (segmentos xs) tiene algún elemento que no pertenece a
-- (iniciales xs) ni a (finales xs).
--
-- En el caso de no verificarse, determinar una condición suficiente
```

```haskell
-- para que se verifique.
-- ---------------------------------------------------------------------

-- La propiedad general es
prop_segmentos :: [Int] -> Bool
prop_segmentos xs =
    [ys | ys <- (segmentos xs),
          notElem ys (iniciales xs),
          notElem ys (finales xs)]   /= []

-- La comprobación es
--    *Main> quickCheck prop_segmentos
--    Falsifiable, after 0 tests:
--    [1]

-- Por tanto la propiedad general no se verifica.

-- Una condición suficiente es que la lista xs tenga más de 2 elementos
-- y no tenga elementos repetidos.
prop_segmentos' :: [Int] -> Property
prop_segmentos' xs =
    length xs > 2 && sinRepetidos xs ==>
    [ys | ys <- (segmentos xs),
          notElem ys (iniciales xs),
          notElem ys (finales xs)]   /= []
    where sinRepetidos []     = True
          sinRepetidos (x:xs) = notElem x xs && sinRepetidos xs

-- La comprobación es
--    *Main> quickCheck prop_segmentos'
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir la función
--    subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de las subconjuntos de la lista
-- xs. Por ejemplo,
--    Main> subconjuntos [2,3,4]
--    [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
--    Main> subconjuntos [1,2,3,4]
```

```haskell
--    [[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
--       [2,3,4],  [2,3],  [2,4],  [2],  [3,4],  [3],  [4], []]
-- ---------------------------------------------------------------------

subconjuntos :: [a] -> [[a]]
subconjuntos []     = [[]]
subconjuntos (x:xs) = [x:ys | ys <- sub] ++ sub
                      where sub = subconjuntos xs

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir, por recursión, la función
--    subconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
--    subconjunto [1,3,2,3] [1,2,3]  ==  True
--    subconjunto [1,3,4,3] [1,2,3]  ==  False
-- ---------------------------------------------------------------------

subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto []     _ = True
subconjunto (x:xs) ys = elem x ys && subconjunto xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir, mediante all, la función
--    subconjunto' :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto' xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
--    subconjunto' [1,3,2,3] [1,2,3]  ==  True
--    subconjunto' [1,3,4,3] [1,2,3]  ==  False
-- ---------------------------------------------------------------------

subconjunto' :: Eq a => [a] -> [a] -> Bool
subconjunto' xs ys = all (`elem` ys) xs

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Comprobar con QuickCheck que las funciones subconjunto
-- y subconjunto' son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    subconjunto xs ys == subconjunto' xs ys

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    igualConjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (igualConjunto xs ys) se verifica si las listas xs e ys,
-- vistas como conjuntos, son iguales. Por ejemplo,
--    igualConjunto [1..10] [10,9..1]   ==  True
--    igualConjunto [1..10] [11,10..1]  ==  False
-- ---------------------------------------------------------------------

igualConjunto :: Eq a => [a] -> [a] -> Bool
igualConjunto xs ys = subconjunto xs ys && subconjunto ys xs

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    permutaciones :: Eq a => [a] -> [[a]]
-- tal que (permutaciones xs) es la lista de las permutaciones de la
-- lista xs. Por ejemplo,
--    Main> permutaciones [2,3]
--    [[2,3],[3,2]]
--    Main> permutaciones [1,2,3]
--    [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
-- ---------------------------------------------------------------------

permutaciones :: Eq a => [a] -> [[a]]
permutaciones [] = [[]]
permutaciones xs = [x:p | x <- xs, p <- permutaciones(xs \\ [x])]

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    combinaciones :: Int -> [a] -> [[a]]
-- tal que (combinaciones n xs) es la lista de las combinaciones n-arias
-- de la lista xs. Por ejemplo,
```

```haskell
--    combinaciones 2 [1,2,3,4]  ==  [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
-- ---------------------------------------------------------------------

-- 1ª definición
combinaciones_1 :: Int -> [a] -> [[a]]
combinaciones_1 n xs = [ys | ys <- subconjuntos xs, length ys == n]

-- 2ª definición
combinaciones_2 :: Int -> [a] -> [[a]]
combinaciones_2 0 _          = [[]]
combinaciones_2 _ []         = []
combinaciones_2 (n+1) (x:xs) = [x:ys | ys <- combinaciones_2 n xs] ++
                               combinaciones_2 (n+1) xs

-- Nota. La segunda definición es más eficiente como se comprueba en la
-- siguiente sesión
--    Main> :set +s
--    Main> length (combinaciones_1 2 [1..15])
--    105
--    (0.23 secs, 6394328 bytes)
--    Main> length (combinaciones_2 2 [1..15])
--    105
--    (0.01 secs, 525352 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 10. El triángulo de Pascal es un triángulo de números
--          1
--         1 1
--        1 2 1
--       1  3 3  1
--      1 4  6  4 1
--     1 5 10 10 5 1
--    ..............
-- construido de la siguiente forma
-- * la primera fila está formada por el número 1;
-- * las filas siguientes se construyen sumando los números adyacentes
--   de la fila superior y añadiendo un 1 al principio y al final de la
--   fila.
-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Definir la función
```

```haskell
--    pascal :: Integer -> [Integer]
-- tal que (pascal n) es lan-ésima fila del triángulo de Pascal. Por
-- ejemplo,
--    pascal 6  ==>  [1,5,10,10,5,1]
-- ---------------------------------------------------------------------

pascal :: Integer -> [Integer]
pascal 1 = [1]
pascal n = [1] ++ [x+y | (x,y) <- pares (pascal (n-1))] ++ [1]

-- donde pares xs es la lista formada por los pares de elementos
-- adyacentes de la lista xs. Por ejemplo,
--    pares [1,4,6,4,1]  ==>  [(1,4),(4,6),(6,4),(4,1)]
-- La definición de pares es
pares :: [a] -> [(a,a)]
pares (x:y:xs) = (x,y) : pares (y:xs)
pares _        = []

-- Otra definición de pares, usando zip, es
pares' :: [a] -> [(a,a)]
pares' xs = zip xs (tail xs)

-- Las definiciones son equivalentes
prop_ParesEquivPares' :: [Integer] -> Bool
prop_ParesEquivPares' xs =
    pares xs == pares' xs

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Comprobar con QuickCheck, que la fila n-ésima del
-- triángulo de Pascal tiene n elementos.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_Pascal :: Integer -> Property
prop_Pascal n =
    n >= 1 ==> fromIntegral (length (pascal n)) == n

-- Nótese el uso de la función fromIntegral para tansformar el valor de
-- length (pascal n) de Int a Integer. La comprobación es
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.3. Comprobar con QuickCheck, que la suma de los
-- elementos de la fila n-ésima del triángulo de Pascal es igual a
-- 2^(n-1).
-- ---------------------------------------------------------------------

-- la propiedad es
prop_sumaPascal :: Integer -> Property
prop_sumaPascal n =
    n >= 1 ==> sum (pascal n) == 2^(n-1)

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.4. Comprobar con QuickCheck, que el m-ésimo elemento de
-- la fila (n+1)-ésima del triángulo de Pascal es el número combinatorio
-- (n sobre m) = n! / (k!(n-k)!).
-- ---------------------------------------------------------------------

--  La propiedad es
prop_Combinaciones :: Integer -> Property
prop_Combinaciones n =
    n >= 1 ==> pascal n == [comb (n-1) m | m <- [0..n-1]]

-- donde (fact n) es el factorial de n y (comb n k) es el número
-- combinatorio n sobre k.
fact :: Integer -> Integer
fact n = product [1..n]

comb :: Integer -> Integer -> Integer
comb n k = (fact n) `div` ((fact k) * (fact (n-k)))

-- La comprobación es
--    Main> quickCheck prop_Combinaciones
```

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

Relación 22

Grafos

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

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Nota: Los grafos se pueden representar mediante la lista de sus
-- arcos. Por ejemplo, el grafo
--                           2
--                         / | \
--                        /  |  \
--                       1   |   3
--                           |  /
--                           | /
--                           |/
--                           4
-- puede representarse mediante la lista [(1,2),(2,3),(2,4),(3,4)]. En
-- los siguientes ejercicios se usará dicha representación.
-- ---------------------------------------------------------------------

type Grafo a = [(a,a)]

ejGrafo1 :: Grafo Int
ejGrafo1 = [(1,2),(2,3),(2,4),(3,4)]

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

```haskell
-- Ejercicio 1: Definir la función
--    adyacentes :: Grafo a -> a -> [a]
-- tal que (adyacentes g x) es la lista de los nodos adyacentes al nodo
-- x en el grafo g. Por ejemplo,
--    adyacentes [(1,2),(2,3),(2,4),(3,4)] 3  ==  [4,2]
-- ---------------------------------------------------------------------

adyacentes :: Eq a => Grafo a -> a -> [a]
adyacentes g x =
    [y | (z,y) <- g, z==x] ++ [y | (y,z) <- g, z==x]

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    nodos :: Grafo a -> [a]
-- tal que (nodos g) es el conjunto de los nodos del grafo g. Por
-- ejemplo,
--    nodos [(1,2),(2,3),(2,4),(3,4)]  ==  [2,3,4,1]
-- ---------------------------------------------------------------------

nodos :: Eq a => Grafo a -> [a]
nodos g =
    nub ([y | (z,y) <- g] ++ [y | (y,z) <- g])

-- ---------------------------------------------------------------------
-- Ejercicio 4: Definir la función
--    caminos :: Eq a => Grafo a -> a -> a -> [Camino a]
-- tal que (caminos g x y ) es la lista de los caminos en el grafo g
-- desde el nodo x al nodo y. Por ejemplo,
--    caminos [(1,2),(2,3),(2,4),(3,4)] 1 4  ==  [[1,2,4],[1,2,3,4]]
-- El tipo Camino es una lista de nodos.
-- ---------------------------------------------------------------------

type Camino a = [a]

caminos :: Eq a => Grafo a -> a -> a -> [Camino a]
caminos g x y =
    caminosAux g x [y]

-- (caminosAux g x vis) es la lista de los caminos desde el nodo x al
-- primer nodo del camino parcial vis (con nodos distintos a los de vis)
```

```haskell
-- junto vis.
caminosAux :: Eq a => Grafo a -> a -> [a] -> [Camino a]
caminosAux g x vis@(y:ys)
    | x==y      = [vis]
    | otherwise = concat [caminosAux g x (z:vis) |
                          z <- adyacentes g y,
                          notElem z vis]

-- ---------------------------------------------------------------------
-- Ejercicio 5: Definir la función
--    Eq a => Grafo a -> Bool
-- tal que (conectado g) se verifica si el grafo g está conectado; es
-- decir, existe un camino entre cada par de vértices. Por ejemplo,
--    conectado [(1,2),(2,3),(2,4),(3,4)]  ==  True
--    conectado [(1,2),(3,4)]              ==  False
-- ---------------------------------------------------------------------

conectado :: Eq a => Grafo a -> Bool
conectado g =
    null [(x,y) | x <- nodos g, y <- nodos g, null (caminos g x y)]

-- ---------------------------------------------------------------------
-- Ejercicio 6: Definir la función
--    tieneCiclos :: Eq a => Grafo a -> Bool
-- tal que (tieneCiclos g) se verifica si en el grafo g hay ciclos. Por
-- ejemplo,
--    tieneCiclos [(1,2),(2,3),(2,4),(3,4)]  ==  True
--    tieneCiclos [(1,2),(2,3),(2,4)]        ==  False
-- ---------------------------------------------------------------------

tieneCiclos :: Eq a => Grafo a -> Bool
tieneCiclos g =
    [(x,y) |
     x <- nodos g,
     y <- adyacentes g x,
     not (null [c | c <- caminos g x y, length c > 2])] /= []

-- ---------------------------------------------------------------------
-- Ejercicio 7: Definir la función
--    esArbol :: Eq a => Grafo a -> Bool
```

```haskell
-- tal que (esArbol g) se verifica si g es  un árbol; es decir, g es un
-- grafo conectado sin ciclos. Por ejemplo,
--    esArbol [(1,2),(2,3),(2,4),(3,4)]  ==  False
--    esArbol [(1,2),(2,3),(2,4)]        ==  True
-- ---------------------------------------------------------------------

esArbol :: Eq a => Grafo a -> Bool
esArbol g =
    conectado g && not(tieneCiclos g)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    caminosHamiltonianos :: Eq a => Grafo a -> [Camino a]
-- tal que (caminosHamiltonianos g) es la lista de los caminos
-- hamiltoniano en el grafo g (es decir, es la lista de los caminos en g
-- que pasa por todos sus nodos). Por ejemplo,
--    *Main> caminosHamiltonianos ejGrafo1
--    [[3,4,2,1],[4,3,2,1],[1,2,4,3],[1,2,3,4]]
-- ---------------------------------------------------------------------

caminosHamiltonianos :: Eq a => Grafo a -> [Camino a]
caminosHamiltonianos g =
    [xs | x <- ns,
          y <- ns \\ [x],
          xs <- caminos g x y,
          length xs == length ns]
    where ns = nodos g
```