# Soluciones de ejercicios de "Informática de 1º de Matemáticas"

(Curso 2019–20)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Sevilla, 9 de agosto de 2020

# 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 2019–20.

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 del curso [2].

[1] http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/2019-20-I1M-temas-PF.pdf
[2] http://www.glc.us.es/~jalonso/ejerciciosI1M2019G4

# Relación 1

# Definiciones por composición sobre números, listas y booleanos

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

-- En esta relación se plantean ejercicios con definiciones de funciones
-- por composición sobre números, listas y booleanos.
--
-- Para solucionar los ejercicios puede ser útil el manual de
-- funciones de Haskell que se encuentra en http://bit.ly/1uJZiqi y su
-- resumen en http://bit.ly/ZwSMHO

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función sumaMonedas tal que
-- (sumaMonedas a b c d e) es la suma de los euros correspondientes a
```

```
-- a monedas de 1 euro, b de 2 euros, c de 5 euros, d 10 euros y
-- e de 20 euros. Por ejemplo,
--    sumaMonedas 0 0 0 0 1  ==  20
--    sumaMonedas 0 0 8 0 3  == 100
--    sumaMonedas 1 1 1 1 1  ==  38
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función areaDeCoronaCircular tal que
-- (areaDeCoronaCircular r1 r2) es el área de una corona circular de
-- radio interior r1 y radio exterior r2. Por ejemplo,
--    areaDeCoronaCircular 1 2 == 9.42477796076938
--    areaDeCoronaCircular 2 5 == 65.97344572538566
--    areaDeCoronaCircular 3 5 == 50.26548245743669
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función ultimaCifra tal que (ultimaCifra x)
-- es la última cifra del número x. Por ejemplo,
--    ultimaCifra 325  ==  5
-- Indicación: Usar la función rem
-- ---------------------------------------------------------------------

ultimaCifra x = rem x 10

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función maxTres tal que (maxTres x y z) es
```

```
-- el máximo de x, y y z. Por ejemplo,
--    maxTres 6 2 4  ==  6
--    maxTres 6 7 4  ==  7
--    maxTres 6 7 9  ==  9
-- Indicación: Usar la función max.
-- ---------------------------------------------------------------------

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

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

rota1 xs = tail xs ++ [head xs]

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

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

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

rango xs = [minimum xs, maximum xs]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función palindromo tal que (palindromo xs) se
```

```haskell
-- verifica si xs es un palíndromo; es decir, es lo mismo leer xs de
-- izquierda a derecha que de derecha a izquierda. Por ejemplo,
--    palindromo [3,2,5,2,3]    ==  True
--    palindromo [3,2,5,6,2,3]  ==  False
-- ---------------------------------------------------------------------

palindromo xs = xs == reverse xs

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función interior tal que (interior xs) es la
-- lista obtenida eliminando los extremos de la lista xs. Por ejemplo,
--    interior [2,5,3,7,3]  ==  [5,3,7]
--    interior [2..7]       ==  [3,4,5,6]
-- ---------------------------------------------------------------------

interior xs = tail (init xs)

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función finales tal que (finales n xs) es la
-- lista formada por los n finales elementos de xs. Por ejemplo,
--    finales 3 [2,5,4,7,9,6]  ==  [7,9,6]
-- ---------------------------------------------------------------------

finales n xs = drop (length xs - n) xs

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función segmento tal que (segmento m n xs) es
-- la lista de los elementos de xs comprendidos entre las posiciones m y
-- n. Por ejemplo,
--    segmento 3 4 [3,4,1,2,7,9,0]  ==  [1,2]
--    segmento 3 5 [3,4,1,2,7,9,0]  ==  [1,2,7]
--    segmento 5 3 [3,4,1,2,7,9,0]  ==  []
-- ---------------------------------------------------------------------

segmento m n xs = drop (m-1) (take n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función extremos tal que (extremos n xs) es
-- la lista formada por los n primeros elementos de xs y los n finales
-- elementos de xs. Por ejemplo,
```

```
--    extremos 3 [2,6,7,1,2,4,5,8,9,2,3]  ==  [2,6,7,9,2,3]
-- ---------------------------------------------------------------------

extremos n xs = take n xs ++ drop (length xs - n) xs

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función mediano tal que (mediano x y z) es el
-- número mediano de los tres números x, y y z. Por ejemplo,
--    mediano 3 2 5  ==  3
--    mediano 2 4 5  ==  4
--    mediano 2 6 5  ==  5
--    mediano 2 6 6  ==  6
-- Indicación: Usar maximum y minimum.
-- ---------------------------------------------------------------------

mediano x y z = x + y + z- minimum [x,y,z] - maximum [x,y,z]

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función tresIguales tal que
-- (tresIguales x y z) se verifica si los elementos x, y y z son
-- iguales. Por ejemplo,
--    tresIguales 4 4 4  ==  True
--    tresIguales 4 3 4  ==  False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función tresDiferentes tal que
-- (tresDiferentes x y z) se verifica si los elementos x, y y z son
-- distintos. Por ejemplo,
--    tresDiferentes 3 5 2  ==  True
--    tresDiferentes 3 5 3  ==  False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función cuatroIguales tal que
-- (cuatroIguales x y z u) se verifica si los elementos x, y, z y u son
```

```
-- iguales. Por ejemplo,
--    cuatroIguales 5 5 5 5   ==  True
--    cuatroIguales 5 5 4 5   ==  False
-- Indicación: Usar la función tresIguales.
-- ---------------------------------------------------------------------

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

# Relación 2

# Definiciones con condicionales, guardas o patrones

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con definiciones elementales
-- (no recursivas) de funciones que usan condicionales, guardas o
-- patrones.
--
-- Estos ejercicios se corresponden con el tema 4 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-4.html

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    divisionSegura :: Double -> Double -> Double
-- tal que (divisionSegura x y) es x/y si y no es cero y 9999 en caso
-- contrario. Por ejemplo,
--    divisionSegura 7 2  ==  3.5
--    divisionSegura 7 0  ==  9999.0
-- ---------------------------------------------------------------------

divisionSegura :: Double -> Double -> Double
divisionSegura _ 0 = 9999
divisionSegura x y = x/y

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. La disyunción excluyente xor de dos fórmulas se
```

```
-- verifica si una es verdadera y la otra es falsa. Su tabla de verdad
-- es
--    x     | y     | xor x y
--    ------+-------+---------
--    True  | True  | False
--    True  | False | True
--    False | True  | True
--    False | False | False
--
-- Definir la función
--    xor1 :: Bool -> Bool -> Bool
-- tal que (xor1 x y) es la disyunción excluyente de x e y, calculada a
-- partir de la tabla de verdad. Usar 4 ecuaciones, una por cada línea
-- de la tabla.
-- ---------------------------------------------------------------------

xor1 :: Bool -> Bool -> Bool
xor1 True  True  = False
xor1 True  False = True
xor1 False True  = True
xor1 False False = False

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    xor2 :: Bool -> Bool -> Bool
-- tal que (xor2 x y) es la disyunción excluyente de x e y, calculada a
-- partir de la tabla de verdad y patrones. Usar 2 ecuaciones, una por
-- cada valor del primer argumento.
-- ---------------------------------------------------------------------

xor2 :: Bool -> Bool -> Bool
xor2 True  y = not y
xor2 False y = y

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función
--    xor3 :: Bool -> Bool -> Bool
-- tal que (xor3 x y) es la disyunción excluyente de x e y, calculada
-- a partir de la disyunción (||), conjunción (&&) y negación (not).
-- Usar 1 ecuación.
```

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

-- 1ª definición:
xor3 :: Bool -> Bool -> Bool
xor3 x y = (x || y) && not (x && y)

-- 2ª definición:
xor3b :: Bool -> Bool -> Bool
xor3b x y = (x && not y) || (y && not x)

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Definir la función
--    xor4 :: Bool -> Bool -> Bool
-- tal que (xor4 x y) es la disyunción excluyente de x e y, calculada
-- a partir de desigualdad (/=). Usar 1 ecuación.
-- ---------------------------------------------------------------------

xor4 :: Bool -> Bool -> Bool
xor4 x y = x /= y

-- ---------------------------------------------------------------------
-- Ejercicio 3. Las dimensiones de los rectángulos puede representarse
-- por pares; por ejemplo, (5,3) representa a un rectángulo de base 5 y
-- altura 3.
--
-- Definir la función
--    mayorRectangulo :: (Num a, Ord a) => (a,a) -> (a,a) -> (a,a)
-- tal que (mayorRectangulo r1 r2) es el rectángulo de mayor área entre
-- r1 y r2. Por ejemplo,
--    mayorRectangulo (4,6) (3,7)  ==  (4,6)
--    mayorRectangulo (4,6) (3,8)  ==  (4,6)
--    mayorRectangulo (4,6) (3,9)  ==  (3,9)
-- ---------------------------------------------------------------------

mayorRectangulo :: (Num a, Ord a) => (a,a) -> (a,a) -> (a,a)
mayorRectangulo (a,b) (c,d)
  | a*b >= c*d = (a,b)
  | otherwise  = (c,d)

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

```haskell
-- Ejercicio 4. Definir la función
--    intercambia :: (a,b) -> (b,a)
-- tal que (intercambia p)  es el punto obtenido intercambiando las
-- coordenadas del punto p. Por ejemplo,
--    intercambia (2,5)  ==  (5,2)
--    intercambia (5,2)  ==  (2,5)
-- ---------------------------------------------------------------------

intercambia :: (a,b) -> (b,a)
intercambia (x,y) = (y,x)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    distancia :: (Double,Double) -> (Double,Double) -> Double
-- tal que (distancia p1 p2) es la distancia entre los puntos p1 y
-- p2. Por ejemplo,
--    distancia (1,2) (4,6)  ==  5.0
-- ---------------------------------------------------------------------

distancia :: (Double,Double) -> (Double,Double) -> Double
distancia (x1,y1) (x2,y2) = sqrt((x1-x2)**2+(y1-y2)**2)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir una función
--    ciclo :: [a] -> [a]
-- tal que (ciclo xs) es la lista obtenida permutando cíclicamente los
-- elementos de la lista xs, pasando el último elemento al principio de
-- la lista. Por ejemplo,
--    ciclo [2,5,7,9]  == [9,2,5,7]
--    ciclo []         == []
--    ciclo [2]        == [2]
-- ---------------------------------------------------------------------

ciclo :: [a] -> [a]
ciclo [] = []
ciclo xs = last xs : init xs

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    numeroMayor :: (Num a, Ord a) => a -> a -> a
```

```haskell
-- tal que (numeroMayor x y) es el mayor número de dos cifras que puede
-- construirse con los dígitos x e y. Por ejemplo,
--    numeroMayor 2 5 ==  52
--    numeroMayor 5 2 ==  52
-- ---------------------------------------------------------------------

-- 1ª definición:
numeroMayor :: (Num a, Ord a) => a -> a -> a
numeroMayor x y = 10 * max x y + min x y

-- 2ª definición:
numeroMayor2 :: (Num a, Ord a) => a -> a -> a
numeroMayor2 x y
  | x > y     = 10*x+y
  | otherwise = 10*y+x

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    numeroDeRaices :: (Floating t, Ord t) => t -> t -> t -> Int
-- tal que (numeroDeRaices a b c) es el número de raíces reales de la
-- ecuación a*x^2 + b*x + c = 0. Por ejemplo,
--    numeroDeRaices 2 0 3    ==  0
--    numeroDeRaices 4 4 1    ==  1
--    numeroDeRaices 5 23 12  ==  2
-- Nota: Se supone que a es no nulo.
-- ---------------------------------------------------------------------

numeroDeRaices :: Double -> Double -> Double -> Int
numeroDeRaices a b c | d < 0     = 0
                     | d == 0    = 1
                     | otherwise = 2
    where d = b**2-4*a*c

-- 2ª solución
numeroDeRaices2 :: Double -> Double -> Double -> Int
numeroDeRaices2 a b c = 1 + round (signum (b**2-4*a*c))

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    raices :: Double -> Double -> Double -> [Double]
```

```haskell
-- tal que (raices a b c) es la lista de las raíces reales de la
-- ecuación ax^2 + bx + c = 0. Por ejemplo,
--    raices 1 3 2    ==  [-1.0,-2.0]
--    raices 1 (-2) 1 ==  [1.0,1.0]
--    raices 1 0 1    ==  []
-- Nota: Se supone que a es no nulo.
-- ---------------------------------------------------------------------

raices :: Double -> Double -> Double -> [Double]
raices a b c
  | d >= 0    = [(-b+e)/t,(-b-e)/t]
  | otherwise = []
  where d = b**2 - 4*a*c
        e = sqrt d
        t = 2*a

-- ---------------------------------------------------------------------
-- Ejercicio 10. En geometría, la fórmula de Herón, descubierta por
-- Herón de Alejandría, dice que el área de un triángulo cuyo lados
-- miden a, b y c es la raíz cuadrada de s(s-a)(s-b)(s-c) donde s es el
-- semiperímetro
--    s = (a+b+c)/2
--
-- Definir la función
--    area :: Double -> Double -> Double -> Double
-- tal que (area a b c) es el área del triángulo de lados a, b y c. Por
-- ejemplo,
--    area 3 4 5  ==  6.0
-- ---------------------------------------------------------------------

area :: Double -> Double -> Double -> Double
area a b c = sqrt (s*(s-a)*(s-b)*(s-c))
  where s = (a+b+c)/2

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Los intervalos cerrados se pueden representar mediante
-- una lista de dos números (el primero es el extremo inferior del
-- intervalo y el segundo el superior).
--
-- Definir la función
```

```haskell
--    interseccion :: Ord a => [a] -> [a] -> [a]
-- tal que (interseccion i1 i2) es la intersección de los intervalos i1 e
-- i2. Por ejemplo,
--    interseccion [] [3,5]     ==  []
--    interseccion [3,5] []     ==  []
--    interseccion [2,4] [6,9]  ==  []
--    interseccion [2,6] [6,9]  ==  [6,6]
--    interseccion [2,6] [0,9]  ==  [2,6]
--    interseccion [2,6] [0,4]  ==  [2,4]
--    interseccion [4,6] [0,4]  ==  [4,4]
--    interseccion [5,6] [0,4]  ==  []
-- ---------------------------------------------------------------------

interseccion :: Ord a => [a] -> [a] -> [a]
interseccion [] _ = []
interseccion _ [] = []
interseccion [a1,b1] [a2,b2]
  | a <= b    = [a,b]
  | otherwise = []
  where a = max a1 a2
        b = min b1 b2
interseccion _ _ = error "Imposible"

-- ---------------------------------------------------------------------
-- Ejercicio 12.1. Los números racionales pueden representarse mediante
-- pares de números enteros. Por ejemplo, el número 2/5 puede
-- representarse mediante el par (2,5).
--
-- Definir la función
--    formaReducida :: (Int,Int) -> (Int,Int)
-- tal que (formaReducida x) es la forma reducida del número racional
-- x. Por ejemplo,
--    formaReducida (4,10)  ==  (2,5)
--    formaReducida (0,5)   ==  (0,1)
-- ---------------------------------------------------------------------

formaReducida :: (Int,Int) -> (Int,Int)
formaReducida (0,_) = (0,1)
formaReducida (a,b) = (x * signum (a*b), y)
  where c = gcd a b
```

```
        x = abs (a `div` c)
        y = abs (b `div` c)

-- ---------------------------------------------------------------------
-- Ejercicio 12.2. Definir la función
--    sumaRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
-- tal que (sumaRacional x y) es la suma de los números racionales x e
-- y, expresada en forma reducida. Por ejemplo,
--    sumaRacional (2,3) (5,6)  ==  (3,2)
--    sumaRacional (3,5) (-3,5) ==  (0,1)
-- ---------------------------------------------------------------------

sumaRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
sumaRacional (a,b) (c,d) = formaReducida (a*d+b*c, b*d)

-- ---------------------------------------------------------------------
-- Ejercicio 12.3. Definir la función
--    productoRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
-- tal que (productoRacional x y) es el producto de los números
-- racionales x e y, expresada en forma reducida. Por ejemplo,
--    productoRacional (2,3) (5,6)  ==  (5,9)
-- ---------------------------------------------------------------------

productoRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
productoRacional (a,b) (c,d) = formaReducida (a*c, b*d)

-- ---------------------------------------------------------------------
-- Ejercicio 12.4. Definir la función
--    igualdadRacional :: (Int,Int) -> (Int,Int) -> Bool
-- tal que (igualdadRacional x y) se verifica si los números racionales
-- x e y son iguales. Por ejemplo,
--    igualdadRacional (6,9) (10,15)  ==  True
--    igualdadRacional (6,9) (11,15)  ==  False
--    igualdadRacional (0,2) (0,-5)   ==  True
-- ---------------------------------------------------------------------

igualdadRacional :: (Int,Int) -> (Int,Int) -> Bool
igualdadRacional (a,b) (c,d) =
  a*d == b*c
```

# Relación 3

# Definiciones por comprensión

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

-- En esta relación se presentan ejercicios con definiciones por
-- comprensión correspondientes al tema 5 que se encuentra
--    http://www.cs.us.es/~jalonso/cursos/i1m-18temas/tema-5.html

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. (Problema 6 del proyecto Euler) En los distintos
-- apartados de este ejercicio se definen funciones para resolver el
-- problema 6 del proyecto Euler https://www.projecteuler.net/problem=6
--
-- Definir la función
--    suma :: Integer -> Integer
-- tal (suma n) es la suma de los n primeros números. Por ejemplo,
--    suma 3  ==  6
--    length (show (suma (10^100)))  ==  200
-- ---------------------------------------------------------------------

-- 1ª definición
suma :: Integer -> Integer
```

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

-- 2ª definición
suma2 :: Integer -> Integer
suma2 n = (1+n)*n `div` 2

-- Equivalencia:
prop_suma :: Integer -> Property
prop_suma n =
  n >= 0 ==> suma n == suma2 n

-- Comprobación
--    λ> quickCheck prop_suma
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
--    λ> suma (10^7)
--    50000005000000
--    (3.39 secs, 1,612,715,016 bytes)
--    λ> suma2 (10^7)
--    50000005000000
--    (0.01 secs, 414,576 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2 Definir la función
--    sumaDeCuadrados :: Integer -> Integer
-- tal que (sumaDeCuadrados n) es la suma de los cuadrados de los
-- primeros n números; es decir, 1^2 + 2^2 + ... + n^2. Por ejemplo,
--    sumaDeCuadrados 3    ==  14
--    sumaDeCuadrados 100  ==  338350
--    length (show (sumaDeCuadrados (10^100)))  ==  300
-- ---------------------------------------------------------------------

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

-- 2ª solución
sumaDeCuadrados2 :: Integer -> Integer
sumaDeCuadrados2 n = n*(n+1)*(2*n+1) `div` 6
```

```haskell
-- Equivalencia:
prop_sumaDeCuadrados :: Integer -> Property
prop_sumaDeCuadrados n =
  n >= 0 ==> sumaDeCuadrados n == sumaDeCuadrados2 n

-- Comprobación:
--    λ> quickCheck prop_sumaDeCuadrados
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia:
--    λ> sumaDeCuadrados (10^7)
--    333333383333335000000
--    (14.74 secs, 8,025,345,712 bytes)
--    λ> sumaDeCuadrados2 (10^7)
--    333333383333335000000
--    (0.01 secs, 422,168 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    euler6 :: Integer -> Integer
-- tal que (euler6 n) es la diferencia entre el cuadrado de la suma
-- de los n primeros números y la suma de los cuadrados de los n
-- primeros números. Por ejemplo,
--    euler6 10       ==  2640
--    euler6 (10^10)  ==  250000000016666666664166666665000000000
-- ---------------------------------------------------------------------

-- 1ª solución
euler6 :: Integer -> Integer
euler6 n = (suma n)^2 - sumaDeCuadrados n

-- 2ª solución
euler6b :: Integer -> Integer
euler6b n = (suma2 n)^ 2 - sumaDeCuadrados2 n

-- Comparación de eficiencia
--    λ> euler6 (10^7)
--    250000016666664166666500000
--    (17.86 secs, 9,637,652,608 bytes)
```

```
--    λ> euler6b (10^7)
--    25000001666666416666665000000
--    (0.01 secs, 426,656 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir por comprensión la función
--    replica :: Int -> a -> [a]
-- tal que (replica n x) es la lista formada por n copias del elemento
-- x. Por ejemplo,
--    replica 4 7     ==  [7,7,7,7]
--    replica 3 True  ==  [True, True, True]
-- Nota: La función replica es equivalente a la predefinida replicate.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Los triángulos aritméticos se forman como sigue
--     1
--     2  3
--     4  5  6
--     7  8  9 10
--    11 12 13 14 15
--    16 17 18 19 20 21
--
-- Definir la función
--    linea :: Integer -> [Integer]
-- tal que (linea n) es la línea n-ésima de los triángulos
-- aritméticos. Por ejemplo,
--    linea 4  ==  [7,8,9,10]
--    linea 5  ==  [11,12,13,14,15]
--    head (linea (10^20))  ==  49999999999999999995000000000000000001
-- ---------------------------------------------------------------------

-- 1ª definición
linea1 :: Integer -> [Integer]
linea1 n = [suma (n-1)+1..suma n]

-- 2ª definición
```

```haskell
linea2 :: Integer -> [Integer]
linea2 n = [s+1..s+n]
  where s = suma (n-1)

-- 3ª definición
linea3 :: Integer -> [Integer]
linea3 n = [s+1..s+n]
  where s = suma2 (n-1)

-- Equivalencia:
prop_linea :: Integer -> Property
prop_linea n =
  n >= 0
  ==>
     linea1 n == linea2 n
  && linea1 n == linea2 n

-- Comprobación:
--    λ> quickCheck prop_linea
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
--    λ> head (linea1 (3*10^6))
--    4499998500001
--    (2.14 secs, 967,773,888 bytes)
--    λ> head (linea2 (3*10^6))
--    4499998500001
--    (0.79 secs, 484,092,656 bytes)
--    λ> head (linea3 (3*10^6))
--    4499998500001
--    (0.01 secs, 414,824 bytes)

-- En lo que sigue, usaremos la 3ª definición
linea :: Integer -> [Integer]
linea = linea3

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    triangulo :: Integer -> [[Integer]]
-- tal que (triangulo n) es el triángulo aritmético de altura n. Por
```

```haskell
-- ejemplo,
--    triangulo 3  ==  [[1],[2,3],[4,5,6]]
--    triangulo 4  ==  [[1],[2,3],[4,5,6],[7,8,9,10]]
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Un número natural n se denomina abundante si es menor
-- que la suma de sus divisores propios. Por ejemplo, 12 y 30 son
-- abundantes pero 5 y 28 no lo son.
--
-- Definir la función
--    numeroAbundante :: Int -> Bool
-- tal que (numeroAbundante n) se verifica si n es un número
-- abundante. Por ejemplo,
--    numeroAbundante 5  == False
--    numeroAbundante 12 == True
--    numeroAbundante 28 == False
--    numeroAbundante 30 == True
```

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

numeroAbundante :: Int -> Bool
numeroAbundante n = n < sum (divisores n)

divisores :: Int -> [Int]
divisores n = [m | m <- [1..n-1], n `mod` m == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    numerosAbundantesMenores :: Int -> [Int]
-- tal que (numerosAbundantesMenores n) es la lista de números
-- abundantes menores o iguales que n. Por ejemplo,
--    numerosAbundantesMenores 50  ==  [12,18,20,24,30,36,40,42,48]
--    numerosAbundantesMenores 48  ==  [12,18,20,24,30,36,40,42,48]
-- ---------------------------------------------------------------------

numerosAbundantesMenores :: Int -> [Int]
numerosAbundantesMenores n = [x | x <- [1..n], numeroAbundante x]

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir la función
--    todosPares :: Int -> Bool
-- tal que (todosPares n) se verifica si todos los números abundantes
-- menores o iguales que n son pares. Por ejemplo,
--    todosPares 10    ==  True
--    todosPares 100   ==  True
--    todosPares 1000  ==  False
-- ---------------------------------------------------------------------

todosPares :: Int -> Bool
todosPares n = and [even x | x <- numerosAbundantesMenores n]

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Definir la constante
--    primerAbundanteImpar :: Int
-- que calcule el primer número natural abundante impar. Determinar el
-- valor de dicho número.
-- ---------------------------------------------------------------------
```

```haskell
primerAbundanteImpar :: Int
primerAbundanteImpar = head [x | x <- [1,3..], numeroAbundante x]

-- Su cálculo es
--    ghci> primerAbundanteImpar
--    945

-- ---------------------------------------------------------------------
-- Ejercicio 6 (Problema 1 del proyecto Euler) Definir la función
--    euler1 :: Int -> Int
-- tal que (euler1 n) es la suma de todos los múltiplos de 3 ó 5 menores
-- que n. Por ejemplo,
--    euler1 10  ==  23
--
-- Calcular la suma de todos los múltiplos de 3 ó 5 menores que 1000.
-- ---------------------------------------------------------------------

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

-- Cálculo:
--    ghci> euler1 1000
--    233168

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    circulo :: Int -> Int
-- tal que (circulo n) es el la cantidad de pares de números naturales
-- (x,y) que se encuentran dentro del círculo de radio n. Por ejemplo,
--    circulo 3  ==  9
--    circulo 4  ==  15
--    circulo 5  ==  22
--    circulo 100  ==  7949
-- ---------------------------------------------------------------------

circulo :: Int -> Int
circulo n = length [(x,y) | x <- [0..n], y <- [0..n], x*x+y*y < n*n]

-- La eficiencia puede mejorarse con
```

```haskell
circulo2 :: Int -> Int
circulo2 n = length [(x,y) | x <- [0..n-1]
                           , y <- [0..raizCuadradaEntera (n*n - x*x)]
                           , x*x+y*y < n*n]

-- (raizCuadradaEntera n) es la parte entera de la raíz cuadrada de
-- n. Por ejemplo,
--    raizCuadradaEntera 17  ==  4
raizCuadradaEntera :: Int -> Int
raizCuadradaEntera n = truncate (sqrt (fromIntegral n))

-- Comparación de eficiencia
--    λ> circulo (10^4)
--    78549754
--    (73.44 secs, 44,350,688,480 bytes)
--    λ> circulo2 (10^4)
--    78549754
--    (59.71 secs, 36,457,043,240 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir la función
--    aproxE :: Double -> [Double]
-- tal que (aproXE n) es la lista cuyos elementos son los términos de la
-- sucesión (1+1/m)**m desde 1 hasta n. Por ejemplo,
--    aproxE 1 == [2.0]
--    aproxE 4 == [2.0,2.25,2.37037037037037,2.44140625]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. ¿Cuál es el límite de la sucesión (1+1/m)**m ?
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Definir la función
--    errorAproxE :: Double -> Double
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir la función
--    aproxLimSeno :: Double -> [Double]
-- tal que (aproxLimSeno n) es la lista cuyos elementos son los términos
-- de la sucesión
--    sen(1/m)
--    --------
--      1/m
-- desde 1 hasta n. Por ejemplo,
--    aproxLimSeno 1 == [0.8414709848078965]
--    aproxLimSeno 2 == [0.8414709848078965,0.958851077208406]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. ¿Cuál es el límite de la sucesión sen(1/m)/(1/m) ?
-- ---------------------------------------------------------------------

-- El límite es 1.

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Definir la función
--    errorLimSeno :: Double -> Double
-- tal que (errorLimSeno x) es el menor número de términos de la sucesión
-- sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
-- que x. Por ejemplo,
```

```
--    errorLimSeno 0.1     ==   2.0
--    errorLimSeno 0.01    ==   5.0
--    errorLimSeno 0.001   ==  13.0
--    errorLimSeno 0.0001  ==  41.0
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Definir la función
--    calculaPi :: Double -> Double
-- tal que (calculaPi n) es la aproximación del número pi calculada
-- mediante la expresión
--    4*(1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1))
-- Por ejemplo,
--    calculaPi 3    ==  2.8952380952380956
--    calculaPi 300  ==  3.1449149035588526
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Definir la función
--    errorPi :: Double -> Double
-- tal que (errorPi x) es el menor número de términos de la serie
--    4*(1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1))
-- necesarios para obtener pi con un error menor que x. Por ejemplo,
--    errorPi 0.1    ==    9.0
--    errorPi 0.01   ==   99.0
--    errorPi 0.001  ==  999.0
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Una terna (x,y,z) de enteros positivos es pitagórica
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. 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) = length [1 | n <- [x,y,z], even n]

-- ---------------------------------------------------------------------
-- Ejercicio 11.3. 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 11.4. Demostrar la conjetura para todas las ternas
-- pitagóricas.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 12.1. (Problema 9 del proyecto Euler). Una terna pitagórica
-- es una terna de números naturales (a,b,c) tal que a<b<c y
-- a^2+b^2=c^2. Por ejemplo (3,4,5) es una terna pitagórica.
--
-- Definir la función
--    ternasPitagoricas :: Integer -> [[Integer]]
-- tal que (ternasPitagoricas x) es la lista de las ternas pitagóricas
-- cuya suma es x. Por ejemplo,
--    ternasPitagoricas 12  ==  [(3,4,5)]
--    ternasPitagoricas 60  ==  [(10,24,26),(15,20,25)]
-- ---------------------------------------------------------------------

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 12.2. Definir la constante
--    euler9 :: Integer
-- tal que euler9 es producto abc donde (a,b,c) es la única terna
-- pitagórica tal que a+b+c=1000.
--
-- Calcular el valor de euler9.
-- ---------------------------------------------------------------------

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

-- El cálculo del valor de euler9 es
--    ghci> euler9
--    31875000

-- ---------------------------------------------------------------------
-- Ejercicio 13. 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
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir, por comprensión, la función
--    sumaConsecutivos :: [Int] -> [Int]
-- tal que (sumaConsecutivos xs) es la suma de los pares de elementos
-- consecutivos de la lista xs. Por ejemplo,
--    sumaConsecutivos [3,1,5,2]  ==  [4,6,7]
--    sumaConsecutivos [3]        ==  []
-- ---------------------------------------------------------------------
```

```haskell
sumaConsecutivos :: [Int] -> [Int]
sumaConsecutivos xs = [x+y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 15. Los polinomios pueden representarse de forma dispersa o
-- densa. Por ejemplo, el polinomio 6x^4-5x^2+4x-7 se puede representar
-- de forma dispersa por [6,0,-5,4,-7] y de forma densa por
-- [(4,6),(2,-5),(1,4),(0,-7)].
--
-- Definir la función
--    densa :: [Int] -> [(Int,Int)]
-- tal que (densa xs) es la representación densa del polinomio cuya
-- representación dispersa es xs. Por ejemplo,
--   densa [6,0,-5,4,-7]  ==  [(4,6),(2,-5),(1,4),(0,-7)]
--   densa [6,0,0,3,0,4]  ==  [(5,6),(2,3),(0,4)]
-- ---------------------------------------------------------------------

densa :: [Int] -> [(Int,Int)]
densa xs = [(x,y) | (x,y) <- zip [n-1,n-2..0] xs, y /= 0]
  where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 16. La bases de datos sobre actividades de personas pueden
-- representarse mediante listas de elementos de la forma (a,b,c,d),
-- donde a es el nombre de la persona, b su actividad, c su fecha de
-- nacimiento y d la de su fallecimiento. Un ejemplo es la siguiente que
-- usaremos a lo largo de este ejercicio,
-- ---------------------------------------------------------------------

personas :: [(String,String,Int,Int)]
personas = [("Cervantes","Literatura",1547,1616),
            ("Velazquez","Pintura",1599,1660),
            ("Picasso","Pintura",1881,1973),
            ("Beethoven","Musica",1770,1823),
            ("Poincare","Ciencia",1854,1912),
            ("Quevedo","Literatura",1580,1654),
            ("Goya","Pintura",1746,1828),
            ("Einstein","Ciencia",1879,1955),
            ("Mozart","Musica",1756,1791),
```

```
            ("Botticelli","Pintura",1445,1510),
            ("Borromini","Arquitectura",1599,1667),
            ("Bach","Musica",1685,1750)]

-- ---------------------------------------------------------------------
-- Ejercicio 16.1. Definir la función
--    nombres :: [(String,String,Int,Int)] -> [String]
-- tal que (nombres bd) es la lista de los nombres de las personas de la
-- base de datos bd. Por ejemplo,
--    ghci> nombres personas
--     ["Cervantes","Velazquez","Picasso","Beethoven","Poincare",
--      "Quevedo","Goya","Einstein","Mozart","Botticelli","Borromini","Bach"]
-- ---------------------------------------------------------------------

nombres :: [(String,String,Int,Int)] -> [String]
nombres bd = [x | (x,_,_,_) <- bd]

-- ---------------------------------------------------------------------
-- Ejercicio 16.2. Definir la función
--    musicos :: [(String,String,Int,Int)] -> [String]
-- tal que (musicos bd) es la lista de los nombres de los músicos de la
-- base de datos bd. Por ejemplo,
--    musicos personas  ==  ["Beethoven","Mozart","Bach"]
-- ---------------------------------------------------------------------

musicos :: [(String,String,Int,Int)] -> [String]
musicos bd = [x | (x,"Musica",_,_) <- bd]

-- ---------------------------------------------------------------------
-- Ejercicio 16.3. Definir la función
--    seleccion :: [(String,String,Int,Int)] -> String -> [String]
-- tal que (seleccion bd m) es la lista de los nombres de las personas
-- de la base de datos bd cuya actividad es m. Por ejemplo,
--    ghci> seleccion personas "Pintura"
--    ["Velazquez","Picasso","Goya","Botticelli"]
--    ghci> seleccion personas "Musica"
--    ["Beethoven","Mozart","Bach"]
-- ---------------------------------------------------------------------

seleccion :: [(String,String,Int,Int)] -> String -> [String]
```

```haskell
seleccion bd m = [ x | (x,m',_,_) <- bd, m == m' ]

-- ---------------------------------------------------------------------
-- Ejercicio 16.4. Definir, usando el apartado anterior, la función
--    musicos' :: [(String,String,Int,Int)] -> [String]
-- tal que (musicos' bd) es la lista de los nombres de los músicos de la
-- base de datos bd. Por ejemplo,
--    ghci> musicos' personas
--    ["Beethoven","Mozart","Bach"]
-- ---------------------------------------------------------------------

musicos' :: [(String,String,Int,Int)] -> [String]
musicos' bd = seleccion bd "Musica"

-- ---------------------------------------------------------------------
-- Ejercicio 16.5. Definir la función
--    vivas :: [(String,String,Int,Int)] -> Int -> [String]
-- tal que (vivas bd a) es la lista de los nombres de las personas de la
-- base de datos bd  que estaban vivas en el año a. Por ejemplo,
--    ghci> vivas personas 1600
--    ["Cervantes","Velazquez","Quevedo","Borromini"]
-- ---------------------------------------------------------------------

vivas :: [(String,String,Int,Int)] -> Int -> [String]
vivas ps a = [x | (x,_,a1,a2) <- ps, a1 <= a, a <= a2]
```

# Relación 4

# Definiciones por recursión

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

-- En esta relación se presentan ejercicios con definiciones por
-- recursión correspondientes al tema 6 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-6.html

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

import Test.QuickCheck

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que la función potencia es
```

```haskell
-- equivalente a la predefinida (^).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_potencia :: Integer -> Integer -> Property
prop_potencia x n =
  n >= 0 ==> potencia x n == x^n

-- La comprobación es
--    ghci> quickCheck prop_potencia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Dados dos números naturales, a y b, es posible
-- calcular su máximo común divisor mediante el Algoritmo de
-- Euclides. Este algoritmo se puede resumir en la siguiente fórmula:
--    mcd(a,b) = a,                    si b = 0
--             = mcd (b, a módulo b), si b > 0
--
-- Definir la función
--    mcd :: Integer -> Integer -> Integer
-- tal que (mcd a b) es el máximo común divisor de a y b calculado
-- mediante el algoritmo de Euclides. Por ejemplo,
--    mcd 30 45  ==  15
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir y comprobar la propiedad prop_mcd según la
-- cual el máximo común divisor de dos números a y b (ambos mayores que
-- 0) es siempre mayor o igual que 1 y además es menor o igual que el
-- menor de los números a  y b.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd :: Integer -> Integer -> Property
prop_mcd a b =
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. 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 :: Integer -> Integer -> Property
prop_mcd_div a b =
  a > 0 && b > 0 ==> mcd a b <= max a b `div` 2

-- Al verificarla, se obtiene
--    ghci> 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' :: Integer -> Integer -> Property
prop_mcd_div' a b =
  a > 0 && b > 0 && a /= b ==> mcd a b <= max a b `div` 2

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1, Definir por recursión la función
--    pertenece :: Eq a => a -> [a] -> Bool
-- tal que (pertenece x xs) se verifica si x pertenece a la lista xs. Por
-- ejemplo,
```

```haskell
--    pertenece 3 [2,3,5]  ==  True
--    pertenece 4 [2,3,5]  ==  False
-- ---------------------------------------------------------------------

pertenece :: Eq a => a -> [a] -> Bool
pertenece _ []     = False
pertenece x (y:ys) = x == y || pertenece x ys

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con quickCheck que pertenece es equivalente
-- a elem.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_pertenece :: Eq a => a -> [a] -> Bool
prop_pertenece x xs = pertenece x xs == elem x xs

-- La comprobación es
--    ghci> quickCheck prop_pertenece
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por recursión la función
--    concatenaListas :: [[a]] -> [a]
-- tal que (concatenaListas xss) es la lista obtenida concatenando las
-- listas de xss. Por ejemplo,
--    concatenaListas [[1..3],[5..7],[8..10]]  ==  [1,2,3,5,6,7,8,9,10]
-- ---------------------------------------------------------------------

concatenaListas :: [[a]] -> [a]
concatenaListas []       = []
concatenaListas (xs:xss) = xs ++ concatenaListas xss

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que concatenaListas es
-- equivalente a concat.
-- ---------------------------------------------------------------------

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

```haskell
prop_concat xss = concatenaListas xss == concat xss

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir por recursión la función
--    coge :: Int -> [a] -> [a]
-- tal que (coge n xs) es la lista de los n primeros elementos de
-- xs. Por ejemplo,
--    coge 3 [4..12]  =>  [4,5,6]
-- ---------------------------------------------------------------------

coge :: Int -> [a] -> [a]
coge n _  | n <= 0 = []
coge _ []          = []
coge n (x:xs)      = x : coge (n-1) xs

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que coge es equivalente a
-- take.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_coge :: Int -> [Int] -> Bool
prop_coge n xs =
  coge n xs == take n xs

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir, por recursión, la función
--    sumaCuadradosR :: Integer -> Integer
-- tal que (sumaCuadradosR n) es la suma de los cuadrados de los números
-- de 1 a n. Por ejemplo,
--    sumaCuadradosR 4  ==  30
-- ---------------------------------------------------------------------

sumaCuadradosR :: Integer -> Integer
sumaCuadradosR 0 = 0
sumaCuadradosR n = n^2 + sumaCuadradosR (n-1)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Comprobar con QuickCheck si sumaCuadradosR n es igual a
-- n(n+1)(2n+1)/6.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_SumaCuadrados :: Integer -> Property
prop_SumaCuadrados n =
  n >= 0 ==>
    sumaCuadradosR n == n * (n+1) * (2*n+1) `div` 6

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir, por comprensión, la función
--    sumaCuadradosC :: Integer --> Integer
-- tal que (sumaCuadradosC n) es la suma de los cuadrados de los números
-- de 1 a n. Por ejemplo,
--    sumaCuadradosC 4  ==  30
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.4. Comprobar con QuickCheck que las funciones
-- sumaCuadradosR y sumaCuadradosC son equivalentes sobre los números
-- naturales.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaCuadradosR :: Integer -> Property
prop_sumaCuadradosR n =
    n >= 0 ==> sumaCuadradosR n == sumaCuadradosC n

-- La comprobación es
--    ghci> quickCheck prop_sumaCuadrados
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir, por recursión, la función
--    digitosR :: Integer -> [Integer]
-- tal que (digitosR n) es la lista de los dígitos del número n. Por
-- ejemplo,
--    digitosR 320274  ==  [3,2,0,2,7,4]
-- ---------------------------------------------------------------------

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

digitosR' :: Integer -> [Integer]
digitosR' n
  | n < 10    = [n]
  | otherwise = (n `rem` 10) : digitosR' (n `div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir, por comprensión, la función
--    digitosC :: Integer -> [Integer]
-- tal que (digitosC n) es la lista de los dígitos del número n. Por
-- ejemplo,
--    digitosC 320274  ==  [3,2,0,2,7,4]
-- Indicación: Usar las funciones show y read.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Comprobar con QuickCheck que las funciones digitosR y
-- digitosC son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_digitos :: Integer -> Property
prop_digitos n =
    n >= 0 ==>
    digitosR n == digitosC n
```

```haskell
-- La comprobación es
--    ghci> quickCheck prop_digitos
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir, por recursión, la función
--    sumaDigitosR :: Integer -> Integer
-- tal que (sumaDigitosR n) es la suma de los dígitos de n. Por ejemplo,
--    sumaDigitosR 3     ==  3
--    sumaDigitosR 2454  == 15
--    sumaDigitosR 20045 == 11
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Definir, sin usar recursión, la función
--    sumaDigitosNR :: Integer -> Integer
-- tal que (sumaDigitosNR n) es la suma de los dígitos de n. Por ejemplo,
--    sumaDigitosNR 3     ==  3
--    sumaDigitosNR 2454  == 15
--    sumaDigitosNR 20045 == 11
-- ---------------------------------------------------------------------

sumaDigitosNR :: Integer -> Integer
sumaDigitosNR n = sum (digitosC n)

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Comprobar con QuickCheck que las funciones sumaDigitosR
-- y sumaDigitosNR son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaDigitos :: Integer -> Property
prop_sumaDigitos n =
    n >= 0 ==>
```

```
    sumaDigitosR n == sumaDigitosNR n

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

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir, por recursión, la función
--    listaNumeroR :: [Integer] -> Integer
-- tal que (listaNumeroR xs) es el número formado por los dígitos xs. Por
-- ejemplo,
--    listaNumeroR [5]        == 5
--    listaNumeroR [1,3,4,7]  == 1347
--    listaNumeroR [0,0,1]    == 1
-- ---------------------------------------------------------------------

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

listaNumeroR' :: [Integer] -> Integer
listaNumeroR' []     = 0
listaNumeroR' (x:xs) = x + 10 * listaNumeroR' xs

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Definir, por comprensión, la función
--    listaNumeroC :: [Integer] -> Integer
-- tal que (listaNumeroC xs) es el número formado por los dígitos xs. Por
-- ejemplo,
--    listaNumeroC [5]        == 5
--    listaNumeroC [1,3,4,7]  == 1347
--    listaNumeroC [0,0,1]    == 1
-- ---------------------------------------------------------------------

-- 1ª definición:
listaNumeroC :: [Integer] -> Integer
listaNumeroC xs = sum [y*10^n | (y,n) <- zip (reverse xs) [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Comprobar con QuickCheck que las funciones
-- listaNumeroR y listaNumeroC son equivalentes.
```

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

-- La propiedad es
prop_listaNumero :: [Integer] -> Bool
prop_listaNumero xs =
  listaNumeroR xs == listaNumeroC xs

-- La comprobación es
--    ghci> quickCheck prop_listaNumero
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Definir, por recursión, la función
--    mayorExponenteR :: Integer -> Integer -> Integer
-- tal que (mayorExponenteR a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
--    mayorExponenteR 2 8    ==  3
--    mayorExponenteR 2 9    ==  0
--    mayorExponenteR 5 100  ==  2
--    mayorExponenteR 2 60   ==  2
--
-- Nota: Se supone que a es mayor que 1.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Definir, por comprensión, la función
--    mayorExponenteC :: Integer -> Integer -> Integer
-- tal que (mayorExponenteC a b) es el exponente de la mayor potencia de
-- a que divide a b. Por ejemplo,
--    mayorExponenteC 2 8    ==  3
--    mayorExponenteC 5 100  ==  2
--    mayorExponenteC 5 101  ==  0
--
-- Nota: Se supone que a es mayor que 1.
-- ---------------------------------------------------------------------
```

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

# Relación 5

# Operaciones conjuntistas con listas

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En estas relación se definen operaciones conjuntistas sobre listas.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    subconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys; es decir, si todos los elementos de xs pertenecen a ys. Por
-- ejemplo,
--    subconjunto [3,2,3] [2,5,3,5]  ==  True
--    subconjunto [3,2,3] [2,5,6,5]  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys =
  [x | x <- xs, x `elem` ys] == xs
```

```haskell
-- 2ª definición (por recursión)
subconjuntoR :: Eq a => [a] -> [a] -> Bool
subconjuntoR [] _      = True
subconjuntoR (x:xs) ys = x `elem` ys && subconjuntoR xs ys

-- La propiedad de equivalencia es
prop_subconjuntoR :: [Int] -> [Int] -> Bool
prop_subconjuntoR xs ys =
  subconjuntoR xs ys == subconjunto xs ys

-- La comprobación es
--    ghci> quickCheck prop_subconjuntoR
--    +++ OK, passed 100 tests.

-- 3ª definición (con all)
subconjuntoA :: Eq a => [a] -> [a] -> Bool
subconjuntoA xs ys = all (`elem` ys) xs

-- La propiedad de equivalencia es
prop_subconjuntoA :: [Int] -> [Int] -> Bool
prop_subconjuntoA xs ys =
  subconjunto xs ys == subconjuntoA xs ys

-- La comprobación es
--    ghci> quickCheck prop_subconjuntoA
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    iguales :: Eq a => [a] -> [a] -> Bool
-- tal que (iguales xs ys) se verifica si xs e ys son iguales; es decir,
-- tienen los mismos elementos. Por ejemplo,
--    iguales [3,2,3] [2,3]    ==  True
--    iguales [3,2,3] [2,3,2]  ==  True
--    iguales [3,2,3] [2,3,4]  ==  False
--    iguales [2,3] [4,5]      ==  False
-- ---------------------------------------------------------------------

iguales :: Eq a => [a] -> [a] -> Bool
```

```haskell
iguales xs ys =
  subconjunto xs ys && subconjunto ys xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    union :: Eq a => [a] -> [a] -> [a]
-- tal que (union xs ys) es la unión de los conjuntos xs e ys. Por
-- ejemplo,
--    union [3,2,5] [5,7,3,4]  ==  [3,2,5,7,4]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
union :: Eq a => [a] -> [a] -> [a]
union xs ys = xs ++ [y | y <- ys, y `notElem` xs]

-- 2ª definición (por recursión)
unionR :: Eq a => [a] -> [a] -> [a]
unionR []     ys = ys
unionR (x:xs) ys | x `elem` ys = xs `union` ys
                 | otherwise   = x : xs `union` ys

-- La propiedad de equivalencia es
prop_union :: [Int] -> [Int] -> Bool
prop_union xs ys =
  union xs ys `iguales` unionR xs ys

-- La comprobación es
--    ghci> quickCheck prop_union
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Nota. En los ejercicios de comprobación de propiedades, cuando se
-- trata con igualdades se usa la igualdad conjuntista (definida por la
-- función iguales) en lugar de la igualdad de lista (definida por ==)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que la unión es conmutativa.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_union_conmutativa :: [Int] -> [Int] -> Bool
prop_union_conmutativa xs ys =
  union xs ys `iguales` union ys xs

-- La comprobación es
--    ghci> quickCheck prop_union_conmutativa
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    interseccion :: Eq a => [a] -> [a] -> [a]
-- tal que (interseccion xs ys) es la intersección de xs e ys. Por
-- ejemplo,
--    interseccion [3,2,5] [5,7,3,4]  ==  [3,5]
--    interseccion [3,2,5] [9,7,6,4]  ==  []
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
interseccion :: Eq a => [a] -> [a] -> [a]
interseccion xs ys =
  [x | x <- xs, x `elem` ys]

-- 2ª definición (por recursión):
interseccionR :: Eq a => [a] -> [a] -> [a]
interseccionR []     _  = []
interseccionR (x:xs) ys
  | x `elem` ys = x : interseccionR xs ys
  | otherwise   = interseccionR xs ys

-- La propiedad de equivalencia es
prop_interseccion :: [Int] -> [Int] -> Bool
prop_interseccion xs ys =
  interseccion xs ys `iguales` interseccionR xs ys

-- La comprobación es
--    ghci> quickCheck prop_interseccion
--    +++ OK, passed 100 tests.

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

```haskell
-- Ejercicio 4.2. Comprobar con QuickCheck si se cumple la siguiente
-- propiedad
--    A ∪ (B ∩ C) = (A ∪ B) ∩ C
-- donde se considera la igualdad como conjuntos. En el caso de que no
-- se cumpla verificar el contraejemplo calculado por QuickCheck.
-- ---------------------------------------------------------------------

prop_union_interseccion :: [Int] -> [Int] -> [Int] -> Bool
prop_union_interseccion xs ys zs =
  iguales (union xs (interseccion ys zs))
          (interseccion (union xs ys) zs)

-- La comprobación es
--    ghci> quickCheck prop_union_interseccion
--    *** Failed! Falsifiable (after 3 tests and 2 shrinks):
--    [0]
--    []
--    []
--
-- Por tanto, la propiedad no se cumple y un contraejemplo es
--    A = [0], B = [] y C = []
-- ya que entonces,
--    A ∪ (B ∩ C) = [0] ∪ ([] ∩ []) = [0] ∪ [] = [0]
--    (A ∪ B) ∩ C = ([0] ∪ []) ∩ [] = [0] ∩ [] = []

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir la función
--    producto :: [a] -> [a] -> [(a,a)]
-- tal que (producto xs ys) es el producto cartesiano de xs e ys. Por
-- ejemplo,
--    producto [1,3] [2,4] == [(1,2),(1,4),(3,2),(3,4)]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
producto :: [a] -> [a] -> [(a,a)]
producto xs ys = [(x,y) | x <- xs, y <- ys]

-- 2ª definición (por recursión):
productoR :: [a] -> [a] -> [(a,a)]
productoR []     _  = []
```

```haskell
productoR (x:xs) ys = [(x,y) | y <- ys] ++ productoR xs ys

-- La propiedad de equivalencia es
prop_producto :: [Int] -> [Int] -> Bool
prop_producto xs ys =
  producto xs ys `iguales` productoR xs ys

-- La comprobación es
--    ghci> quickCheck prop_producto
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que el número de elementos
-- de (producto xs ys) es el producto del número de elementos de xs y de
-- ys.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_elementos_producto :: [Int] -> [Int] -> Bool
prop_elementos_producto xs ys =
  length (producto xs ys) == length xs * length ys

-- La comprobación es
--    ghci> quickCheck prop_elementos_producto
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir la función
--    subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de las subconjuntos de la lista
-- xs. Por ejemplo,
--    ghci> subconjuntos [2,3,4]
--    [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
--    ghci> subconjuntos [1,2,3,4]
--    [[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 []     = [[]]
```

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

-- Cambiando la comprensión por map se obtiene
subconjuntos' :: [a] -> [[a]]
subconjuntos' []     = [[]]
subconjuntos' (x:xs) = sub ++ map (x:) sub
  where sub = subconjuntos' xs

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Comprobar con QuickChek que el número de elementos de
-- (subconjuntos xs) es 2 elevado al número de elementos de xs.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=7}) prop_subconjuntos
-- ---------------------------------------------------------------------

-- La propiedad es
prop_subconjuntos :: [Int] -> Bool
prop_subconjuntos xs =
  length (subconjuntos xs) == 2 ^ length xs

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_subconjuntos
--    +++ OK, passed 100 tests.
```

# Relación 6

# El algoritmo de Luhn

```
-- ---------------------------------------------------------------------
-- § Introducción                                                      --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es estudiar un algoritmo para validar
-- algunos identificadores numéricos como los números de algunas tarjetas
-- de crédito; por ejemplo, las de tipo Visa o Master Card.
--
-- El algoritmo que vamos a estudiar es el algoritmo de Luhn consistente
-- en aplicar los siguientes pasos a los dígitos del número de la
-- tarjeta.
--    1. Se invierten los dígitos del número; por ejemplo, [9,4,5,5] se
--       transforma en [5,5,4,9].
--    2. Se duplican los dígitos que se encuentra en posiciones impares
--       (empezando a contar en 0); por ejemplo, [5,5,4,9] se transforma
--       en [5,10,4,18].
--    3. Se suman los dígitos de cada número; por ejemplo, [5,10,4,18]
--       se transforma en 5 + (1 + 0) + 4 + (1 + 8) = 19.
--    4. Si el último dígito de la suma es 0, el número es válido; y no
--       lo es, en caso contrario.
--
-- A los números válidos, los llamaremos números de Luhn.

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    digitosInv :: Integer -> [Integer]
-- tal que (digitosInv n) es la lista de los dígitos del número n. en
-- orden inverso. Por ejemplo,
```

```haskell
--    digitosR 320274  ==  [4,7,2,0,2,3]
-- ---------------------------------------------------------------------

-- 1ª solución
digitosInv :: Integer -> [Integer]
digitosInv n
  | n < 10    = [n]
  | otherwise = (n `rem` 10) : digitosInv (n `div` 10)

-- 2ª solución
digitosInv2 :: Integer -> [Integer]
digitosInv2 n = [read [x] | x <- reverse (show n)]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    doblePosImpar :: [Integer] -> [Integer]
-- tal que (doblePosImpar ns) es la lista obtenida doblando los
-- elementos en las posiciones impares (empezando a contar en cero y
-- dejando igual a los que están en posiciones pares. Por ejemplo,
--    doblePosImpar [4,9,5,5]    ==  [4,18,5,10]
--    doblePosImpar [4,9,5,5,7]  ==  [4,18,5,10,7]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión)
doblePosImpar :: [Integer] -> [Integer]
doblePosImpar []       = []
doblePosImpar [x]      = [x]
doblePosImpar (x:y:zs) = x : 2*y : doblePosImpar zs

-- 2ª definición (por recursión)
doblePosImpar2 :: [Integer] -> [Integer]
doblePosImpar2 (x:y:zs) = x : 2*y : doblePosImpar2 zs
doblePosImpar2 xs       = xs

-- 3ª definición (por comprensión)
doblePosImpar3 :: [Integer] -> [Integer]
doblePosImpar3 xs = [f n x | (n,x) <- zip [0..] xs]
  where f n x | odd n     = 2*x
              | otherwise = x
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    sumaDigitos :: [Integer] -> Integer
-- tal que (sumaDigitos ns) es la suma de los dígitos de ns. Por
-- ejemplo,
--    sumaDigitos [10,5,18,4] = 1 + 0 + 5 + 1 + 8 + 4 =
--                            = 19
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
sumaDigitos :: [Integer] -> Integer
sumaDigitos ns = sum [sum (digitosInv n) | n <- ns]

-- 2ª definición (por recursión):
sumaDigitos2 :: [Integer] -> Integer
sumaDigitos2 []     = 0
sumaDigitos2 (n:ns) = sum (digitosInv n) + sumaDigitos2 ns

-- 3ª definición (con orden superior):
sumaDigitos3 :: [Integer] -> Integer
sumaDigitos3 = sum . map (sum . digitosInv)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    ultimoDigito :: Integer -> Integer
-- tal que (ultimoDigito n) es el último dígito de n. Por ejemplo,
--    ultimoDigito 123 == 3
--    ultimoDigito   0 == 0
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    luhn :: Integer -> Bool
-- tal que (luhn n) se verifica si n es un número de Luhn. Por ejemplo,
--    luhn 5594589764218858  ==  True
--    luhn 1234567898765432  ==  False
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución
luhn :: Integer -> Bool
luhn n =
  ultimoDigito (sumaDigitos (doblePosImpar (digitosInv n))) == 0

-- 2ª solución
luhn2 =
  (==0) . ultimoDigito . sumaDigitos . doblePosImpar . digitosInv

-- ---------------------------------------------------------------------
-- § Referencias                                                       --
-- ---------------------------------------------------------------------

-- Esta relación es una adaptación del primer trabajo del curso "CIS 194:
-- Introduction to Haskell (Spring 2015)" de la Univ. de Pensilvania,
-- impartido por Noam Zilberstein. El trabajo se encuentra en
-- http://www.cis.upenn.edu/~cis194/hw/01-intro.pdf
--
-- En el artículo [Algoritmo de Luhn](http://bit.ly/1FGGWsC) de la
-- Wikipedia se encuentra información del algoritmo
```

# Relación 7

# Funciones de orden superior y definiciones por plegados

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

-- Esta relación contiene ejercicios con funciones de orden superior y
-- definiciones por plegado correspondientes al tema 7
-- http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-7.html

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    segmentos :: (a -> Bool) -> [a] -> [a]
-- tal que (segmentos p xs) es la lista de los segmentos de xs cuyos
-- elementos verifican la propiedad p. Por ejemplo,
--    segmentos even [1,2,0,4,9,6,4,5,7,2]  ==  [[2,0,4],[6,4],[2]]
--    segmentos odd  [1,2,0,4,9,6,4,5,7,2]  ==  [[1],[9],[5,7]]
-- ---------------------------------------------------------------------

segmentos :: (a -> Bool) -> [a] -> [[a]]
segmentos _ [] = []
segmentos p (x:xs)
```

```
  | p x       = takeWhile p (x:xs) : segmentos p (dropWhile p xs)
  | otherwise = segmentos p xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por comprensión, la función
--    relacionadosC :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionadosC r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
--    relacionadosC (<) [2,3,7,9]               ==  True
--    relacionadosC (<) [2,3,1,9]               ==  False
-- ---------------------------------------------------------------------

relacionadosC :: (a -> a -> Bool) -> [a] -> Bool
relacionadosC r xs = and [r x y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    relacionadosR :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionadosR r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
--    relacionadosR (<) [2,3,7,9]               ==  True
--    relacionadosR (<) [2,3,1,9]               ==  False
-- ---------------------------------------------------------------------

relacionadosR :: (a -> a -> Bool) -> [a] -> Bool
relacionadosR r (x:y:zs) = r x y && relacionadosR r (y:zs)
relacionadosR _ _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    agrupa :: Eq a => [[a]] -> [[a]]
-- tal que (agrupa xss) es la lista de las listas obtenidas agrupando
-- los primeros elementos, los segundos, ... Por ejemplo,
--    agrupa [[1..6],[7..9],[10..20]]  ==  [[1,7,10],[2,8,11],[3,9,12]]
--    agrupa []                        ==  []
-- ---------------------------------------------------------------------

agrupa :: Eq a => [[a]] -> [[a]]
agrupa []  = []
agrupa xss
```

```
  | [] `elem` xss = []
  | otherwise     = primeros xss : agrupa (restos xss)
  where primeros = map head
        restos   = map tail

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickChek que la longitud de todos los
-- elementos de (agrupa xs) es igual a la longitud de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_agrupa :: [[Int]] -> Bool
prop_agrupa xss =
  and [length xs == n | xs <- agrupa xss]
  where n = length xss

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir, por recursión, la función
--    concatR :: [[a]] -> [a]
-- tal que (concatR xss) es la concatenación de las listas de xss. Por
-- ejemplo,
--    concatR [[1,3],[2,4,6],[1,9]]  ==  [1,3,2,4,6,1,9]
-- ---------------------------------------------------------------------

concatR :: [[a]] -> [a]
concatR []       = []
concatR (xs:xss) = xs ++ concatR xss

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, usando foldr, la función
--    concatP :: [[a]] -> [a]
-- tal que (concatP xss) es la concatenación de las listas de xss. Por
-- ejemplo,
--    concatP [[1,3],[2,4,6],[1,9]]  ==  [1,3,2,4,6,1,9]
-- ---------------------------------------------------------------------

concatP :: [[a]] -> [a]
concatP = foldr (++) []

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

```haskell
-- Ejercicio 4.3. Comprobar con QuickCheck que la funciones concatR,
-- concatP y concat son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_concat :: [[Int]] -> Bool
prop_concat xss =
  concatR xss == ys && concatP xss == ys
  where ys = concat xss

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Comprobar con QuickCheck que la longitud de
-- (concatP xss) es la suma de las longitudes de los elementos de xss.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_longConcat :: [[Int]] -> Bool
prop_longConcat xss =
  length (concatP xss) == sum [length xs | xs <- xss]

-- La comprobación es
--    ghci> quickCheck prop_longConcat
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, por comprensión, la función
--    filtraAplicaC :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaC f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaC (4+) (<3) [1..7]  =>  [5,6]
-- ---------------------------------------------------------------------

filtraAplicaC :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplicaC f p xs = [f x | x <- xs, p x]

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

```haskell
-- Ejercicio 5.2. Definir, usando map y filter, la función
--    filtraAplicaMF :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaMF f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaMF (4+) (<3) [1..7]  =>  [5,6]
-- ---------------------------------------------------------------------

filtraAplicaMF :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplicaMF f p xs = map f (filter p xs)

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir, por recursión, la función
--    filtraAplicaR :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaR f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaR (4+) (<3) [1..7]  =>  [5,6]
-- ---------------------------------------------------------------------

filtraAplicaR :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplicaR _ _ [] = []
filtraAplicaR f p (x:xs) | p x       = f x : filtraAplicaR f p xs
                         | otherwise = filtraAplicaR f p xs

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Definir, por plegado, la función
--    filtraAplicaP :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaP f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaP (4+) (<3) [1..7]  =>  [5,6]
-- ---------------------------------------------------------------------

filtraAplicaP :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplicaP f p = foldr g []
  where g x y | p x       = f x : y
              | otherwise = y

-- La definición por plegado usando lambda es
filtraAplicaP2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplicaP2 f p =
  foldr (\x y -> if p x then f x : y else y) []
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir, mediante recursión, la función
--    maximumR :: Ord a => [a] -> a
-- tal que (maximumR xs) es el máximo de la lista xs. Por ejemplo,
--    maximumR [3,7,2,5]                  ==  7
--    maximumR ["todo","es","falso"]      ==  "todo"
--    maximumR ["menos","alguna","cosa"]  ==  "menos"
--
-- Nota: La función maximumR es equivalente a la predefinida maximum.
-- ---------------------------------------------------------------------

maximumR :: Ord a => [a] -> a
maximumR [x]      = x
maximumR (x:y:ys) = max x (maximumR (y:ys))
maximumR _        = error "Imposible"

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. La función de plegado foldr1 está definida por
--    foldr1 :: (a -> a -> a) -> [a] -> a
--    foldr1 _ [x]    =  x
--    foldr1 f (x:xs) =  f x (foldr1 f xs)
--
-- Definir, mediante plegado con foldr1, la función
--    maximumP :: Ord a => [a] -> a
-- tal que (maximumR xs) es el máximo de la lista xs. Por ejemplo,
--    maximumP [3,7,2,5]                  ==  7
--    maximumP ["todo","es","falso"]      ==  "todo"
--    maximumP ["menos","alguna","cosa"]  ==  "menos"
--
-- Nota: La función maximumP es equivalente a la predefinida maximum.
-- ---------------------------------------------------------------------

maximumP :: Ord a => [a] -> a
maximumP = foldr1 max
```

# Relación 8

# Funciones sobre cadenas

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

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

-- ------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    sumaDigitosC :: String -> Int
-- tal que (sumaDigitosC xs) es la suma de los dígitos de la cadena
-- xs. Por ejemplo,
--    sumaDigitosC "SE 2431 X"  ==  10
-- Nota: Usar las funciones (isDigit c) que se verifica si el carácter c
-- es un dígito y (digitToInt d) que es el entero correspondiente al
-- dígito d.
-- ------------------------------------------------------------------

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

-- ------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    sumaDigitosR :: String -> Int
-- tal que (sumaDigitosR xs) es la suma de los dígitos de la cadena
-- xs. Por ejemplo,
--    sumaDigitosR "SE 2431 X"  ==  10
```

```haskell
-- Nota: Usar las funciones isDigit y digitToInt.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaDigitosC :: String -> Bool
prop_sumaDigitosC xs =
  sumaDigitosC xs == sumaDigitosR xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por comprensión, la función
--    mayusculaInicial :: String -> String
-- tal que (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
--    mayusculaInicial "sEviLLa"  ==  "Sevilla"
--    mayusculaInicial ""         ==  ""
-- Nota: Usar las funciones (toLower c) que es el carácter c en
-- minúscula y (toUpper c) que es el carácter c en mayúscula.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
```

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

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

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

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

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

-- También se puede definir
comprueba_mayusculaInicial :: IO ()
comprueba_mayusculaInicial =
  quickCheck prop_mayusculaInicial

-- La comprobación es
--    λ> comprueba_mayusculaInicial
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Se consideran las siguientes reglas de mayúsculas
-- iniciales para los títulos:
--    * la primera palabra comienza en mayúscula y
```

```haskell
--    * todas las palabras que tienen 4 letras como mínimo empiezan
--      con mayúsculas
-- Definir, por comprensión, la función
--    titulo :: [String] -> [String]
-- tal que (titulo ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--    ghci> titulo ["eL","arTE","DE","La","proGraMacion"]
--    ["El","Arte","de","la","Programacion"]
-- ---------------------------------------------------------------------

titulo :: [String] -> [String]
titulo []     = []
titulo (p:ps) = mayusculaInicial p : [transforma q | q <- 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 3.2. Definir, por recursión, la función
--    tituloRec :: [String] -> [String]
-- tal que (tituloRec ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--    ghci> 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 (q:qs) = transforma q : tituloRecAux qs

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

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

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

-- La comprobación es
--    ghci> quickCheck prop_titulo
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir, por comprensión, la función
--    posiciones :: String -> Char -> [Int]
-- tal que (posiciones xs y) es la lista de la posiciones del carácter y
-- en la cadena xs. Por ejemplo,
--    posiciones "Salamamca" 'a'  ==  [1,3,5,8]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por recursión, la función
--    posicionesR :: String -> Char -> [Int]
-- tal que (posicionesR xs y) es la lista de la posiciones del
-- carácter y en la cadena xs. Por ejemplo,
--    posicionesR "Salamamca" 'a'  ==  [1,3,5,8]
-- ---------------------------------------------------------------------

posicionesR :: String -> Char -> [Int]
posicionesR xs y = posicionesAux xs y 0
  where
    posicionesAux [] _ _ = []
    posicionesAux (a:as) b n | a == b    = n : posicionesAux as b (n+1)
                             | otherwise = posicionesAux as b (n+1)

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

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

-- La propiedad es
prop_posiciones :: String -> Char -> Bool
prop_posiciones xs y =
  posiciones xs y == posicionesR xs y

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, por recursión, la función
--    contieneR :: String -> String -> Bool
-- tal que (contieneR xs ys) se verifica si ys es una subcadena de
-- xs. Por ejemplo,
--    contieneR "escasamente" "casa"   ==  True
--    contieneR "escasamente" "cante"  ==  False
--    contieneR "" ""                  ==  True
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica
-- si ys es un prefijo de xs.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir, por comprensión, la función
--    contiene :: String -> String -> Bool
-- tal que (contiene xs ys) se verifica si ys es una subcadena de
-- xs. Por ejemplo,
--    contiene "escasamente" "casa"      ==  True
--    contiene "escasamente" "cante"     ==  False
--    contiene "casado y casada" "casa"  ==  True
--    contiene "" ""                     ==  True
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica
-- si ys es un prefijo de xs.
```

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

contiene :: String -> String -> Bool
contiene xs ys =
  or [ys `isPrefixOf` zs | zs <- sufijos xs]

-- (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]]

-- Notas:
-- 1. La función sufijos es equivalente a la predefinida tails.
-- 2. contiene se puede definir usando la predefinida isInfixOf

contiene2 :: String -> String -> Bool
contiene2 xs ys = ys `isInfixOf` xs

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

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

-- La comprobación es
--    ghci> quickCheck prop_contiene
--    +++ OK, passed 100 tests.
```

# Relación 9

# Tipos de datos algebraicos: Árboles binarios

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

-- En esta relación se presenta ejercicios sobre árboles binarios
-- definidos como tipos de datos algebraicos.
--
-- Los ejercicios corresponden al tema 9 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-9.html

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Control.Monad

-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se trabajará con los árboles
-- binarios definidos como sigue
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- Por ejemplo, el árbol
--         9
--        / \
```

```haskell
--       /   \
--      3     7
--     / \
--    2   4
-- se representa por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
-- ---------------------------------------------------------------------

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
             deriving (Show, Eq)

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    nHojas :: Arbol a -> Int
-- tal que (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--    nHojas (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  3
-- ---------------------------------------------------------------------

nHojas :: Arbol a -> Int
nHojas (H _)     = 1
nHojas (N _ i d) = nHojas i + nHojas d

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    nNodos :: Arbol a -> Int
-- tal que (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--    nNodos (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  2
-- ---------------------------------------------------------------------

nNodos :: Arbol a -> Int
nNodos (H _)     = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d

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

```haskell
prop_nHojas :: Arbol Int -> Bool
prop_nHojas x =
  nHojas x == nNodos x + 1

-- La comprobación es
--    ghci> quickCheck prop_nHojas
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    profundidad :: Arbol a -> Int
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
--    profundidad (N 9 (N 3 (H 2) (H 4)) (H 7))              ==  2
--    profundidad (N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7))  ==  3
--    profundidad (N 4 (N 5 (H 4) (H 2)) (N 3 (H 7) (H 4)))  ==  2
-- ---------------------------------------------------------------------

profundidad :: Arbol a -> Int
profundidad (H _)     = 0
profundidad (N _ i d) = 1 + max (profundidad i) (profundidad d)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. 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
--    ghci> quickCheck prop_nNodosProfundidad
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    preorden :: Arbol a -> [a]
-- tal que (preorden x) es la lista correspondiente al recorrido
```

```haskell
-- 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,
--    preorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [9,3,2,4,7]
-- ---------------------------------------------------------------------

preorden :: Arbol a -> [a]
preorden (H x)     = [x]
preorden (N x i d) = x : preorden i ++ preorden d

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol más el número de hojas.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_length_preorden :: Arbol Int -> Bool
prop_length_preorden x =
   length (preorden x) == nNodos x + nHojas x

-- La comprobación es
--    ghci> quickCheck prop_length_preorden
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    postorden :: Arbol a -> [a]
-- tal que (postorden x) es la lista correspondiente al recorrido
-- postorden del árbol x; es decir, primero recorre el subárbol
-- izquierdo, a continuación el subárbol derecho y, finalmente, la raíz
-- del árbol. Por ejemplo,
--    postorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [2,4,3,7,9]
-- ---------------------------------------------------------------------

postorden :: Arbol a -> [a]
postorden (H x)     = [x]
postorden (N x i d) = postorden i ++ postorden d ++ [x]

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

```haskell
-- Ejercicio 3.4. 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,
--    preordenIt (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [9,3,2,4,7]
--
-- Nota: No usar (++) en la definición
-- ---------------------------------------------------------------------

preordenIt :: Arbol a -> [a]
preordenIt x' = preordenItAux x' []
    where preordenItAux (H x) xs     = x:xs
          preordenItAux (N x i d) xs =
              x : preordenItAux i (preordenItAux d xs)

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. 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
--    ghci> quickCheck prop_preordenIt
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--    espejo (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 7) (N 3 (H 4) (H 2))
-- ---------------------------------------------------------------------

espejo :: Arbol a -> Arbol a
espejo (H x)     = H x
```

```haskell
espejo (N x i d) = N x (espejo d) (espejo i)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que para todo árbol x,
--    espejo (espejo x) = x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_espejo :: Arbol Int -> Bool
prop_espejo x =
  espejo (espejo x) == x

-- La comprobación es
--    ghci> quickCheck prop_espejo
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. 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
--    ghci> quickCheck prop_reverse_preorden_espejo
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. 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)
```

```haskell
-- La comprobación es
--    ghci> quickCheck prop_recorrido
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. La función take está definida por
--    take :: Int -> [a] -> [a]
--    take 0            = []
--    take (n+1) []     = []
--    take (n+1) (x:xs) = x : take n xs
--
-- Definir la función
--    takeArbol ::  Int -> Arbol a -> Arbol a
-- tal que (takeArbol n t) es el subárbol de t de profundidad n. Por
-- ejemplo,
--    takeArbol 0 (N 9 (N 3 (H 2) (H 4)) (H 7)) == H 9
--    takeArbol 1 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 3) (H 7)
--    takeArbol 2 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--    takeArbol 3 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
-- ---------------------------------------------------------------------

takeArbol :: Int -> Arbol a -> Arbol a
takeArbol _ (H x)     = H x
takeArbol 0 (N x _ _) = H x
takeArbol n (N x i d) =
    N x (takeArbol (n-1) i) (takeArbol (n-1) d)

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que la profundidad de
-- (takeArbol n x) es menor o igual que n, para todo número natural n y
-- todo árbol x.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_takeArbol :: Int -> Arbol Int -> Property
prop_takeArbol n x =
  n >= 0 ==> profundidad (takeArbol n x) <= n

-- La comprobación es
```

```
--    ghci> quickCheck prop_takeArbol
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. 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,...
-- 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,
--    takeArbol 0 (repeatArbol 3) == H 3
--    takeArbol 1 (repeatArbol 3) == N 3 (H 3) (H 3)
--    takeArbol 2 (repeatArbol 3) == N 3 (N 3 (H 3) (H 3)) (N 3 (H 3) (H 3))
-- ---------------------------------------------------------------------

repeatArbol :: a -> Arbol a
repeatArbol x = N x t t
  where t = repeatArbol x

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. La función
--    replicate :: Int -> a -> [a]
-- está definida por
--    replicate n = take n . repeat
-- es tal que (replicate n x) es la lista de longitud n cuyos elementos
-- son x. Por ejemplo,
--    replicate 3 5  ==  [5,5,5]
--
-- Definir la función
--    replicateArbol :: Int -> a -> Arbol a
-- tal que (replicate n x) es el árbol de profundidad n cuyos nodos son
-- x. Por ejemplo,
--    replicateArbol 0 5  ==  H 5
--    replicateArbol 1 5  ==  N 5 (H 5) (H 5)
```

```haskell
--    replicateArbol 2 5  ==  N 5 (N 5 (H 5) (H 5)) (N 5 (H 5) (H 5))
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Comprobar con QuickCheck que el número de hojas de
-- (replicateArbol n x) es 2^n, para todo número natural n
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=7}) prop_replicateArbol
-- ---------------------------------------------------------------------

-- La propiedad es
prop_replicateArbol :: Int -> Int -> Property
prop_replicateArbol n x =
  n >= 0 ==> nHojas (replicateArbol n x) == 2^n

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_replicateArbol
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir la función
--    mapArbol :: (a -> a) -> Arbol a -> Arbol a
-- tal que (mapArbol f x) es el árbol obtenido aplicándole a cada nodo de
-- x la función f. Por ejemplo,
--    ghci> mapArbol (*2) (N 9 (N 3 (H 2) (H 4)) (H 7))
--    N 18 (N 6 (H 4) (H 8)) (H 14)
-- ---------------------------------------------------------------------

mapArbol :: (a -> a) -> Arbol a -> Arbol a
mapArbol f (H x)     = H (f x)
mapArbol f (N x i d) = N (f x) (mapArbol f i) (mapArbol f d)

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Comprobar con QuickCheck que
--    (mapArbol (1+)) . espejo = espejo . (mapArbol (1+))
```

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

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

-- La comprobación es
--    ghci> quickCheck prop_mapArbol_espejo
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Comprobar con QuickCheck que
--    (map (1+)) . preorden = preorden . (mapArbol (1+))
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- 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       = fmap H arbitrary
      arbol n | n>0 = oneof [fmap H arbitrary,
                             liftM3 N arbitrary subarbol subarbol]
                      where subarbol = arbol (div n 2)
      arbol _       = error "Imposible"
```

# Relación 10

# Tipos de datos algebraicos

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

-- En esta relación se presenta ejercicios sobre distintos tipos de
-- datos algebraicos. Concretamente,
--    * Árboles binarios:
--      + Árboles binarios con valores en los nodos.
--      + Árboles binarios con valores en las hojas.
--      + Árboles binarios con valores en las hojas y en los nodos.
--      + Árboles booleanos.
--    * Árboles generales
--    * Expresiones aritméticas
--      + Expresiones aritméticas básicas.
--      + Expresiones aritméticas con una variable.
--      + Expresiones aritméticas con varias variables.
--      + Expresiones aritméticas generales.
--      + Expresiones aritméticas con tipo de operaciones.
--    * Expresiones vectoriales
--
-- Los ejercicios corresponden al tema 9 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-9.html

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Los árboles binarios con valores en los nodos se
-- pueden definir por
--    data Arbol1 a = H1
--                  | N1 a (Arbol1 a) (Arbol1 a)
```

```haskell
--                   deriving (Show, Eq)
-- Por ejemplo, el árbol
--          9
--         / \
--        /   \
--       8     6
--      / \   / \
--     3   2 4   5
-- se puede representar por
--    N1 9 (N1 8 (N1 3 H1 H1) (N1 2 H1 H1)) (N1 6 (N1 4 H1 H1) (N1 5 H1 H1))
--
-- Definir por recursión la función
--    sumaArbol :: Num a => Arbol1 a -> a
-- tal (sumaArbol x) es la suma de los valores que hay en el árbol
-- x. Por ejemplo,
--    ghci> sumaArbol (N1 2 (N1 5 (N1 3 H1 H1) (N1 7 H1 H1)) (N1 4 H1 H1))
--    21
-- ---------------------------------------------------------------------

data Arbol1 a = H1
              | N1 a (Arbol1 a) (Arbol1 a)
  deriving (Show, Eq)

sumaArbol :: Num a => Arbol1 a -> a
sumaArbol H1         = 0
sumaArbol (N1 x i d) = x + sumaArbol i + sumaArbol d

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    mapArbol :: (a -> b) -> Arbol1 a -> Arbol1 b
-- tal que (mapArbol f x) es el árbol que resulta de sustituir cada nodo
-- n del árbol x por (f n). Por ejemplo,
--    ghci> mapArbol (+1) (N1 2 (N1 5 (N1 3 H1 H1) (N1 7 H1 H1)) (N1 4 H1 H1))
--    N1 3 (N1 6 (N1 4 H1 H1) (N1 8 H1 H1)) (N1 5 H1 H1)
-- ---------------------------------------------------------------------

mapArbol :: (a -> b) -> Arbol1 a -> Arbol1 b
mapArbol _ H1         = H1
mapArbol f (N1 x i d) = N1 (f x) (mapArbol f i) (mapArbol f d)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    ramaIzquierda :: Arbol1 a -> [a]
-- tal que (ramaIzquierda a) es la lista de los valores de los nodos de
-- la rama izquierda del árbol a. Por ejemplo,
--    ghci> ramaIzquierda (N1 2 (N1 5 (N1 3 H1 H1) (N1 7 H1 H1)) (N1 4 H1 H1))
--    [2,5,3]
-- ---------------------------------------------------------------------

ramaIzquierda :: Arbol1 a -> [a]
ramaIzquierda H1         = []
ramaIzquierda (N1 x i _) = x : ramaIzquierda i

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Diremos que un árbol está balanceado si para cada nodo
-- v la diferencia entre el número de nodos (con valor) de sus subárboles
-- izquierdo y derecho es menor o igual que uno.
--
-- Definir la función
--    balanceado :: Arbol1 a -> Bool
-- tal que (balanceado a) se verifica si el árbol a está balanceado. Por
-- ejemplo,
--    balanceado (N1 5 H1 (N1 3 H1 H1))           == True
--    balanceado (N1 5 H1 (N1 3 (N1 4 H1 H1) H1)) == False
-- ---------------------------------------------------------------------

balanceado :: Arbol1 a -> Bool
balanceado H1         = True
balanceado (N1 _ i d) = abs (numeroNodos i - numeroNodos d) <= 1

-- (numeroNodos a) es el número de nodos del árbol a. Por ejemplo,
--    numeroNodos (N1 5 H1 (N1 3 H1 H1)) ==  2
numeroNodos :: Arbol1 a -> Int
numeroNodos H1         = 0
numeroNodos (N1 _ i d) = 1 + numeroNodos i + numeroNodos d

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios con valores en las hojas se pueden
-- definir por
--    data Arbol2 a = H2 a
```

```
--                  | N2 (Arbol2 a) (Arbol2 a)
--                  deriving Show
-- Por ejemplo, los árboles
--    árbol1          árbol2       árbol3     árbol4
--       o              o            o          o
--      / \            / \          / \        / \
--     1   o          o   3        o   3      o   1
--        / \        / \          / \        / \
--       2   3      1   2        1   4      2   3
-- se representan por
--    arbol1, arbol2, arbol3, arbol4 :: Arbol2 Int
--    arbol1 = N2 (H2 1) (N2 (H2 2) (H2 3))
--    arbol2 = N2 (N2 (H2 1) (H2 2)) (H2 3)
--    arbol3 = N2 (N2 (H2 1) (H2 4)) (H2 3)
--    arbol4 = N2 (N2 (H2 2) (H2 3)) (H2 1)
--
-- Definir la función
--    igualBorde :: Eq a => Arbol2 a -> Arbol2 a -> Bool
-- tal que (igualBorde t1 t2) se verifica si los bordes de los árboles
-- t1 y t2 son iguales. Por ejemplo,
--    igualBorde arbol1 arbol2  ==  True
--    igualBorde arbol1 arbol3  ==  False
--    igualBorde arbol1 arbol4  ==  False
-- ---------------------------------------------------------------------

data Arbol2 a = N2 (Arbol2 a) (Arbol2 a)
              | H2 a
              deriving Show

arbol1, arbol2, arbol3, arbol4 :: Arbol2 Int
arbol1 = N2 (H2 1) (N2 (H2 2) (H2 3))
arbol2 = N2 (N2 (H2 1) (H2 2)) (H2 3)
arbol3 = N2 (N2 (H2 1) (H2 4)) (H2 3)
arbol4 = N2 (N2 (H2 2) (H2 3)) (H2 1)

igualBorde :: Eq a => Arbol2 a -> Arbol2 a -> Bool
igualBorde t1 t2 = borde t1 == borde t2

-- (borde t) es el borde del árbol t; es decir, la lista de las hojas
-- del árbol t leídas de izquierda a derecha. Por ejemplo,
```

```haskell
--    borde arbol4  ==  [2,3,1]
borde :: Arbol2 a -> [a]
borde (N2 i d) = borde i ++ borde d
borde (H2 x)   = [x]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol3 a = H3 a
--                  | N3 a (Arbol3 a) (Arbol3 a)
--                  deriving Show
-- Por ejemplo, los árboles
--         5              8             5           5
--        / \            / \           / \         / \
--       /   \          /   \         /   \       /   \
--      9     7        9     3       9     2     4     7
--     / \   / \      / \   / \     / \               / \
--    1   4 6   8    1   4 6   2   1   4             6   2
-- se pueden representar por
--    ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol3 Int
--    ej3arbol1 = N3 5 (N3 9 (H3 1) (H3 4)) (N3 7 (H3 6) (H3 8))
--    ej3arbol2 = N3 8 (N3 9 (H3 1) (H3 4)) (N3 3 (H3 6) (H3 2))
--    ej3arbol3 = N3 5 (N3 9 (H3 1) (H3 4)) (H3 2)
--    ej3arbol4 = N3 5 (H3 4) (N3 7 (H3 6) (H3 2))
--
-- Definir la función
--    igualEstructura :: Arbol3 -> Arbol3 -> Bool
-- tal que (igualEstructura a1 a1) se verifica si los árboles a1 y a2
-- tienen la misma estructura. Por ejemplo,
--    igualEstructura ej3arbol1 ej3arbol2 == True
--    igualEstructura ej3arbol1 ej3arbol3 == False
--    igualEstructura ej3arbol1 ej3arbol4 == False
-- ---------------------------------------------------------------------

data Arbol3 a = H3 a
              | N3 a (Arbol3 a) (Arbol3 a)
              deriving (Show, Eq)

ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol3 Int
ej3arbol1 = N3 5 (N3 9 (H3 1) (H3 4)) (N3 7 (H3 6) (H3 8))
```

```haskell
ej3arbol2 = N3 8 (N3 9 (H3 1) (H3 4)) (N3 3 (H3 6) (H3 2))
ej3arbol3 = N3 5 (N3 9 (H3 1) (H3 4)) (H3 2)
ej3arbol4 = N3 5 (H3 4) (N3 7 (H3 6) (H3 2))

igualEstructura :: Arbol3 a -> Arbol3 a -> Bool
igualEstructura (H3 _) (H3 _)             = True
igualEstructura (N3 _ i1 d1) (N3 _ i2 d2) =
  igualEstructura i1 i2 &&
  igualEstructura d1 d2
igualEstructura _ _                       = False

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    algunoArbol :: Arbol3 t -> (t -> Bool) -> Bool
-- tal que (algunoArbol a p) se verifica si algún elemento del árbol a
-- cumple la propiedad p. Por ejemplo,
--    algunoArbol (N3 5 (N3 3 (H3 1) (H3 4)) (H3 2)) (>4)  ==  True
--    algunoArbol (N3 5 (N3 3 (H3 1) (H3 4)) (H3 2)) (>7)  ==  False
-- ---------------------------------------------------------------------

algunoArbol :: Arbol3 a -> (a -> Bool) -> Bool
algunoArbol (H3 x) p     = p x
algunoArbol (N3 x i d) p = p x || algunoArbol i p || algunoArbol d p

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Un elemento de un árbol se dirá de nivel k si aparece
-- en el árbol a distancia k  de la raíz.
--
-- Definir la función
--    nivel :: Int -> Arbol3 a -> [a]
-- tal que (nivel k a) es la lista de los elementos de nivel k del árbol
-- a. Por ejemplo,
--    nivel 0 (N3 7 (N3 2 (H3 5) (H3 4)) (H3 9))  ==  [7]
--    nivel 1 (N3 7 (N3 2 (H3 5) (H3 4)) (H3 9))  ==  [2,9]
--    nivel 2 (N3 7 (N3 2 (H3 5) (H3 4)) (H3 9))  ==  [5,4]
--    nivel 3 (N3 7 (N3 2 (H3 5) (H3 4)) (H3 9))  ==  []
-- ---------------------------------------------------------------------

nivel :: Int -> Arbol3 a -> [a]
nivel 0 (H3 x)     = [x]
```

```haskell
nivel 0 (N3 x _  _) = [x]
nivel _ (H3 _ )     = []
nivel k (N3 _ i d)  = nivel (k-1) i ++ nivel (k-1) d

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Los divisores medios de un número son los que ocupan
-- la posición media entre los divisores de n, ordenados de menor a
-- mayor. Por ejemplo, los divisores de 60 son
-- [1,2,3,4,5,6,10,12,15,20,30,60] y sus divisores medios son 6 y 10.
--
-- El árbol de factorización de un número compuesto n se construye de la
-- siguiente manera:
--    * la raíz es el número n,
--    * la rama izquierda es el árbol de factorización de su divisor
--      medio menor y
--    * la rama derecha es el árbol de factorización de su divisor
--      medio mayor
-- Si el número es primo, su árbol de factorización sólo tiene una hoja
-- con dicho número. Por ejemplo, el árbol de factorización de 60 es
--        60
--       /  \
--      6    10
--     / \   / \
--    2   3 2   5
--
-- Definir la función
--    arbolFactorizacion :: Int -> Arbol3
-- tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
-- ejemplo,
--    arbolFactorizacion 60 == N3 60 (N3 6 (H3 2) (H3 3)) (N3 10 (H3 2) (H3 5))
--    arbolFactorizacion 45 == N3 45 (H3 5) (N3 9 (H3 3) (H3 3))
--    arbolFactorizacion 7  == H3 7
--    arbolFactorizacion 9  == N3 9 (H3 3) (H3 3)
--    arbolFactorizacion 14 == N3 14 (H3 2) (H3 7)
--    arbolFactorizacion 28 == N3 28 (N3 4 (H3 2) (H3 2)) (H3 7)
--    arbolFactorizacion 84 == N3 84 (H3 7) (N3 12 (H3 3) (N3 4 (H3 2) (H3 2)))
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============
```

```haskell
arbolFactorizacion :: Int -> Arbol3 Int
arbolFactorizacion n
  | esPrimo n = H3 n
  | otherwise = N3 n (arbolFactorizacion x) (arbolFactorizacion y)
  where (x,y) = divisoresMedio n

-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 9  ==  False
esPrimo :: Int -> Bool
esPrimo n = divisores n == [1,n]

-- (divisoresMedio n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio 30  ==  (5,6)
--    divisoresMedio  7  ==  (1,7)
--    divisoresMedio 16  ==  (4,4)
divisoresMedio :: Int -> (Int,Int)
divisoresMedio n = (n `div` x,x)
  where xs = divisores n
        x  = xs !! (length xs `div` 2)

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--    divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª definición
-- =============
arbolFactorizacion2 :: Int -> Arbol3 Int
arbolFactorizacion2 n
    | x == 1    = H3 n
    | otherwise = N3 n (arbolFactorizacion x) (arbolFactorizacion y)
    where (x,y) = divisoresMedio n

-- (divisoresMedio2 n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio2 30  ==  (5,6)
--    divisoresMedio2  7  ==  (1,7)
divisoresMedio2 :: Int -> (Int,Int)
```

```haskell
divisoresMedio2 n = (n `div` x,x)
  where m = ceiling (sqrt (fromIntegral n))
        x = head [y | y <- [m..n], n `rem` y == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los árboles con operaciones booleanas
-- definidos por
--    data ArbolB = HB Bool
--                | Conj ArbolB ArbolB
--                | Disy ArbolB ArbolB
--                | Neg ArbolB
--
-- Por ejemplo, los árboles
--                Conj                              Conj
--               /   \                             /   \
--              /     \                           /     \
--           Disy      Conj                    Disy      Conj
--          /   \       /  \                  /   \      /   \
--       Conj    Neg   Neg True           Conj    Neg   Neg  True
--       /  \    |     |                  /  \    |     |
--    True False False False           True False True  False
--
-- se definen por
--    ej1, ej2:: ArbolB
--    ej1 = Conj (Disy (Conj (HB True) (HB False))
--                     (Neg (HB False)))
--               (Conj (Neg (HB False))
--                     (HB True))
--
--    ej2 = Conj (Disy (Conj (HB True) (HB False))
--                     (Neg (HB True)))
--               (Conj (Neg (HB False))
--                     (HB True))
--
-- Definir la función
--    valorB :: ArbolB -> Bool
-- tal que (valorB ar) es el resultado de procesar el árbol realizando
-- las operaciones booleanas especificadas en los nodos. Por ejemplo,
--    valorB ej1 == True
--    valorB ej2 == False
```

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

data ArbolB = HB Bool
            | Conj ArbolB ArbolB
            | Disy ArbolB ArbolB
            | Neg ArbolB

ej1, ej2 :: ArbolB
ej1 = Conj (Disy (Conj (HB True) (HB False))
                 (Neg (HB False)))
           (Conj (Neg (HB False))
                 (HB True))

ej2 = Conj (Disy (Conj (HB True) (HB False))
                 (Neg (HB True)))
           (Conj (Neg (HB False))
                 (HB True))

valorB :: ArbolB -> Bool
valorB (HB x)     = x
valorB (Neg a)    = not (valorB a)
valorB (Conj i d) = valorB i && valorB d
valorB (Disy i d) = valorB i || valorB d

-- ---------------------------------------------------------------------
-- Ejercicio 5. Los árboles generales se pueden representar mediante el
-- siguiente tipo de dato
--    data ArbolG a = N a [ArbolG a]
--                  deriving (Eq, Show)
-- Por ejemplo, los árboles
--      1              3              3
--     / \            /|\            / | \
--    2   3          / | \          /  |  \
--        |         5  4  7        5   4   7
--        4         |     /\       |   |  / \
--                  6    2  1      6   1 2   1
--                                    / \
--                                   2   3
--                                       |
--                                       4
```

```haskell
-- se representan por
--    ejG1, ejG2, ejG3 :: ArbolG Int
--    ejG1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ejG2 = N 3 [N 5 [N 6 []],
--                N 4 [],
--                N 7 [N 2 [], N 1 []]]
--    ejG3 = N 3 [N 5 [N 6 []],
--                N 4 [N 1 [N 2 [],N 3 [N 4 []]]],
--                N 7 [N 2 [], N 1 []]]
--
-- Definir la función
--     ramifica :: ArbolG a -> ArbolG a -> (a -> Bool) -> ArbolG a
-- tal que (ramifica a1 a2 p) el árbol que resulta de añadir una copia
-- del árbol a2 a los nodos de a1 que cumplen un predicado p. Por
-- ejemplo,
--    ghci> ramifica ejG1 (N 8 []) (>4)
--    N 1 [N 2 [],N 3 [N 4 []]]
--    ghci> ramifica ejG1 (N 8 []) (>3)
--    N 1 [N 2 [],N 3 [N 4 [N 8 []]]]
--    ghci> ramifica ejG1 (N 8 []) (>2)
--    N 1 [N 2 [],N 3 [N 4 [N 8 []],N 8 []]]
--    ghci> ramifica ejG1 (N 8 []) (>1)
--    N 1 [N 2 [N 8 []],N 3 [N 4 [N 8 []],N 8 []]]
--    ghci> ramifica ejG1 (N 8 []) (>0)
--    N 1 [N 2 [N 8 []],N 3 [N 4 [N 8 []],N 8 []],N 8 []]
-- ----------------------------------------------------------------------

data ArbolG a = N a [ArbolG a]
              deriving (Eq, Show)

ejG1, ejG2, ejG3 :: ArbolG Int
ejG1 = N 1 [N 2 [],N 3 [N 4 []]]
ejG2 = N 3 [N 5 [N 6 []],
            N 4 [],
            N 7 [N 2 [], N 1 []]]
ejG3 = N 3 [N 5 [N 6 []],
            N 4 [N 1 [N 2 [],N 3 [N 4 []]]],
            N 7 [N 2 [], N 1 []]]

ramifica :: ArbolG a -> ArbolG a -> (a -> Bool) -> ArbolG a
```

```
ramifica (N x xs) a2 p
  | p x       = N x ([ramifica a a2 p | a <- xs] ++ [a2])
  | otherwise = N x  [ramifica a a2 p | a <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Las expresiones aritméticas básicas pueden
-- representarse usando el siguiente tipo de datos
--    data Expr1 = C1 Int
--               | S1 Expr1 Expr1
--               | P1 Expr1 Expr1
--               deriving Show
-- Por ejemplo, la expresión 2*(3+7) se representa por
--    P1 (C1 2) (S1 (C1 3) (C1 7))
--
-- Definir la función
--    valor :: Expr1 -> Int
-- tal que (valor e) es el valor de la expresión aritmética e. Por
-- ejemplo,
--    valor (P1 (C1 2) (S1 (C1 3) (C1 7)))  ==  20
-- ---------------------------------------------------------------------

data Expr1 = C1 Int
           | S1 Expr1 Expr1
           | P1 Expr1 Expr1
           deriving (Show, Eq)

valor :: Expr1 -> Int
valor (C1 x)   = x
valor (S1 x y) = valor x + valor y
valor (P1 x y) = valor x * valor y

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir la función
--    aplica :: (Int -> Int) -> Expr1 -> Expr1
-- tal que (aplica f e) es la expresión obtenida aplicando la función f
-- a cada uno de los números de la expresión e. Por ejemplo,
--    ghci> aplica (+2) (s1 (p1 (c1 3) (c1 5)) (p1 (c1 6) (c1 7)))
--    s1 (p1 (c1 5) (c1 7)) (p1 (c1 8) (c1 9))
--    ghci> aplica (*2) (s1 (p1 (c1 3) (c1 5)) (p1 (c1 6) (c1 7)))
--    s1 (p1 (c1 6) (c1 10)) (p1 (c1 12) (c1 14))
```

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

aplica :: (Int -> Int) -> Expr1 -> Expr1
aplica f (C1 x)     = C1 (f x)
aplica f (S1 e1 e2) = S1 (aplica f e1) (aplica f e2)
aplica f (P1 e1 e2) = P1 (aplica f e1) (aplica f e2)

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Las expresiones aritméticas construidas con una
-- variable (denotada por X), los números enteros y las operaciones de
-- sumar y multiplicar se pueden representar mediante el tipo de datos
-- Expr2 definido por
--    data Expr2 = X
--               | C2 Int
--               | S2 Expr2 Expr2
--               | P2 Expr2 Expr2
-- Por ejemplo, la expresión "X*(13+X)" se representa por
-- "P2 X (S2 (C2 13) X)".
--
-- Definir la función
--    valorE :: Expr2 -> Int -> Int
-- tal que (valorE e n) es el valor de la expresión e cuando se
-- sustituye su variable por n. Por ejemplo,
--    valorE (P2 X (S2 (C2 13) X)) 2  ==  30
-- ---------------------------------------------------------------------

data Expr2 = X
           | C2 Int
           | S2 Expr2 Expr2
           | P2 Expr2 Expr2

valorE :: Expr2 -> Int -> Int
valorE X          n = n
valorE (C2 a)     _ = a
valorE (S2 e1 e2) n = valorE e1 n + valorE e2 n
valorE (P2 e1 e2) n = valorE e1 n * valorE e2 n

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir la función
--    numVars :: Expr2 -> Int
```

```haskell
-- tal que (numVars e) es el número de variables en la expresión e. Por
-- ejemplo,
--    numVars (C2 3)                 ==  0
--    numVars X                      ==  1
--    numVars (P2 X (S2 (C2 13) X))  ==  2
-- ----------------------------------------------------------------------

numVars :: Expr2 -> Int
numVars X        = 1
numVars (C2 _)   = 0
numVars (S2 a b) = numVars a + numVars b
numVars (P2 a b) = numVars a + numVars b

-- ----------------------------------------------------------------------
-- Ejercicio 8.1. Las expresiones aritméticas con variables pueden
-- representarse usando el siguiente tipo de datos
--    data Expr3 = C3 Int
--               | V3 Char
--               | S3 Expr3 Expr3
--               | P3 Expr3 Expr3
--               deriving Show
-- Por ejemplo, la expresión 2*(a+5) se representa por
--    P3 (C3 2) (S3 (V3 'a') (C3 5))
--
-- Definir la función
--    valor3 :: Expr3 -> [(Char,Int)] -> Int
-- tal que (valor3 x e) es el valor3 de la expresión x en el entorno e (es
-- decir, el valor3 de la expresión donde las variables de x se sustituyen
-- por los valores según se indican en el entorno e). Por ejemplo,
--    ghci> valor3 (P3 (C3 2) (S3 (V3 'a') (V3 'b'))) [('a',2),('b',5)]
--    14
-- ----------------------------------------------------------------------

data Expr3 = C3 Int
           | V3 Char
           | S3 Expr3 Expr3
           | P3 Expr3 Expr3
           deriving (Show, Eq)

valor3 :: Expr3 -> [(Char,Int)] -> Int
```

```haskell
valor3 (C3 x)   _ = x
valor3 (V3 x)   e = head [y | (z,y) <- e, z == x]
valor3 (S3 x y) e = valor3 x e + valor3 y e
valor3 (P3 x y) e = valor3 x e * valor3 y e

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Definir la función
--    sumas :: Expr3 -> Int
-- tal que (sumas e) es el número de sumas en la expresión e. Por
-- ejemplo,
--    sumas (P3 (V3 'z') (S3 (C3 3) (V3 'x')))  ==  1
--    sumas (S3 (V3 'z') (S3 (C3 3) (V3 'x')))  ==  2
--    sumas (P3 (V3 'z') (P3 (C3 3) (V3 'x')))  ==  0
-- ---------------------------------------------------------------------

sumas :: Expr3 -> Int
sumas (V3 _)   = 0
sumas (C3 _)   = 0
sumas (S3 x y) = 1 + sumas x + sumas y
sumas (P3 x y) = sumas x + sumas y

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Definir la función
--    sustitucion :: Expr3 -> [(Char, Int)] -> Expr3
-- tal que (sustitucion e s) es la expresión obtenida sustituyendo las
-- variables de la expresión e según se indica en la sustitución s. Por
-- ejemplo,
--    ghci> sustitucion (P3 (V3 'z') (S3 (C3 3) (V3 'x'))) [('x',7),('z',9)]
--    P3 (C3 9) (S3 (C3 3) (C3 7))
--    ghci> sustitucion (P3 (V3 'z') (S3 (C3 3) (V3 'y'))) [('x',7),('z',9)]
--    P3 (C3 9) (S3 (C3 3) (V3 'y'))
-- ---------------------------------------------------------------------

sustitucion :: Expr3 -> [(Char, Int)] -> Expr3
sustitucion e []          = e
sustitucion (V3 c) ((d,n):ps)
  | c == d    = C3 n
  | otherwise             = sustitucion (V3 c) ps
sustitucion (C3 n) _      = C3 n
sustitucion (S3 e1 e2) ps = S3 (sustitucion e1 ps) (sustitucion e2 ps)
```

```haskell
sustitucion (P3 e1 e2) ps = P3 (sustitucion e1 ps) (sustitucion e2 ps)

-- ---------------------------------------------------------------------
-- Ejercicio 8.4. Definir la función
--    reducible :: Expr3 -> Bool
-- tal que (reducible a) se verifica si a es una expresión reducible; es
-- decir, contiene una operación en la que los dos operandos son números.
-- Por ejemplo,
--    reducible (S3 (C3 3) (C3 4))               == True
--    reducible (S3 (C3 3) (V3 'x'))             == False
--    reducible (S3 (C3 3) (P3 (C3 4) (C3 5)))   == True
--    reducible (S3 (V3 'x') (P3 (C3 4) (C3 5))) == True
--    reducible (S3 (C3 3) (P3 (V3 'x') (C3 5))) == False
--    reducible (C3 3)                           == False
--    reducible (V3 'x')                         == False
-- ---------------------------------------------------------------------

reducible :: Expr3 -> Bool
reducible (C3 _)             = False
reducible (V3 _)             = False
reducible (S3 (C3 _) (C3 _)) = True
reducible (S3 a b)           = reducible a || reducible b
reducible (P3 (C3 _) (C3 _)) = True
reducible (P3 a b)           = reducible a || reducible b

-- ---------------------------------------------------------------------
-- Ejercicio 9. Las expresiones aritméticas generales se pueden definir
-- usando el siguiente tipo de datos
--    data Expr4 = C4 Int
--               | Y
--               | S4 Expr4 Expr4
--               | R4 Expr4 Expr4
--               | P4 Expr4 Expr4
--               | E4 Expr4 Int
--               deriving (Eq, Show)
-- Por ejemplo, la expresión
--    3*x - (x+2)^7
-- se puede definir por
--    R4 (P4 (C4 3) Y) (E4 (S4 Y (C4 2)) 7)
--
```

```haskell
-- Definir la función
--    maximo :: Expr4 -> [Int] -> (Int,[Int])
-- tal que (maximo e xs) es el par formado por el máximo valor de la
-- expresión e para los puntos de xs y en qué puntos alcanza el
-- máximo. Por ejemplo,
--    ghci> maximo (E4 (S4 (C4 10) (P4 (R4 (C4 1) Y) Y)) 2) [-3..3]
--    (100,[0,1])
-- ---------------------------------------------------------------------

data Expr4 = C4 Int
           | Y
           | S4 Expr4 Expr4
           | R4 Expr4 Expr4
           | P4 Expr4 Expr4
           | E4 Expr4 Int
           deriving (Eq, Show)

maximo :: Expr4 -> [Int] -> (Int,[Int])
maximo e ns = (m,[n | n <- ns, valor4 e n == m])
  where m = maximum [valor4 e n | n <- ns]

valor4 :: Expr4 -> Int -> Int
valor4 (C4 x) _ = x
valor4 Y      n = n
valor4 (S4 e1 e2) n = valor4 e1 n + valor4 e2 n
valor4 (R4 e1 e2) n = valor4 e1 n - valor4 e2 n
valor4 (P4 e1 e2) n = valor4 e1 n * valor4 e2 n
valor4 (E4 e1 m1) n = valor4 e1 n ^ m1

-- ---------------------------------------------------------------------
-- Ejercicio 10. Las operaciones de suma, resta y  multiplicación se
-- pueden representar mediante el siguiente tipo de datos
--    data Op = Su | Re | Mu
-- La expresiones aritméticas con dichas operaciones se pueden
-- representar mediante el siguiente tipo de dato algebraico
--    data Expr5 = C5 Int
--               | A Op Expr5 Expr
-- Por ejemplo, la expresión
--    (7-3)+(2*5)
-- se representa por
```

```haskell
--    A Su (A Re (C5 7) (C5 3)) (A Mu (C5 2) (C5 5))
--
-- Definir la función
--    valorEG :: Expr5 -> Int
-- tal que (valorEG e) es el valorEG de la expresión e. Por ejemplo,
--    valorEG (A Su (A Re (C5 7) (C5 3)) (A Mu (C5 2) (C5 5)))  ==  14
--    valorEG (A Mu (A Re (C5 7) (C5 3)) (A Su (C5 2) (C5 5)))  ==  28
-- ---------------------------------------------------------------------

data Op = Su | Re | Mu

data Expr5 = C5 Int | A Op Expr5 Expr5

-- 1ª definición
valorEG :: Expr5 -> Int
valorEG (C5 x)      = x
valorEG (A o e1 e2) = aplica2 o (valorEG e1) (valorEG e2)
  where aplica2 :: Op -> Int -> Int -> Int
        aplica2 Su x y = x+y
        aplica2 Re x y = x-y
        aplica2 Mu x y = x*y

-- 2ª definición
valorEG2 :: Expr5 -> Int
valorEG2 (C5 n)    = n
valorEG2 (A o x y) = (sig o) (valorEG2 x) (valorEG2 y)
  where sig Su = (+)
        sig Mu = (*)
        sig Re = (-)

-- ---------------------------------------------------------------------
-- Ejercicio 11. Se consideran las expresiones vectoriales formadas por
-- un vector, la suma de dos expresiones vectoriales o el producto de un
-- entero por una expresión vectorial. El siguiente tipo de dato define
-- las expresiones vectoriales
--    data ExpV = Vec Int Int
--              | Sum ExpV ExpV
--              | Mul Int ExpV
--              deriving Show
--
```

```haskell
-- Definir la función
--    valorEV :: ExpV -> (Int,Int)
-- tal que (valorEV e) es el valorEV de la expresión vectorial c. Por
-- ejemplo,
--    valorEV (Vec 1 2)                                    ==  (1,2)
--    valorEV (Sum (Vec 1 2 ) (Vec 3 4))                   ==  (4,6)
--    valorEV (Mul 2 (Vec 3 4))                            ==  (6,8)
--    valorEV (Mul 2 (Sum (Vec 1 2 ) (Vec 3 4)))           ==  (8,12)
--    valorEV (Sum (Mul 2 (Vec 1 2)) (Mul 2 (Vec 3 4)))    ==  (8,12)
-- ----------------------------------------------------------------------

data ExpV = Vec Int Int
          | Sum ExpV ExpV
          | Mul Int ExpV
  deriving Show

-- 1ª solución
-- ===========
valorEV :: ExpV -> (Int,Int)
valorEV (Vec x y)   = (x,y)
valorEV (Sum e1 e2) = (x1+x2,y1+y2)
  where (x1,y1) = valorEV e1
        (x2,y2) = valorEV e2
valorEV (Mul n e)   = (n*x,n*y)
  where (x,y) = valorEV e

-- 2ª solución
-- ===========
valorEV2 :: ExpV -> (Int,Int)
valorEV2 (Vec a b)   = (a, b)
valorEV2 (Sum e1 e2) = suma (valorEV2 e1) (valorEV2 e2)
valorEV2 (Mul n e1)  = multiplica n (valorEV2 e1)

suma :: (Int,Int) -> (Int,Int) -> (Int,Int)
suma (a,b) (c,d) = (a+c,b+d)

multiplica :: Int -> (Int, Int) -> (Int, Int)
multiplica n (a,b) = (n*a,n*b)
```

# Relación 11

# Listas infinitas y evaluación perezosa

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

-- En esta relación se presentan ejercicios con listas infinitas y
-- evaluación perezosa. Estos ejercicios corresponden al tema 10 que
-- se encuentra en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-10.html

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. 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
-- 1ª definición:
repite1 :: a -> [a]
repite1 x = x : repite1 x

-- 2ª definición:
repite2 :: a -> [a]
repite2 x = ys
  where ys = x:ys

-- La 2ª definición es más eficiente:
--    ghci> last (take 100000000 (repite1 5))
--    5
--    (46.56 secs, 16001567944 bytes)
--    ghci> last (take 100000000 (repite2 5))
--    5
--    (2.34 secs, 5601589608 bytes)

-- Usaremos como repite la 2ª definición
repite :: a -> [a]
repite = repite2

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por comprensión, la función
--    repiteC :: a -> [a]
-- tal que (repiteC x) es la lista infinita cuyos elementos son x. Por
-- ejemplo,
--    repiteC 5           ==  [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
--    take 3 (repiteC 5)  ==  [5,5,5]
--
-- Nota: La función repiteC es equivalente a la función repeat definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------

repiteC :: a -> [a]
repiteC x = [x | _ <- [1..]]

-- La función repite2 es más eficiente que repiteC
--    ghci> last (take 10000000 (repiteC 5))
--    5
```

```
--    (6.05 secs, 1,997,740,536 bytes)
--    ghci> last (take 10000000 (repite2 5))
--    5
--    (0.31 secs, 541,471,280 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por recursión, la función
--    repiteFinitaR :: Int-> a -> [a]
-- tal que (repiteFinitaR n x) es la lista con n elementos iguales a
-- x. Por ejemplo,
--    repiteFinitaR 3 5  ==  [5,5,5]
--
-- Nota: La función repiteFinitaR es equivalente a la función replicate
-- definida en el preludio de Haskell.
-- ---------------------------------------------------------------------

repiteFinitaR :: Int -> a -> [a]
repiteFinitaR n x | n <= 0    = []
                  | otherwise = x : repiteFinitaR (n-1) x

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por comprensión, la función
--    repiteFinitaC :: Int-> a -> [a]
-- tal que (repiteFinitaC n x) es la lista con n elementos iguales a
-- x. Por ejemplo,
--    repiteFinitaC 3 5  ==  [5,5,5]
--
-- Nota: La función repiteFinitaC es equivalente a la función replicate
-- definida en el preludio de Haskell.
-- ---------------------------------------------------------------------

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

-- La función repiteFinitaC es más eficiente que repiteFinitaR
--    ghci> last (repiteFinitaR 10000000 5)
--    5
--    (17.04 secs, 2,475,222,448 bytes)
--    ghci> last (repiteFinitaC 10000000 5)
--    5
```

```haskell
--    (5.43 secs, 1,511,227,176 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir, usando repite, la función
--    repiteFinita :: Int-> a -> [a]
-- tal que (repiteFinita n x) es la lista con n elementos iguales a
-- x. Por ejemplo,
--    repiteFinita 3 5  ==  [5,5,5]
--
-- Nota: La función repiteFinita es equivalente a la función replicate
-- definida en el preludio de Haskell.
-- ---------------------------------------------------------------------

repiteFinita :: Int -> a -> [a]
repiteFinita n x = take n (repite x)

-- La función repiteFinita es más eficiente que repiteFinitaC
--    ghci> last (repiteFinitaC 10000000 5)
--    5
--    (5.43 secs, 1,511,227,176 bytes)
--    ghci> last (repiteFinita 10000000 5)
--    5
--    (0.29 secs, 541,809,248 bytes)

-- 2ª definición
repiteFinita2 :: Int -> a -> [a]
repiteFinita2 n = take n . repite

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Comprobar con QuickCheck que las funciones
-- repiteFinitaR, repiteFinitaC y repiteFinita son equivalentes a
-- replicate.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=7}) prop_repiteFinitaEquiv
-- ---------------------------------------------------------------------

-- La propiedad es
prop_repiteFinitaEquiv :: Int -> Int -> Bool
```

```haskell
prop_repiteFinitaEquiv n x =
    repiteFinitaR n x == y &&
    repiteFinitaC n x == y &&
    repiteFinita  n x == y
    where y = replicate n x

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=20}) prop_repiteFinitaEquiv
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Comprobar con QuickCheck que la longitud de
-- (repiteFinita n x) es n, si n es positivo y 0 si no lo es.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=30}) prop_repiteFinitaLongitud
-- ---------------------------------------------------------------------

-- La propiedad es
prop_repiteFinitaLongitud :: Int -> Int -> Bool
prop_repiteFinitaLongitud n x
  | n > 0     = length (repiteFinita n x) == n
  | otherwise = null (repiteFinita n x)

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=30}) prop_repiteFinitaLongitud
--    +++ OK, passed 100 tests.

-- La expresión de la propiedad se puede simplificar
prop_repiteFinitaLongitud2 :: Int -> Int -> Bool
prop_repiteFinitaLongitud2 n x =
    length (repiteFinita n x) == (if n > 0 then n else 0)

-- ---------------------------------------------------------------------
-- Ejercicio 2.6. Comprobar con QuickCheck que todos los elementos de
-- (repiteFinita n x) son iguales a x.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_repiteFinitaIguales :: Int -> Int -> Bool
prop_repiteFinitaIguales n x =
  all (==x) (repiteFinita n x)

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=30}) prop_repiteFinitaIguales
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir, por comprensión, la función
--    ecoC :: String -> String
-- tal que (ecoC 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,
--    ecoC "abcd"  ==  "abbcccdddd"
-- ---------------------------------------------------------------------

ecoC :: String -> String
ecoC xs = concat [replicate i x | (i,x) <- zip [1..] xs]

-- 2ª definición
ecoC1 :: String -> String
ecoC1 = concat . zipWith replicate [1..]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por recursión, la función
--    ecoR :: String -> String
-- tal que (ecoR 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,
--    ecoR "abcd"  ==  "abbcccdddd"
-- ---------------------------------------------------------------------

ecoR :: String -> String
ecoR = aux 1
  where aux _ []     = []
        aux n (x:xs) = replicate n x ++ aux (n+1) xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. 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,
--    ghci> itera (+1) 3
--    [3,4,5,6,7,8,9,10,11,12,{Interrupted!}
--    ghci> itera (*2) 1
--    [1,2,4,8,16,32,64,{Interrupted!}
--    ghci> itera (`div` 10) 1972
--    [1972,197,19,1,0,0,0,0,0,0,{Interrupted!}
--
-- Nota: La función itera 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 5.1. Definir, por recursión, la función
--    agrupaR :: Int -> [a] -> [[a]]
-- tal que (agrupaR n xs) es la lista formada por listas de n elementos
-- consecutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--    ghci> agrupaR 2 [3,1,5,8,2,7]
--    [[3,1],[5,8],[2,7]]
--    ghci> agrupaR 2 [3,1,5,8,2,7,9]
--    [[3,1],[5,8],[2,7],[9]]
--    ghci> agrupaR 5 "todo necio confunde valor y precio"
--    ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-- ---------------------------------------------------------------------------

agrupaR :: Int -> [a] -> [[a]]
agrupaR _ [] = []
agrupaR n xs = take n xs : agrupaR n (drop n xs)

-- ---------------------------------------------------------------------------
-- Ejercicio 5.2. Definir, de manera no recursiva con iterate, la función
```

```haskell
--    agrupa :: Int -> [a] -> [[a]]
-- tal que (agrupa n xs) es la lista formada por listas de n elementos
-- consecutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--    ghci> agrupa 2 [3,1,5,8,2,7]
--    [[3,1],[5,8],[2,7]]
--    ghci> agrupa 2 [3,1,5,8,2,7,9]
--    [[3,1],[5,8],[2,7],[9]]
--    ghci> agrupa 5 "todo necio confunde valor y precio"
--    ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-- ---------------------------------------------------------------------

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]]

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Comprobar con QuickCheck que todos los grupos de
-- (agrupa n xs) tienen longitud n (salvo el último que puede tener una
-- longitud menor).
-- ---------------------------------------------------------------------

-- La 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
```

```haskell
--    ghci> quickCheck prop_AgrupaLongitud
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Comprobar con QuickCheck que combinando todos los
-- grupos de (agrupa n xs) se obtiene la lista xs.
-- ---------------------------------------------------------------------

-- La segunda propiedad es
prop_AgrupaCombina :: Int -> [Int] -> Property
prop_AgrupaCombina n xs =
    n > 0 ==> concat (agrupa n xs) == xs

-- La comprobación es
--    ghci> quickCheck prop_AgrupaCombina
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. 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.
--
-- Definir la función
--    siguiente :: Integer -> Integer
-- tal que (siguiente n) es el siguiente de n en la sucesión de
-- Collatz. Por ejemplo,
--    siguiente 13  ==  40
--    siguiente 40  ==  20
-- ---------------------------------------------------------------------

siguiente :: Integer -> Integer
siguiente n | even n    = n `div` 2
            | otherwise = 3*n+1

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir, por recursión, la función
--    collatzR :: Integer -> [Integer]
-- tal que (collatzR n) es la órbita de CollatzR de n hasta alcanzar el
-- 1. Por ejemplo,
--    collatzR 13  ==  [13,40,20,10,5,16,8,4,2,1]
-- ---------------------------------------------------------------------

collatzR :: Integer -> [Integer]
collatzR 1 = [1]
collatzR n = n : collatzR (siguiente n)

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir, sin recursión y con iterate, 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]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 6.4. Definir la función
--    menorCollatzMayor :: Int -> Integer
-- tal que (menorCollatzMayor x) es el menor número cuya órbita de
-- Collatz tiene más de x elementos. Por ejemplo,
--    menorCollatzMayor 100  ==  27
-- ---------------------------------------------------------------------

menorCollatzMayor :: Int -> Integer
menorCollatzMayor x = head [y | y <- [1..], length (collatz y) > x]

-- ---------------------------------------------------------------------
-- Ejercicio 6.5. 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
-- ---------------------------------------------------------------------

-- 1ª definición
menorCollatzSupera :: Integer -> Integer
menorCollatzSupera x =
  head [n | n <- [1..], any (> x) (collatz n)]

-- 2ª definición
menorCollatzSupera2 :: Integer -> Integer
menorCollatzSupera2 x =
  head [y | y <- [1..], maximum (collatz y) > x]

-- ---------------------------------------------------------------------
-- Ejercicio 7. 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..])
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir, usando la criba de Eratóstenes, la constante
--    primos :: Integral a => [a]
-- cuyo valor es la lista de los números primos. Por ejemplo,
--    take 10 primos  ==  [2,3,5,7,11,13,17,19,23,29]
-- ---------------------------------------------------------------------

primos :: Integral a => [a]
primos = criba [2..]
  where criba []     = []
        criba (n:ns) = n : criba (elimina n ns)
        elimina n xs = [x | x <- xs, x `mod` n /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Definir la función
--    primo :: Integral a => a -> Bool
-- tal que (primo n) se verifica si n es primo. Por ejemplo,
--    primo 7  ==  True
--    primo 9  ==  False
-- ---------------------------------------------------------------------

primo :: Int -> Bool
primo n = head (dropWhile (<n) primos) == n

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Definir la función
--    sumaDeDosPrimos :: Int -> [(Int,Int)]
-- tal que (sumaDeDosPrimos n) es la lista de las distintas
-- descomposiciones de n como suma de dos números primos. Por ejemplo,
--    sumaDeDosPrimos 30  ==  [(7,23),(11,19),(13,17)]
--    sumaDeDosPrimos 10  ==  [(3,7),(5,5)]
-- Calcular, usando la función sumaDeDosPrimos, el menor número que
-- puede escribirse de 10 formas distintas como suma de dos primos.
-- ---------------------------------------------------------------------

sumaDeDosPrimos :: Int -> [(Int,Int)]
sumaDeDosPrimos n =
  [(x,n-x) | x <- primosN, primo (n-x)]
  where primosN = takeWhile (<= (n `div` 2)) primos
```

```haskell
-- El cálculo es
--    ghci> head [x | x <- [1..], length (sumaDeDosPrimos x) == 10]
--    114

-- ---------------------------------------------------------------------
-- § La lista infinita de factoriales                                  --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir, por comprensión, la función
--    factoriales1 :: [Integer]
-- tal que factoriales1 es la lista de los factoriales. Por ejemplo,
--    take 10 factoriales1  ==  [1,1,2,6,24,120,720,5040,40320,362880]
-- ---------------------------------------------------------------------

factoriales1 :: [Integer]
factoriales1 = [factorial n | n <- [0..]]

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 4  ==  24
factorial :: Integer -> Integer
factorial n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Definir, usando zipWith, la función
--    factoriales2 :: [Integer]
-- tal que factoriales2 es la lista de los factoriales. Por ejemplo,
--    take 10 factoriales2  ==  [1,1,2,6,24,120,720,5040,40320,362880]
-- ---------------------------------------------------------------------

factoriales2 :: [Integer]
factoriales2 = 1 : zipWith (*) [1..] factoriales2

-- El cálculo es
--    take 4 factoriales2
--    = take 4 (1 : zipWith (*) [1..] factoriales2)
--    = 1 : take 3 (zipWith (*) [1..] factoriales2)
--    = 1 : take 3 (zipWith (*) [1..] [1|R1])          {R1 es tail factoriales2}
--    = 1 : take 3 (1 : zipWith (*) [2..] R1)
```

```
--    = 1 : 1 : take 2 (zipWith (*) [2..] [1|R2])       {R2 es drop 2 factoriales
--    = 1 : 1 : take 2 (2 : zipWith (*) [3..] R2)
--    = 1 : 1 : 2 : take 1 (zipWith (*) [3..] [2|R3])    {R3 es drop 3 factoriale
--    = 1 : 1 : 2 : take 1 (6 : zipWith (*) [4..] R3)
--    = 1 : 1 : 2 : 6 : take 0 (zipWith (*) [4..] R3)
--    = 1 : 1 : 2 : 6 : []
--    = [1, 1, 2, 6]

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 3000 factoriales1 in (sum xs - sum xs)
--    let xs = take 3000 factoriales2 in (sum xs - sum xs)
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> let xs = take 3000 factoriales1 in (sum xs - sum xs)
--    0
--    (17.51 secs, 5631214332 bytes)
--    ghci> let xs = take 3000 factoriales2 in (sum xs - sum xs)
--    0
--    (0.04 secs, 17382284 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 9.4. Definir, por recursión, la función
--    factoriales3 :: [Integer]
-- tal que factoriales3 es la lista de los factoriales. Por ejemplo,
--    take 10 factoriales3  ==  [1,1,2,6,24,120,720,5040,40320,362880]
-- ---------------------------------------------------------------------

factoriales3 :: [Integer]
factoriales3 = 1 : aux 1 [1..]
  where aux x (y:ys) = z : aux z ys
          where z = x*y

-- El cálculo es
--    take 4 factoriales3
--    = take 4 (1 : aux 1 [1..])
--    = 1 : take 3 (aux 1 [1..])
--    = 1 : take 3 (1 : aux 1 [2..])
```

```
--    = 1 : 1 : take 2 (aux 1 [2..])
--    = 1 : 1 : take 2 (2 : aux 2 [3..])
--    = 1 : 1 : 2 : take 1 (aux 2 [3..])
--    = 1 : 1 : 2 : take 1 (6 : aux 6 [4..])
--    = 1 : 1 : 2 : 6 : take 0 (aux 6 [4..])
--    = 1 : 1 : 2 : 6 : []
--    = [1,1,2,6]

-- ---------------------------------------------------------------------
-- Ejercicio 9.5. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 3000 factoriales2 in (sum xs - sum xs)
--    let xs = take 3000 factoriales3 in (sum xs - sum xs)
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> let xs = take 3000 factoriales2 in (sum xs - sum xs)
--    0
--    (0.04 secs, 17382284 bytes)
--    ghci> let xs = take 3000 factoriales3 in (sum xs - sum xs)
--    0
--    (0.04 secs, 18110224 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 9.6. Definir, usando scanl1, la función
--    factoriales4 :: [Integer]
-- tal que factoriales4 es la lista de los factoriales. Por ejemplo,
--    take 10 factoriales4  ==  [1,1,2,6,24,120,720,5040,40320,362880]
-- ---------------------------------------------------------------------

factoriales4 :: [Integer]
factoriales4 = 1 : scanl1 (*) [1..]

-- ---------------------------------------------------------------------
-- Ejercicio 9.7. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 3000 factoriales3 in (sum xs - sum xs)
--    let xs = take 3000 factoriales4 in (sum xs - sum xs)
-- ---------------------------------------------------------------------
```

```
-- El cálculo es
--    ghci> let xs = take 3000 factoriales3 in (sum xs - sum xs)
--    0
--    (0.04 secs, 18110224 bytes)
--    ghci> let xs = take 3000 factoriales4 in (sum xs - sum xs)
--    0
--    (0.03 secs, 11965328 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 9.8. Definir, usando iterate, la función
--    factoriales5 :: [Integer]
-- tal que factoriales5 es la lista de los factoriales. Por ejemplo,
--    take 10 factoriales5  ==  [1,1,2,6,24,120,720,5040,40320,362880]
-- ---------------------------------------------------------------------

factoriales5 :: [Integer]
factoriales5 = map snd (iterate f (1,1))
  where f (x,y) = (x+1,x*y)

-- El cálculo es
--    take 4 factoriales5
--    = take 4 (map snd aux)
--    = take 4 (map snd (iterate f (1,1)))
--    = take 4 (map snd [(1,1),(2,1),(3,2),(4,6),...])
--    = take 4 [1,1,2,6,...]
--    = [1,1,2,6]

-- ---------------------------------------------------------------------
-- Ejercicio 9.9. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 3000 factoriales4 in (sum xs - sum xs)
--    let xs = take 3000 factoriales5 in (sum xs - sum xs)
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> let xs = take 3000 factoriales4 in (sum xs - sum xs)
--    0
--    (0.04 secs, 18110224 bytes)
--    ghci> let xs = take 3000 factoriales5 in (sum xs - sum xs)
--    0
```

```
--    (0.03 secs, 11965760 bytes)

-- ---------------------------------------------------------------------
-- § La sucesión de Fibonacci                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. La sucesión de Fibonacci está definida por
--    f(0) = 0
--    f(1) = 1
--    f(n) = f(n-1)+f(n-2), si n > 1.
--
-- Definir la función
--    fib :: Integer -> Integer
-- tal que (fib n) es el n-ésimo término de la sucesión de Fibonacci.
-- Por ejemplo,
--    fib 8  ==  21
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Definir, por comprensión, la función
--    fibs1 :: [Integer]
-- tal que fibs1 es la sucesión de Fibonacci. Por ejemplo,
--    take 10 fibs1  ==  [0,1,1,2,3,5,8,13,21,34]
-- ---------------------------------------------------------------------

fibs1 :: [Integer]
fibs1 = [fib n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 10.3. Definir, por recursión, la función
--    fibs2 :: [Integer]
-- tal que fibs2 es la sucesión de Fibonacci. Por ejemplo,
--    take 10 fibs2  ==  [0,1,1,2,3,5,8,13,21,34]
-- ---------------------------------------------------------------------
```

```haskell
fibs2 :: [Integer]
fibs2 = aux 0 1
  where aux x y = x : aux y (x+y)

-- ---------------------------------------------------------------------
-- Ejercicio 10.4. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 30 fibs1 in (sum xs - sum xs)
--    let xs = take 30 fibs2 in (sum xs - sum xs)
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> let xs = take 30 fibs1 in (sum xs - sum xs)
--    0
--    (6.02 secs, 421589672 bytes)
--    ghci> let xs = take 30 fibs2 in (sum xs - sum xs)
--    0
--    (0.01 secs, 515856 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 10.5. Definir, por recursión con zipWith, la función
--    fibs3 :: [Integer]
-- tal que fibs3 es la sucesión de Fibonacci. Por ejemplo,
--    take 10 fibs3  ==  [0,1,1,2,3,5,8,13,21,34]
-- ---------------------------------------------------------------------

fibs3 :: [Integer]
fibs3 = 0 : 1: zipWith (+) fibs3 (tail fibs3)

-- ---------------------------------------------------------------------
-- Ejercicio 10.6. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 40000 fibs2 in (sum xs - sum xs)
--    let xs = take 40000 fibs3 in (sum xs - sum xs)
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> let xs = take 40000 fibs2 in (sum xs - sum xs)
--    0
```

```
--    (0.90 secs, 221634544 bytes)
--    ghci> let xs = take 40000 fibs3 in (sum xs - sum xs)
--    0
--    (1.14 secs, 219448176 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 10.7. Definir, por recursión con acumuladores, la función
--    fibs4 :: [Integer]
-- tal que fibs4 es la sucesión de Fibonacci. Por ejemplo,
--    take 10 fibs4  ==  [0,1,1,2,3,5,8,13,21,34]
-- ---------------------------------------------------------------------

fibs4 :: [Integer]
fibs4 = fs
  where (xs,ys,fs) = (zipWith (+) ys fs, 1:xs, 0:ys)

-- El cálculo de fibs4 es
--   +-------------------------+-----------------+--------------------+
--   | xs = zipWith (+) ys fs | ys = 1:xs       | fs = 0:ys          |
--   +-------------------------+-----------------+--------------------+
--   |                         | 1:...           | 0:...              |
--   |                         | ^               | ^                  |
--   | 1:...                   | 1:1:...         | 0:1:1:...          |
--   |                         |   ^             |   ^                |
--   | 1:2:...                 | 1:1:2:...       | 0:1:1:2:...        |
--   |                         |     ^           |     ^              |
--   | 1:2:3:...               | 1:1:2:3:...     | 0:1:1:2:3:...      |
--   |                         |       ^         |       ^            |
--   | 1:2:3:5:...             | 1:1:2:3:5:...   | 0:1:1:2:3:5:...    |
--   |                         |         ^       |         ^          |
--   | 1:2:3:5:8:...           | 1:1:2:3:5:8:... | 0:1:1:2:3:5:8:...  |
--   +-------------------------+-----------------+--------------------+
-- En la tercera columna se va construyendo la sucesión.

-- ---------------------------------------------------------------------
-- Ejercicio 10.8. Comparar el tiempo y espacio necesarios para calcular
-- las siguientes expresiones
--    let xs = take 40000 fibs3 in (sum xs - sum xs)
--    let xs = take 40000 fibs4 in (sum xs - sum xs)
-- ---------------------------------------------------------------------
```

```haskell
-- El cálculo es
--    ghci> let xs = take 40000 fibs2 in (sum xs - sum xs)
--    0
--    (0.90 secs, 221634544 bytes)
--    ghci> let xs = take 40000 fibs4 in (sum xs - sum xs)
--    0
--    (0.84 secs, 219587064 bytes)

-- ---------------------------------------------------------------------
-- § El triángulo de Pascal                                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. 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.
--
-- Definir, con iterate, la función
--    pascal1 :: [[Integer]]
-- tal que pascal es la lista de las líneas del triángulo de Pascal. Por
-- ejemplo,
--    ghci> take 6 pascal1
--    [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
-- ---------------------------------------------------------------------

pascal1 :: [[Integer]]
pascal1 = iterate f [1]
  where f xs = zipWith (+) (0:xs) (xs++[0])
```

```haskell
-- Por ejemplo,
--    xs      = [1,2,1]
--    0:xs    = [0,1,2,1]
--    xs++[0] = [1,2,1,0]
--    +       = [1,3,3,1]

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. Definir por recursión la función
--    pascal2 :: [[Integer]]
-- tal que pascal es la lista de las líneas del triángulo de Pascal. Por
-- ejemplo,
--    ghci> take 6 pascal2
--    [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
-- ---------------------------------------------------------------------

pascal2 :: [[Integer]]
pascal2 = [1] : map f pascal2
  where f xs = zipWith (+) (0:xs) (xs++[0])

-- ---------------------------------------------------------------------
-- Ejercicio 11.3. Escribir la traza del cálculo de la expresión
--    take 4 pascal2
-- ---------------------------------------------------------------------

-- Nota: El cálculo es
--  take 4 pascal2
--  = take 4 ([1] : map f pascal2)
--  = [1] : (take 3 (map f pascal2))
--  = [1] : (take 3 (map f ([1]:R1)))
--  = [1] : (take 3 ((f [1]) : map f R1)))
--  = [1] : (take 3 ((zipWith (+) (0:[1]) ([1]++[0]) : map f R1)))
--  = [1] : (take 3 ((zipWith (+) [0,1] [1,0]) : map f R1)))
--  = [1] : (take 3 ([1,1] : map f R1)))
--  = [1] : [1,1] : (take 2 (map f R1)))
--  = [1] : [1,1] : (take 2 (map f ([1,1]:R2)))
--  = [1] : [1,1] : (take 2 ((f [1,1]) : map f R2)))
--  = [1] : [1,1] : (take 2 ((zipWith (+) (0:[1,1]) ([1,1]++[0])) : map f R2))
--  = [1] : [1,1] : (take 2 ((zipWith (+) [0,1,1] [1,1,0]) : map f R2))
--  = [1] : [1,1] : (take 2 ([1,2,1] : map f R2))
--  = [1] : [1,1] : [1,2,1] : (take 1 (map f R2))
```

```
--  = [1] : [1,1] : [1,2,1] : (take 1 (map f ([1,2,1]:R3)))
--  = [1] : [1,1] : [1,2,1] : (take 1 ((f [1,2,1]) : map f R3))
--  = [1] : [1,1] : [1,2,1] : (take 1 ((zipWith (+) (0:[1,2,1]) ([1,2,1]++[0]))
--                                     : map f R3))
--  = [1] : [1,1] : [1,2,1] : (take 1 ((zipWith (+) [0,1,2,1] [1,2,1,0])
--                                     : map f R3)))
--  = [1] : [1,1] : [1,2,1] : (take 1 ([1,3,3,1] : map f R3)))
--  = [1] : [1,1] : [1,2,1] : [1,3,3,1] : (take 0 (map f R3)))
--  = [1] : [1,1] : [1,2,1] : [1,3,3,1] : []
--  = [[1],[1,1],[1,2,1],[1,3,3,1]]
-- en el cálculo con R1, R2pascal y R3 es el triángulo de
-- Pascal sin el primero, los dos primeros o los tres primeros elementos,
-- respectivamente.
```

# Relación 12

# Aplicaciones de la programación funcional con listas infinitas

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se estudia distintas aplicaciones de la programación
-- funcional que usan listas infinitas
-- + enumeración de los números enteros,
-- + el problema de la bicicleta de Turing y
-- + la sucesión de Golomb,

-- ---------------------------------------------------------------------
-- § Enumeración de los números enteros                                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Los números enteros se pueden ordenar como sigue
--    0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5, -6, 6, -7, 7, ...
-- Definir, por comprensión, la constante
--    enteros :: [Int]
-- tal que enteros es la lista de los enteros con la ordenación
-- anterior. Por ejemplo,
--    take 10 enteros  ==  [0,-1,1,-2,2,-3,3,-4,4,-5]
-- ---------------------------------------------------------------------

-- 1ª definición
enteros :: [Int]
```

```haskell
enteros = 0 : concat [[-x,x] | x <- [1..]]

-- 2ª definición
enteros2 :: [Int]
enteros2 = iterate siguiente 0
  where siguiente x | x >= 0    = -x-1
                    | otherwise = -x

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    posicion :: Int -> Int
-- tal que (posicion x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
--    posicion 2  ==  4
-- ---------------------------------------------------------------------

-- 1ª definición
posicion :: Int -> Int
posicion x = length (takeWhile (/=x) enteros)

-- 2ª definición
posicion2 :: Int -> Int
posicion2 x = aux enteros 0
    where aux (y:ys) n | x == y    = n
                       | otherwise = aux ys (n+1)
          aux _ _ = error "Imposible"

-- 3ª definición
posicion3 :: Int -> Int
posicion3 x = head [n | (n,y) <- zip [0..] enteros, y == x]

-- 4ª definición
posicion4 :: Int -> Int
posicion4 x | x >= 0    = 2*x
            | otherwise = 2*(-x)-1

-- ---------------------------------------------------------------------
-- § El problema de la bicicleta de Turing                            --
-- ---------------------------------------------------------------------
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Cuentan que Alan Turing tenía una bicicleta vieja,
-- que tenía una cadena con un eslabón débil y además uno de los radios
-- de la rueda estaba doblado. Cuando el radio doblado coincidía con el
-- eslabón débil, entonces la cadena se rompía.
--
-- La bicicleta se identifica por los parámetros (i,d,n) donde
-- - i es el número del eslabón que coincide con el radio doblado al
--   empezar a andar,
-- - d es el número de eslabones que se desplaza la cadena en cada
--   vuelta de la rueda y
-- - n es el número de eslabones de la cadena (el número n es el débil).
-- Si i=2 y d=7 y n=25, entonces la lista con el número de eslabón que
-- toca el radio doblado en cada vuelta es
--    [2,9,16,23,5,12,19,1,8,15,22,4,11,18,0,7,14,21,3,10,17,24,6,...
-- Con lo que la cadena se rompe en la vuelta número 14.
--
-- Definir la función
--    eslabones :: Int -> Int -> Int -> [Int]
-- tal que (eslabones i d n) es la lista con los números de eslabones
-- que tocan el radio doblado en cada vuelta en una bicicleta de tipo
-- (i,d,n). Por ejemplo,
--    take 10 (eslabones 2 7 25)  ==  [2,9,16,23,5,12,19,1,8,15]
-- ---------------------------------------------------------------------

eslabones :: Int -> Int -> Int -> [Int]
eslabones i d n = [(i+d*j) `mod` n | j <- [0..]]

-- 2ª definición (con iterate):
eslabones2 :: Int -> Int -> Int -> [Int]
eslabones2 i d n = map (`mod` n) (iterate (+d) i)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    numeroVueltas :: Int -> Int -> Int -> Int
-- tal que (numeroVueltas i d n) es el número de vueltas que pasarán
-- hasta que la cadena se rompa en una bicicleta de tipo (i,d,n). Por
-- ejemplo,
--    numeroVueltas 2 7 25  ==  14
-- ---------------------------------------------------------------------
```

```haskell
numeroVueltas :: Int -> Int -> Int -> Int
numeroVueltas i d n = length (takeWhile (/=0) (eslabones i d n))

-- ---------------------------------------------------------------------
-- § La sucesión de Golomb                                             --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. [Basado en el problema 341 del proyecto Euler]. La
-- sucesión de Golomb {G(n)} es una sucesión auto descriptiva: es la
-- única sucesión no decreciente de números naturales tal que el número
-- n aparece G(n) veces en la sucesión. Los valores de G(n) para los
-- primeros números son los siguientes:
--    n       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
--    G(n)    1 2 2 3 3 4 4 4 5  5  5  6  6  6  6 ...
-- En los apartados de este ejercicio se definirá una función para
-- calcular los términos de la sucesión de Golomb.
--
-- Definir la función
--    golomb :: Int -> Int
-- tal que (golomb n) es el n-ésimo término de la sucesión de Golomb.
-- Por ejemplo,
--    golomb 5  ==  3
--    golomb 9  ==  5
-- Indicación: Se puede usar la función sucGolomb del apartado 2.
-- ---------------------------------------------------------------------

golomb :: Int -> Int
golomb 1 = 1
golomb 2 = 2
golomb n = sucGolomb !! (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    sucGolomb :: [Int]
-- tal que sucGolomb es la lista de los términos de la sucesión de
-- Golomb. Por ejemplo,
--    take 15 sucGolomb  ==  [1,2,2,3,3,4,4,4,5,5,5,6,6,6,6]
-- Indicación: Se puede usar la función subSucGolomb del apartado 3.
```

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

sucGolomb :: [Int]
sucGolomb = subSucGolomb 1

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    subSucGolomb :: Int -> [Int]
-- tal que (subSucGolomb x) es la lista de los términos de la sucesión
-- de Golomb a partir de la primera ocurrencia de x. Por ejemplo,
--    take 10 (subSucGolomb 4)  ==  [4,4,4,5,5,5,6,6,6,6]
-- Indicación: Se puede usar la función golomb del apartado 1.
-- ---------------------------------------------------------------------

subSucGolomb :: Int -> [Int]
subSucGolomb x = replicate (golomb x) x ++ subSucGolomb (x+1)
```

# Relación 13

# El juego del nim y las funciones de entrada/salida

```
-- ---------------------------------------------------------------------
-- § Introducción                                                      --
-- ---------------------------------------------------------------------

-- En el juego del nim el tablero tiene 5 filas numeradas de estrellas,
-- cuyo contenido inicial es el siguiente
--    1: *****
--    2: ****
--    3: ***
--    4: **
--    5: *
-- Dos jugadores retiran por turno una o más estrellas de una fila. El
-- ganador es el jugador que retire la última estrella. En este
-- ejercicio se va implementar el juego del Nim para practicar con las
-- funciones de entrada y salida estudiadas en el tema 13 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-13.html
--
-- Nota: El juego debe de ejecutarse en una consola, no en la shell de
-- emacs.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Data.Char
```

```haskell
-- ---------------------------------------------------------------------
-- § Representación                                                    --
-- ---------------------------------------------------------------------

-- El tablero se representará como una lista de números indicando el
-- número de estrellas de cada fila. Con esta representación, el tablero
-- inicial es [5,4,3,2,1].

-- Representación del tablero.
type Tablero = [Int]

-- inicial es el tablero al principio del juego.
inicial ::  Tablero
inicial = [5,4,3,2,1]

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    finalizado :: Tablero -> Bool
-- tal que (finalizado t) se verifica si t es el tablero de un juego
-- finalizado; es decir, sin estrellas. Por ejemplo,
--    finalizado [0,0,0,0,0]  ==  True
--    finalizado [1,3,0,0,1]  ==  False
-- ---------------------------------------------------------------------

finalizado :: Tablero -> Bool
finalizado = all (== 0)

-- ---------------------------------------------------------------------
-- Ejecicio 2.2. Definir la función
--    valida :: Tablero -> Int -> Int -> Bool
-- tal que (valida t f n) se verifica si se puede coger n estrellas en
-- la fila f del tablero t y n es mayor o igual que 1. Por ejemplo,
--    valida [4,3,2,1,0] 2 3  ==  True
--    valida [4,3,2,1,0] 2 4  ==  False
--    valida [4,3,2,1,0] 2 2  ==  True
--    valida [4,3,2,1,0] 2 0  ==  False
-- ---------------------------------------------------------------------

valida :: Tablero -> Int -> Int -> Bool
```

```haskell
valida t f n = n >= 1 && t !! (f-1) >= n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    jugada :: Tablero -> Int -> Int -> Tablero
-- tal que (jugada t f n) es el tablero obtenido a partir de t
-- eliminando n estrellas de la fila f. Por ejemplo,
--    jugada [4,3,2,1,0] 2 1  ==  [4,2,2,1,0]
-- ---------------------------------------------------------------------

jugada :: Tablero -> Int -> Int -> Tablero
jugada t f n = [if x == f then y-n else y | (x,y) <- zip [1..] t]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la acción
--    nuevaLinea :: IO ()
-- que consiste en escribir una nueva línea. Por ejemplo,
--    ghci> nuevaLinea
--
--    ghci>
-- ---------------------------------------------------------------------

nuevaLinea :: IO ()
nuevaLinea = putChar '\n'

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    estrellas :: Int -> String
-- tal que (estrellas n) es la cadena formada con n estrellas. Por
-- ejemplo,
--    ghci> estrellas 3
--    "* * * "
-- ---------------------------------------------------------------------

estrellas :: Int -> String
estrellas n = concat (replicate n "* ")

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la acción
--    escribeFila :: Int -> Int -> IO ()
```

```haskell
-- tal que (escribeFila f n) escribe en la fila f n estrellas. Por
-- ejemplo,
--    ghci> escribeFila 2 3
--    2: * * *
-- ---------------------------------------------------------------------

escribeFila :: Int -> Int -> IO ()
escribeFila f n = putStrLn (show f ++ ": " ++ estrellas n)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la acción
--    escribeTablero :: Tablero -> IO ()
-- tal que (escribeTablero t) escribe el tablero t. Por
-- ejemplo,
--    ghci> escribeTablero [3,4,1,0,1]
--    1: * * *
--    2: * * * *
--    3: *
--    4:
--    5: *
-- ---------------------------------------------------------------------

escribeTablero :: Tablero -> IO ()
escribeTablero t =
  sequence_ [escribeFila x y | (x,y) <- zip [1..] t]

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la acción
--    leeDigito :: String -> IO Int
-- tal que (leeDigito c) escribe una nueva línea con la cadena "prueba",
-- lee un carácter y comprueba que es un dígito. Además, si el carácter
-- leido es un dígito entonces devuelve el entero correspondiente y si
-- no lo es entonces escribe el mensaje "Entrada incorrecta" y vuelve a
-- leer otro carácter. Por ejemplo,
--    ghci> leeDigito "prueba "
--    prueba 3
--    3
--    ghci> leeDigito "prueba "
--    prueba c
--    ERROR: Entrada incorrecta
```

```
--    prueba 3
--    3
-- ---------------------------------------------------------------------

leeDigito :: String -> IO Int
leeDigito c = do
  putStr c
  x <- getChar
  nuevaLinea
  if isDigit x
    then return (digitToInt x)
    else do putStrLn "ERROR: Entrada incorrecta"
            leeDigito c

-- ---------------------------------------------------------------------
-- Ejercicio 9. Los jugadores se representan por los números 1 y 2.
-- Definir la función
--    siguiente :: Int -> Int
-- tal que (siguiente j) es el jugador siguiente de j.
-- ---------------------------------------------------------------------

siguiente :: Int -> Int
siguiente 1 = 2
siguiente 2 = 1
siguiente _ = error "Imposible"

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la acción
--    juego :: Tablero -> Int -> IO ()
-- tal que (juego t j) es el juego a partir del tablero t y el turno del
-- jugador j. Por ejemplo,
--    ghci> juego [0,1,0,1,0] 2
--
--    1:
--    2: *
--    3:
--    4: *
--    5:
--
--    J 2
```

```
--    Elige una fila: 2
--    Elige cuantas estrellas retiras: 1
--
--    1:
--    2:
--    3:
--    4: *
--    5:
--
--    J 1
--    Elige una fila: 4
--    Elige cuantas estrellas retiras: 1
--
--    1:
--    2:
--    3:
--    4:
--    5:
--
--    J 1 He ganado
-- ----------------------------------------------------------------------

juego :: Tablero -> Int -> IO ()
juego t j = do
  nuevaLinea
  escribeTablero t
  if finalizado t
   then do nuevaLinea
           putStr "J "
           putStr (show (siguiente j))
           putStrLn " He ganado"
   else do nuevaLinea
           putStr "J "
           print j
           f <- leeDigito "Elige una fila: "
           n <- leeDigito "Elige cuantas estrellas retiras: "
           if valida t f n
            then juego (jugada t f n) (siguiente j)
            else do nuevaLinea
                    putStrLn "ERROR: jugada incorrecta"
```

juego t j

```
-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la acción
--    nim :: IO ()
-- consistente en una partida del nim. Por ejemplo (en una consola no en
-- la shell de emacs),
--    ghci> nim
--
--    1: * * * * *
--    2: * * * *
--    3: * * *
--    4: * *
--    5: *
--
--    J 1
--    Elige una fila: 1
--    Elige cuantas estrellas retiras: 4
--
--    1: *
--    2: * * * *
--    3: * * *
--    4: * *
--    5: *
--
--    J 2
--    Elige una fila: 3
--    Elige cuantas estrellas retiras: 3
--
--    1: *
--    2: * * * *
--    3:
--    4: * *
--    5: *
--
--    J 1
--    Elige una fila: 2
--    Elige cuantas estrellas retiras: 4
--
--    1: *
```

```
--    2:
--    3:
--    4: * *
--    5: *
--
--    J 2
--    Elige una fila: 4
--    Elige cuantas estrellas retiras: 1
--
--    1: *
--    2:
--    3:
--    4: *
--    5: *
--
--    J 1
--    Elige una fila: 1
--    Elige cuantas estrellas retiras: 1
--
--    1:
--    2:
--    3:
--    4: *
--    5: *
--
--    J 2
--    Elige una fila: 4
--    Elige cuantas estrellas retiras: 1
--
--    1:
--    2:
--    3:
--    4:
--    5: *
--
--    J 1
--    Elige una fila: 5
--    Elige cuantas estrellas retiras: 1
--
--    1:
```

```
--    2:
--    3:
--    4:
--    5:
--
--    J 1 He ganado
-- ------------------------------------------------------------------------

nim :: IO ()
nim = juego inicial 1
```

# Relación 14

# Vectores y matrices

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es hacer ejercicios sobre vectores y
-- matrices con el tipo de las tablas, definido en el módulo
-- Data.Array y explicado en el tema 18 que se encuentra en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-18.html

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

import Data.Array

-- ---------------------------------------------------------------------
-- Tipos de los vectores y de las matrices                             --
-- ---------------------------------------------------------------------

-- Los vectores son tablas cuyos índices son números naturales.
type Vector a = Array Int a

-- Las matrices son tablas cuyos índices son pares de números
-- naturales.
type Matriz a = Array (Int,Int) a

-- ---------------------------------------------------------------------
-- Operaciones básicas con matrices                                    --
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    listaVector :: Num a => [a] -> Vector a
-- tal que (listaVector xs) es el vector correspondiente a la lista
-- xs. Por ejemplo,
--    ghci> listaVector [3,2,5]
--    array (1,3) [(1,3),(2,2),(3,5)]
-- ---------------------------------------------------------------------

listaVector :: Num a => [a] -> Vector a
listaVector xs = listArray (1,n) xs
  where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    listaMatriz :: Num a => [[a]] -> Matriz a
-- tal que (listaMatriz xss) es la matriz cuyas filas son los elementos
-- de xss. Por ejemplo,
--    ghci> listaMatriz [[1,3,5],[2,4,7]]
--    array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),5),
--                         ((2,1),2),((2,2),4),((2,3),7)]
-- ---------------------------------------------------------------------

listaMatriz :: Num a => [[a]] -> Matriz a
listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
  where m = length xss
        n = length (head xss)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    numFilas :: Num a => Matriz a -> Int
-- tal que (numFilas m) es el número de filas de la matriz m. Por
-- ejemplo,
--    numFilas (listaMatriz [[1,3,5],[2,4,7]])  ==  2
-- ---------------------------------------------------------------------

numFilas :: Num a => Matriz a -> Int
numFilas = fst . snd . bounds
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    numColumnas :: Num a => Matriz a -> Int
-- tal que (numColumnas m) es el número de columnas de la matriz
-- m. Por ejemplo,
--    numColumnas (listaMatriz [[1,3,5],[2,4,7]])  ==  3
-- ---------------------------------------------------------------------

numColumnas :: Num a => Matriz a -> Int
numColumnas = snd . snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    dimension :: Num a => Matriz a -> (Int,Int)
-- tal que (dimension m) es la dimensión de la matriz m. Por ejemplo,
--    dimension (listaMatriz [[1,3,5],[2,4,7]])  ==  (2,3)
-- ---------------------------------------------------------------------

dimension :: Num a => Matriz a -> (Int,Int)
dimension = snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    separa :: Int -> [a] -> [[a]]
-- tal que (separa n xs) es la lista obtenida separando los elementos de
-- xs en grupos de n elementos (salvo el último que puede tener menos de
-- n elementos). Por ejemplo,
--    separa 3 [1..11]  ==  [[1,2,3],[4,5,6],[7,8,9],[10,11]]
-- ---------------------------------------------------------------------

separa :: Int -> [a] -> [[a]]
separa _ [] = []
separa n xs = take n xs : separa n (drop n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    matrizLista :: Num a => Matriz a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
```

```
--    ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> m
--    array ((1,1),(2,3)) [((1,1),5),((1,2),1),((1,3),0),
--                         ((2,1),3),((2,2),2),((2,3),6)]
--    ghci> matrizLista m
--    [[5,1,0],[3,2,6]]
-- ---------------------------------------------------------------------

matrizLista :: Num a => Matriz a -> [[a]]
matrizLista p = separa (numColumnas p) (elems p)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    vectorLista :: Num a => Vector a -> [a]
-- tal que (vectorLista x) es la lista de los elementos del vector
-- v. Por ejemplo,
--    ghci> let v = listaVector [3,2,5]
--    ghci> v
--    array (1,3) [(1,3),(2,2),(3,5)]
--    ghci> vectorLista v
--    [3,2,5]
-- ---------------------------------------------------------------------

vectorLista :: Num a => Vector a -> [a]
vectorLista = elems

-- ---------------------------------------------------------------------
-- Suma de matrices                                                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    sumaMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
-- tal que (sumaMatrices x y) es la suma de las matrices x e y. Por
-- ejemplo,
--    ghci> let m1 = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> let m2 = listaMatriz [[4,6,3],[1,5,2]]
--    ghci> matrizLista (sumaMatrices m1 m2)
--    [[9,7,3],[4,7,8]]
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
sumaMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
sumaMatrices p q =
  array ((1,1),(m,n)) [((i,j),p!(i,j)+q!(i,j))
                       | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p

-- 2ª definición
sumaMatrices2 :: Num a => Matriz a -> Matriz a -> Matriz a
sumaMatrices2 p q =
  listArray (bounds p) (zipWith (+) (elems p) (elems q))

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    filaMat :: Num a => Int -> Matriz a -> Vector a
-- tal que (filaMat i p) es el vector correspondiente a la fila i-ésima
-- de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
--    ghci> filaMat 2 p
--    array (1,3) [(1,3),(2,2),(3,6)]
--    ghci> vectorLista (filaMat 2 p)
--    [3,2,6]
-- ---------------------------------------------------------------------

filaMat :: Num a => Int -> Matriz a -> Vector a
filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
  where n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    columnaMat :: Num a => Int -> Matriz a -> Vector a
-- tal que (columnaMat j p) es el vector correspondiente a la columna
-- j-ésima de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
--    ghci> columnaMat 2 p
--    array (1,3) [(1,1),(2,2),(3,5)]
--    ghci> vectorLista (columnaMat 2 p)
--    [1,2,5]
-- ---------------------------------------------------------------------
```

```haskell
columnaMat :: Num a => Int -> Matriz a -> Vector a
columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]
  where m = numFilas p

-- ---------------------------------------------------------------------
-- Producto de matrices                                                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    prodEscalar :: Num a => Vector a -> Vector a -> a
-- tal que (prodEscalar v1 v2) es el producto escalar de los vectores v1
-- y v2. Por ejemplo,
--    ghci> let v = listaVector [3,1,10]
--    ghci> prodEscalar v v
--    110
-- ---------------------------------------------------------------------

-- 1ª solución
prodEscalar :: Num a => Vector a -> Vector a -> a
prodEscalar v1 v2 =
  sum [i*j | (i,j) <- zip (elems v1) (elems v2)]

-- 2ª solución
prodEscalar2 :: Num a => Vector a -> Vector a -> a
prodEscalar2 v1 v2 =
  sum (zipWith (*) (elems v1) (elems v2))

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    prodMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
-- tal que (prodMatrices p q) es el producto de las matrices p y q. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,1],[2,4]]
--    ghci> prodMatrices p p
--    array ((1,1),(2,2)) [((1,1),11),((1,2),7),((2,1),14),((2,2),18)]
--    ghci> matrizLista (prodMatrices p p)
--    [[11,7],[14,18]]
--    ghci> let q = listaMatriz [[7],[5]]
```

```
--    ghci> prodMatrices p q
--    array ((1,1),(2,1)) [((1,1),26),((2,1),34)]
--    ghci> matrizLista (prodMatrices p q)
--    [[26],[34]]
-- ---------------------------------------------------------------------

prodMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
prodMatrices p q =
  array ((1,1),(m,n))
        [((i,j), prodEscalar (filaMat i p) (columnaMat j q))
        | i <- [1..m], j <- [1..n]]
  where m = numFilas p
        n = numColumnas q

-- ---------------------------------------------------------------------
-- Matriz identidad                                                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    identidad :: Num a => Int -> Matriz a
-- tal que (identidad n) es la matriz identidad de orden n. Por ejemplo,
--    ghci> identidad 3
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                         ((2,1),0),((2,2),1),((2,3),0),
--                         ((3,1),0),((3,2),0),((3,3),1)]
-- ---------------------------------------------------------------------

identidad :: Num a => Int -> Matriz a
identidad n =
  array ((1,1),(n,n))
        [((i,j),f i j) | i <- [1..n], j <- [1..n]]
  where f i j | i == j    = 1
              | otherwise = 0

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    potencia :: Num a => Matriz a -> Int -> Matriz a
-- tal que (potencia p n) es la potencia n-ésima de la matriz cuadrada
-- p. Por ejemplo, si q es la matriz definida por
```

```
--    q1 :: Matriz Int
--    q1 = listArray ((1,1),(2,2)) [1,1,1,0]
-- entonces
--    ghci> potencia q1 2
--    array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),1),((2,2),1)]
--    ghci> potencia q1 3
--    array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),2),((2,2),1)]
--    ghci> potencia q1 4
--    array ((1,1),(2,2)) [((1,1),5),((1,2),3),((2,1),3),((2,2),2)]
-- ¿Qué relación hay entre las potencias de la matriz q y la sucesión de
-- Fibonacci?
-- ---------------------------------------------------------------------

q1 :: Matriz Int
q1 = listArray ((1,1),(2,2)) [1,1,1,0]

potencia :: Num a => Matriz a -> Int -> Matriz a
potencia p 0 = identidad n
  where (_,(n,_)) = bounds p
potencia p n = prodMatrices p (potencia p (n-1))

-- ---------------------------------------------------------------------
-- Traspuestas                                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    traspuesta :: Num a => Matriz a -> Matriz a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> traspuesta p
--    array ((1,1),(3,2)) [((1,1),5),((1,2),3),
--                         ((2,1),1),((2,2),2),
--                         ((3,1),0),((3,2),6)]
--    ghci> matrizLista (traspuesta p)
--    [[5,3],[1,2],[0,6]]
-- ---------------------------------------------------------------------

traspuesta :: Num a => Matriz a -> Matriz a
traspuesta p =
```

```
  array ((1,1),(n,m))
        [((i,j), p!(j,i)) | i <- [1..n], j <- [1..m]]
  where (m,n) = dimension p

-- ---------------------------------------------------------------------
-- Submatriz                                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Tipos de matrices                                                   --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    esCuadrada :: Num a => Matriz a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> esCuadrada p
--    False
--    ghci> let q = listaMatriz [[5,1],[3,2]]
--    ghci> esCuadrada q
--    True
-- ---------------------------------------------------------------------

esCuadrada :: Num a => Matriz a -> Bool
esCuadrada x = numFilas x == numColumnas x

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    esSimetrica :: (Num a, Eq a) => Matriz a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
--    ghci> esSimetrica p
--    True
--    ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
--    ghci> esSimetrica q
--    False
-- ---------------------------------------------------------------------
```

```haskell
esSimetrica :: (Num a, Eq a) => Matriz a -> Bool
esSimetrica x = x == traspuesta x

-- ---------------------------------------------------------------------
-- Diagonales de una matriz                                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    diagonalPral :: Num a => Matriz a -> Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalPral p
--    array (1,2) [(1,5),(2,2)]
--    ghci> vectorLista (diagonalPral p)
--    [5,2]
-- ---------------------------------------------------------------------

diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
  where n = min (numFilas p) (numColumnas p)

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    diagonalSec :: Num a => Matriz a -> Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalSec p
--    array (1,2) [(1,1),(2,3)]
--    ghci> vectorLista (diagonalSec p)
--    [1,3]
--    ghci> let q = traspuesta p
--    ghci> matrizLista q
--    [[5,3],[1,2],[0,6]]
--    ghci> vectorLista (diagonalSec q)
--    [3,1]
-- ---------------------------------------------------------------------
```

```haskell
diagonalSec :: Num a => Matriz a -> Vector a
diagonalSec p = array (1,n) [(i,p!(i,n+1-i)) | i <- [1..n]]
  where n = min (numFilas p) (numColumnas p)

-- ---------------------------------------------------------------------
-- Submatrices                                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la función
--    submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> submatriz 2 3 p
--    array ((1,1),(2,2)) [((1,1),5),((1,2),1),((2,1),4),((2,2),6)]
--    ghci> matrizLista (submatriz 2 3 p)
--    [[5,1],[4,6]]
-- ---------------------------------------------------------------------

-- 1ª solución
submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
  array ((1,1), (m-1,n-1))
        [((k,l), p ! f k l) | k <- [1..m-1], l <- [1.. n-1]]
  where (m,n) = dimension p
        f k l | k < i  && l < j  = (k,l)
              | k >= i && l < j  = (k+1,l)
              | k < i  && l >= j = (k,l+1)
              | otherwise        = (k+1,l+1)

-- 2ª solución
submatriz2 :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz2 i j p =
  listArray ((1,1),(m-1,n-1))
            [p!(k,yl) | k <- [1..m] , k /= i , l <- [1..n] , l /= j]
  where (m,n) = dimension p

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

```
-- Determinante                                                        --
-- ------------------------------------------------------------------------

-- ------------------------------------------------------------------------
-- Ejercicio 22. Definir la función
--    determinante:: Matriz Double -> Double
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> determinante (listArray ((1,1),(3,3)) [2,0,0,0,3,0,0,0,1])
--    6.0
--    ghci> determinante (listArray ((1,1),(3,3)) [1..9])
--    0.0
--    ghci> determinante (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
--    -33.0
-- ------------------------------------------------------------------------

determinante :: Matriz Double -> Double
determinante p
  | (m,n) == (1,1) = p!(1,1)
  | otherwise =
      sum [((-1)^(i+1))*(p!(i,1))*determinante (submatriz i 1 p)
          | i <- [1..m]]
  where (_,(m,n)) = bounds p
```

# Relación 15

# Método de Gauss para triangularizar matrices

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es definir el método de Gauss para
-- triangularizar matrices.

-- Además, en algunos ejemplos de usan matrices con números racionales.
-- En Haskell, el número racional x/y se representa por x%y. El TAD de
-- los números racionales está definido en el módulo Data.Ratio.

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

import Data.Array
import Data.Ratio

-- ---------------------------------------------------------------------
-- Tipos de los vectores y de las matrices                             --
-- ---------------------------------------------------------------------

-- Los vectores son tablas cuyos índices son números naturales.
type Vector a = Array Int a

-- Las matrices son tablas cuyos índices son pares de números
```

```haskell
-- naturales.
type Matriz a = Array (Int,Int) a

-- ---------------------------------------------------------------------
-- Funciones auxiliares                                                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    listaMatriz :: Num a => [[a]] -> Matriz a
-- tal que (listaMatriz xss) es la matriz cuyas filas son los elementos
-- de xss. Por ejemplo,
--    ghci> listaMatriz [[1,3,5],[2,4,7]]
--    array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),5),
--                         ((2,1),2),((2,2),4),((2,3),7)]
-- ---------------------------------------------------------------------

listaMatriz :: Num a => [[a]] -> Matriz a
listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
  where m = length xss
        n = length (head xss)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    separa :: Int -> [a] -> [[a]]
-- tal que (separa n xs) es la lista obtenida separando los elementos de
-- xs en grupos de n elementos (salvo el último que puede tener menos de
-- n elementos). Por ejemplo,
--    separa 3 [1..11]  ==  [[1,2,3],[4,5,6],[7,8,9],[10,11]]
-- ---------------------------------------------------------------------

separa :: Int -> [a] -> [[a]]
separa _ [] = []
separa n xs = take n xs : separa n (drop n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    matrizLista :: Num a => Matriz a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
```

```
--    ghci> m = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> m
--    array ((1,1),(2,3)) [((1,1),5),((1,2),1),((1,3),0),
--                         ((2,1),3),((2,2),2),((2,3),6)]
--    ghci> matrizLista m
--    [[5,1,0],[3,2,6]]
-- ---------------------------------------------------------------------

matrizLista :: Num a => Matriz a -> [[a]]
matrizLista p = separa (numColumnas p) (elems p)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    numFilas :: Num a => Matriz a -> Int
-- tal que (numFilas m) es el número de filas de la matriz m. Por
-- ejemplo,
--    numFilas (listaMatriz [[1,3,5],[2,4,7]])  ==  2
-- ---------------------------------------------------------------------

numFilas :: Num a => Matriz a -> Int
numFilas = fst . snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    numColumnas :: Num a => Matriz a -> Int
-- tal que (numColumnas m) es el número de columnas de la matriz
-- m. Por ejemplo,
--    numColumnas (listaMatriz [[1,3,5],[2,4,7]])  ==  3
-- ---------------------------------------------------------------------

numColumnas :: Num a => Matriz a -> Int
numColumnas = snd . snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    dimension :: Num a => Matriz a -> (Int,Int)
-- tal que (dimension m) es la dimensión de la matriz m. Por ejemplo,
--    dimension (listaMatriz [[1,3,5],[2,4,7]])  ==  (2,3)
-- ---------------------------------------------------------------------
```

```haskell
dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas p)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    diagonalPral :: Num a => Matriz a -> Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalPral p
--    array (1,2) [(1,5),(2,2)]
--    ghci> elems (diagonalPral p)
--    [5,2]
-- ---------------------------------------------------------------------

diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
  where n = min (numFilas p) (numColumnas p)

-- ---------------------------------------------------------------------
-- Transformaciones elementales                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> intercambiaFilas 1 3 p
--    array ((1,1),(3,3)) [((1,1),4),((1,2),6),((1,3),9),
--                         ((2,1),3),((2,2),2),((2,3),6),
--                         ((3,1),5),((3,2),1),((3,3),0)]
--    ghci> matrizLista (intercambiaFilas 1 3 p)
--    [[4,6,9],[3,2,6],[5,1,0]]
-- ---------------------------------------------------------------------

intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaFilas k l p =
  array ((1,1), (m,n))
```

```
        [((i,j), p! f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = (l,j)
              | i == l    = (k,j)
              | otherwise = (i,j)

-- 2ª solución
intercambiaFilas2 :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaFilas2 k l p =
  p // ([((l,i),p!(k,i)) | i <- [1..n]] ++
        [((k,i),p!(l,i)) | i <- [1..n]])
  where n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaColumnas k l p) es la matriz obtenida
-- intercambiando las columnas k y l de la matriz p. Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (intercambiaColumnas 1 3 p)
--    [[0,1,5],[6,2,3],[9,6,4]]
-- ---------------------------------------------------------------------

intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaColumnas k l p =
  array ((1,1), (m,n))
        [((i,j), p ! f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | j == k    = (i,l)
              | j == l    = (i,k)
              | otherwise = (i,j)

-- 2ª solución
intercambiaColumnas2 :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaColumnas2 k l p =
  p // ([((i,l),p!(i,k)) | i <- [1..m]] ++
        [((i,k),p!(i,l)) | i <- [1..m]])
  where m = numFilas p

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

```haskell
-- Ejercicio 10. Definir la función
--    multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
-- tal que (multFilaPor k x p) es a matriz obtenida multiplicando la
-- fila k de la matriz p por el número x. Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (multFilaPor 2 3 p)
--    [[5,1,0],[9,6,18],[4,6,9]]
-- ---------------------------------------------------------------------

multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
multFilaPor k x p =
  array ((1,1), (m,n))
        [((i,j), f i j)  | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = x*(p!(i,j))
              | otherwise = p!(i,j)

-- 2ª solución
multFilaPor2 :: Num a => Int -> a -> Matriz a -> Matriz a
multFilaPor2 k x p =
  p // [((k,i),x * p! (k,i)) | i <- [1..numColumnas p]]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
-- a la fila k d la matriz p. Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (sumaFilaFila 2 3 p)
--    [[5,1,0],[7,8,15],[4,6,9]]
-- ---------------------------------------------------------------------

sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
sumaFilaFila k l p =
  array ((1,1), (m,n))
        [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = p!(i,j) + p!(l,j)
              | otherwise = p!(i,j)
```

```
-- 2ª solución
sumaFilaFila2 :: Num a => Int -> Int -> Matriz a -> Matriz a
sumaFilaFila2 k l p =
  p // [((k,i), p!(l,i) + p!(k,i)) |i <- [1..numColumnas p]]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (sumaFilaPor 2 3 10 p)
--    [[5,1,0],[43,62,96],[4,6,9]]
-- ---------------------------------------------------------------------

sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
sumaFilaPor k l x p =
  array ((1,1), (m,n))
        [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = p!(i,j) + x*p!(l,j)
              | otherwise = p!(i,j)

-- 2ª solución
sumaFilaPor2 :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
sumaFilaPor2 k l x p =
  p // [((k,i), x * p!(l,i) + p!(k,i)) | i <- [1..numColumnas p]]

-- ---------------------------------------------------------------------
-- Triangularización de matrices                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    buscaIndiceDesde :: (Num a, Eq a) =>
--                        Matriz a -> Int -> Int -> Maybe Int
-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
```

```
--    ghci> buscaIndiceDesde p 3 2
--    Just 2
--    ghci> q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaIndiceDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

buscaIndiceDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
  | null xs   = Nothing
  | otherwise = Just (head xs)
  where xs = [k | ((k,j'),y) <- assocs p, j == j', y /= 0, k >= i]

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    buscaPivoteDesde :: (Num a, Eq a) =>
--                        Matriz a -> Int -> Int -> Maybe a
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaPivoteDesde p 3 2
--    Just 6
--    ghci> q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaPivoteDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

buscaPivoteDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
  | null xs   = Nothing
  | otherwise = Just (head xs)
  where xs = [y | ((k,j'),y) <- assocs p, j == j', y /= 0, k >= i]

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    anuladaColumnaDesde :: (Num a, Eq a) =>
--                           Int -> Int -> Matriz a -> Bool
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
```

```
--    ghci> q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> anuladaColumnaDesde q 3 2
--    True
--    ghci> p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> anuladaColumnaDesde p 3 2
--    False
-- ---------------------------------------------------------------------

anuladaColumnaDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
  buscaIndiceDesde p j (i+1) == Nothing

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
--                             Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
--    ghci> p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matriz Double
--    ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
--    [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]
-- ---------------------------------------------------------------------

anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
                         Matriz a -> Int -> Int -> Matriz a
anulaEltoColumnaDesde p j i =
  sumaFilaPor l i (-(p!(l,j)/a)) p
  where Just l = buscaIndiceDesde p j (i+1)
        a      = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    anulaColumnaDesde :: (Fractional a, Eq a) =>
--                         Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando
-- todos los elementos de la columna j de la matriz p por debajo del la
-- posición (i,j) (se supone que el elemnto p_(i,j) es no nulo). Por
-- ejemplo,
--    ghci> p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matriz Double
```

```
--    ghci> matrizLista (anulaColumnaDesde p 2 1)
--    [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
--    ghci> p = listaMatriz [[4,5],[2,7%2],[6,10]]
--    ghci> matrizLista (anulaColumnaDesde p 1 1)
--    [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
-- ---------------------------------------------------------------------

anulaColumnaDesde :: (Fractional a, Eq a) =>
                     Matriz a -> Int -> Int -> Matriz a
anulaColumnaDesde p j i
  | anuladaColumnaDesde p j i = p
  | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i

-- ---------------------------------------------------------------------
-- Algoritmo de Gauss para triangularizar matrices                     --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    elementosNoNulosColDesde :: (Num a, Eq a) =>
--                                Matriz a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
--    ghci> p = listaMatriz [[3,2],[5,1],[0,4]]
--    ghci> elementosNoNulosColDesde p 1 2
--    [5]
-- ---------------------------------------------------------------------

elementosNoNulosColDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> [a]
elementosNoNulosColDesde p j i =
  [x | ((k,j'),x) <- assocs p, x /= 0, j' == j, k >= i]

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    existeColNoNulaDesde :: (Num a, Eq a) =>
--                            Matriz a -> Int -> Int -> Bool
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la fila i; es decir, si la submatriz de p obtenida
-- eliminando las i-1 primeras filas y las j-1 primeras columnas es no
```

```haskell
-- nula. Por ejemplo,
--    ghci> p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> existeColNoNulaDesde p 2 2
--    False
--    ghci> q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> existeColNoNulaDesde q 2 2
-- ---------------------------------------------------------------------

existeColNoNulaDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Bool
existeColNoNulaDesde p j i =
  or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
  where n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
--                                 Matriz a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
--    ghci> p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde p 2 2
--    Just 2
--    ghci> q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
--    ghci> menorIndiceColNoNulaDesde q 2 2
--    Just 3
--    ghci> r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde r 2 2
--    Nothing
-- ---------------------------------------------------------------------

menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
                             Matriz a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
  | null js   = Nothing
  | otherwise = Just (head js)
  where n  = numColumnas p
        js = [j' | j' <- [j..n],
                   not (null (elementosNoNulosColDesde p j' i))]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la función
--    gaussAux :: (Fractional a, Eq a) =>
--                Matriz a -> Int -> Int -> Matriz a
-- tal que (gaussAux p i j) es la matriz que en el que las i-1 primeras
-- filas y las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
--    1. Si la dimensión de p es (i,j), entonces p.
--    2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--       primeras columnas es nulas, entonces p.
--    3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
--    3.1. j' la primera columna a partir de la j donde p tiene
--         algún elemento no nulo a partir de la fila i,
--    3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--         de p,
--    3.3. i' la primera fila a partir de la i donde la columna j de
--         p1 tiene un elemento no nulo,
--    3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--         la matriz p1 y
--    3.5. p' la matriz obtenida anulando todos los elementos de la
--         columna j de p2 por debajo de la fila i.
-- Por ejemplo,
--    ghci> p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
--    ghci> matrizLista (gaussAux p 2 2)
--    [[1.0,2.0,3.0],[1.0,2.0,4.0],[2.0,0.0,1.0]]
-- ---------------------------------------------------------------------

gaussAux :: (Fractional a, Eq a) => Matriz a -> Int -> Int -> Matriz a
gaussAux p i j
  | dimension p == (i,j)             = p                          -- 1
  | not (existeColNoNulaDesde p j i) = p                          -- 2
  | otherwise                        = gaussAux p' (i+1) (j+1)    -- 3
  where Just j' = menorIndiceColNoNulaDesde p j i                 -- 3.1
        p1      = intercambiaColumnas j j' p                      -- 3.2
        Just i' = buscaIndiceDesde p1 j i                         -- 3.3
        p2      = intercambiaFilas i i' p1                        -- 3.4
        p'      = anulaColumnaDesde p2 j i                        -- 3.5

-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir la función
```

```
--    gauss :: (Fractional a, Eq a) => Matriz a -> Matriz a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
--    ghci> p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--    ghci> gauss p
--    array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                         ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                         ((3,1),0.0),((3,2),0.0),((3,3),0.0)]
--    ghci> matrizLista (gauss p)
--    [[1.0,3.0,2.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
--    ghci> p = listaMatriz [[3.0,2,3],[1,2,4],[1,2,5]]
--    ghci> matrizLista (gauss p)
--    [[3.0,2.0,3.0],[0.0,1.3333333333333335,3.0],[0.0,0.0,1.0]]
--    ghci> p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
--    ghci> matrizLista (gauss p)
--    [[3 % 1,2 % 1,3 % 1],[0 % 1,4 % 3,3 % 1],[0 % 1,0 % 1,1 % 1]]
--    ghci> p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
--    ghci> matrizLista (gauss p)
--    [[1.0,3.0,0.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
-- ---------------------------------------------------------------------

gauss :: (Fractional a, Eq a) => Matriz a -> Matriz a
gauss p = gaussAux p 1 1

-- ---------------------------------------------------------------------
-- Determinante                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 23. Definir la función
--    gaussCAux :: (Fractional a, Eq a) =>
--                 Matriz a -> Int -> Int -> Int -> Matriz a
-- tal que (gaussCAux p i j c) es el par (n,q) donde q es la matriz que
-- en el que las i-1 primeras filas y las j-1 primeras columnas son las
-- de p y las restantes están triangularizadas por el método de Gauss;
-- es decir,
--    1. Si la dimensión de p es (i,j), entonces p.
--    2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--       primeras columnas es nulas, entonces p.
--    3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
```

```haskell
--    3.1. j' la primera columna a partir de la j donde p tiene
--         algún elemento no nulo a partir de la fila i,
--    3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--         de p,
--    3.3. i' la primera fila a partir de la i donde la columna j de
--         p1 tiene un elemento no nulo,
--    3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--         la matriz p1 y
--    3.5. p' la matriz obtenida anulando todos los elementos de la
--         columna j de p2 por debajo de la fila i.
-- y n es c más el número de intercambios de columnas y filas que se han
-- producido durante el cálculo. Por ejemplo,
--    ghci> gaussCAux (listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]) 1 1 0
--    (1,array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                            ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                            ((3,1),0.0),((3,2),0.0),((3,3),0.0)])
-- ---------------------------------------------------------------------

gaussCAux :: (Fractional a, Eq a) =>
             Matriz a -> Int -> Int -> Int -> (Int,Matriz a)
gaussCAux p i j c
  | dimension p == (i,j)             = (c,p)                          -- 1
  | not (existeColNoNulaDesde p j i) = (c,p)                          -- 2
  | otherwise                        = gaussCAux p' (i+1) (j+1) c'    -- 3
  where Just j' = menorIndiceColNoNulaDesde p j i                     -- 3.1
        p1      = intercambiaColumnas j j' p                          -- 3.2
        Just i' = buscaIndiceDesde p1 j i                             -- 3.3
        p2      = intercambiaFilas i i' p1                            -- 3.4
        p'      = anulaColumnaDesde p2 j i                            -- 3.5
        c'      = c + signum (abs (j-j')) + signum (abs (i-i'))

-- ---------------------------------------------------------------------
-- Ejercicio 24. Definir la función
--    gaussC :: (Fractional a, Eq a) => Matriz a -> Matriz a
-- tal que (gaussC p) es el par (n,q), donde q es la triangularización
-- de la matriz p por el método de Gauss y n es el número de
-- intercambios de columnas y filas que se han producido durante el
-- cálculo. Por ejemplo,
--    ghci> gaussC (listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]])
--    (1,array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
```

```
--                              ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                              ((3,1),0.0),((3,2),0.0),((3,3),0.0)])
-- ---------------------------------------------------------------------

gaussC :: (Fractional a, Eq a) => Matriz a -> (Int,Matriz a)
gaussC p = gaussCAux p 1 1 0

-- ---------------------------------------------------------------------
-- Ejercicio 25. Definir la función
--    determinante :: (Fractional a, Eq a) => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> determinante (listaMatriz [[1.0,2,3],[1,3,4],[1,2,5]])
--    2.0
-- ---------------------------------------------------------------------

determinante :: (Fractional a, Eq a) => Matriz a -> a
determinante p = (-1)^c * product (elems (diagonalPral p'))
  where (c,p') = gaussC p
```

# Relación 16

# Vectores y matrices con las librerías

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

-- El objetivo de esta relación es adaptar los ejercicios de las
-- relaciones 14 y 15 (sobre vectores y matrices) usando las librerías
-- Data.Vector y Data.Matrix.
--
-- El manual, con ejemplos, de la librería de vectores de encuentra en
-- http://bit.ly/1PNZ6Br y el de matrices en http://bit.ly/1PNZ9ND
--
-- Para instalar las librerías basta escribir en la consola
--    cabal update
--    cabal install vector
--    cabal install matrix

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

import qualified Data.Vector as V
import Data.Matrix
import Data.Ratio
import Data.Maybe

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

```haskell
-- Tipos de los vectores y de las matrices                              --
-- ---------------------------------------------------------------------

-- Los vectores con elementos de tipo a son del tipo (V.Vector a).
-- Los matrices con elementos de tipo a son del tipo (Matrix a).

-- ---------------------------------------------------------------------
-- Operaciones básicas con matrices                                     --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    listaVector :: Num a => [a] -> V.Vector a
-- tal que (listaVector xs) es el vector correspondiente a la lista
-- xs. Por ejemplo,
--    ghci> listaVector [3,2,5]
--    fromList [3,2,5]
-- ---------------------------------------------------------------------

listaVector :: Num a => [a] -> V.Vector a
listaVector = V.fromList

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    listaMatriz :: Num a => [[a]] -> Matrix a
-- tal que (listaMatriz xss) es la matriz cuyas filas son los elementos
-- de xss. Por ejemplo,
--    ghci> listaMatriz [[1,3,5],[2,4,7]]
--    ( 1 3 5 )
--    ( 2 4 7 )
-- ---------------------------------------------------------------------

listaMatriz :: Num a => [[a]] -> Matrix a
listaMatriz = fromLists

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    numFilas :: Num a => Matrix a -> Int
-- tal que (numFilas m) es el número de filas de la matriz m. Por
-- ejemplo,
```

```
--    numFilas (listaMatriz [[1,3,5],[2,4,7]])  ==  2
-- ---------------------------------------------------------------------

numFilas :: Num a => Matrix a -> Int
numFilas = nrows

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    numColumnas :: Num a => Matrix a -> Int
-- tal que (numColumnas m) es el número de columnas de la matriz
-- m. Por ejemplo,
--    numColumnas (listaMatriz [[1,3,5],[2,4,7]])  ==  3
-- ---------------------------------------------------------------------

numColumnas :: Num a => Matrix a -> Int
numColumnas = ncols

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    dimension :: Num a => Matrix a -> (Int,Int)
-- tal que (dimension m) es la dimensión de la matriz m. Por ejemplo,
--    dimension (listaMatriz [[1,3,5],[2,4,7]])  ==  (2,3)
-- ---------------------------------------------------------------------

dimension :: Num a => Matrix a -> (Int,Int)
dimension p = (nrows p, ncols p)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    matrizLista :: Num a => Matrix a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
--    ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> m
--    ( 5 1 0 )
--    ( 3 2 6 )
--    ghci> matrizLista m
--    [[5,1,0],[3,2,6]]
-- ---------------------------------------------------------------------
```

```haskell
matrizLista :: Num a => Matrix a -> [[a]]
matrizLista = toLists

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    vectorLista :: Num a => V.Vector a -> [a]
-- tal que (vectorLista x) es la lista de los elementos del vector
-- v. Por ejemplo,
--    ghci> let v = listaVector [3,2,5]
--    ghci> v
--    fromList [3,2,5]
--    ghci> vectorLista v
--    [3,2,5]
-- ---------------------------------------------------------------------

vectorLista :: Num a => V.Vector a -> [a]
vectorLista = V.toList

-- ---------------------------------------------------------------------
-- Suma de matrices                                                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    sumaMatrices:: Num a => Matrix a -> Matrix a -> Matrix a
-- tal que (sumaMatrices x y) es la suma de las matrices x e y. Por
-- ejemplo,
--    ghci> let m1 = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> let m2 = listaMatriz [[4,6,3],[1,5,2]]
--    ghci>
--    ( 9 7 3 )
--    ( 4 7 8 )
-- ---------------------------------------------------------------------

sumaMatrices :: Num a => Matrix a -> Matrix a -> Matrix a
sumaMatrices = (+)

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    filaMat :: Num a => Int -> Matrix a -> V.Vector a
```

```
-- tal que (filaMat i p) es el vector correspondiente a la fila i-ésima
-- de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
--    ghci> filaMat 2 p
--    fromList [3,2,6]
--    ghci> vectorLista (filaMat 2 p)
--    [3,2,6]
-- ---------------------------------------------------------------------

filaMat :: Num a => Int -> Matrix a -> V.Vector a
filaMat = getRow

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    columnaMat :: Num a => Int -> Matrix a -> V.Vector a
-- tal que (columnaMat j p) es el vector correspondiente a la columna
-- j-ésima de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
--    ghci> columnaMat 2 p
--    fromList [1,2,5]
--    ghci> vectorLista (columnaMat 2 p)
--    [1,2,5]
-- ---------------------------------------------------------------------

columnaMat :: Num a => Int -> Matrix a -> V.Vector a
columnaMat = getCol

-- ---------------------------------------------------------------------
-- Producto de matrices                                              --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    prodEscalar :: Num a => V.Vector a -> V.Vector a -> a
-- tal que (prodEscalar v1 v2) es el producto escalar de los vectores v1
-- y v2. Por ejemplo,
--    ghci> let v = listaVector [3,1,10]
--    ghci> prodEscalar v v
--    110
-- ---------------------------------------------------------------------
```

```haskell
prodEscalar :: Num a => V.Vector a -> V.Vector a -> a
prodEscalar v1 v2 = V.sum (V.zipWith (*) v1 v2)

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    prodMatrices:: Num a => Matrix a -> Matrix a -> Matrix a
-- tal que (prodMatrices p q) es el producto de las matrices p y q. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,1],[2,4]]
--    ghci> prodMatrices p p
--    ( 11  7 )
--    ( 14 18 )
--    ghci> let q = listaMatriz [[7],[5]]
--    ghci> prodMatrices p q
--    ( 26 )
--    ( 34 )
-- ---------------------------------------------------------------------

prodMatrices :: Num a => Matrix a -> Matrix a -> Matrix a
prodMatrices = (*)

-- ---------------------------------------------------------------------
-- Traspuestas y simétricas                                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    traspuesta :: Num a => Matrix a -> Matrix a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> traspuesta p
--    ( 5 3 )
--    ( 1 2 )
--    ( 0 6 )
-- ---------------------------------------------------------------------

traspuesta :: Num a => Matrix a -> Matrix a
traspuesta = transpose
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    esCuadrada :: Num a => Matrix a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> esCuadrada p
--    False
--    ghci> let q = listaMatriz [[5,1],[3,2]]
--    ghci> esCuadrada q
--    True
-- ---------------------------------------------------------------------

esCuadrada :: Num a => Matrix a -> Bool
esCuadrada p = nrows p == ncols p

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    esSimetrica :: (Num a, Eq a) => Matrix a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
--    ghci> esSimetrica p
--    True
--    ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
--    ghci> esSimetrica q
--    False
-- ---------------------------------------------------------------------

esSimetrica :: (Num a, Eq a) => Matrix a -> Bool
esSimetrica x = x == transpose x

-- ---------------------------------------------------------------------
-- Diagonales de una matriz                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    diagonalPral :: Num a => Matrix a -> V.Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
```

```haskell
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalPral p
--    fromList [5,2]
-- ---------------------------------------------------------------------

diagonalPral :: Num a => Matrix a -> V.Vector a
diagonalPral = getDiag

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    diagonalSec :: Num a => Matrix a -> V.Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalSec p
--    fromList [1,3]
--    ghci> let q = traspuesta p
--    ghci> matrizLista q
--    [[5,3],[1,2],[0,6]]
--    ghci> diagonalSec q
--    fromList [1,2]
-- ---------------------------------------------------------------------

diagonalSec :: Num a => Matrix a -> V.Vector a
diagonalSec p = V.fromList [p!(i,n+1-i) | i <- [1..n]]
  where n = min (nrows p) (ncols p)

-- ---------------------------------------------------------------------
-- Submatrices                                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    submatriz :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> submatriz 2 3 p
--    ( 5 1 )
```

```
--    ( 4 6 )
-- ---------------------------------------------------------------------

submatriz :: Num a => Int -> Int -> Matrix a -> Matrix a
submatriz = minorMatrix

-- ---------------------------------------------------------------------
-- Transformaciones elementales                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    intercambiaFilas :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> intercambiaFilas 1 3 p
--    ( 4 6 9 )
--    ( 3 2 6 )
--    ( 5 1 0 )
-- ---------------------------------------------------------------------

intercambiaFilas :: Num a => Int -> Int -> Matrix a -> Matrix a
intercambiaFilas = switchRows

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la función
--    intercambiaColumnas :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (intercambiaColumnas k l p) es la matriz obtenida
-- intercambiando las columnas k y l de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> intercambiaColumnas 1 3 p
--    ( 0 1 5 )
--    ( 6 2 3 )
--    ( 9 6 4 )
-- ---------------------------------------------------------------------

intercambiaColumnas :: Num a => Int -> Int -> Matrix a -> Matrix a
intercambiaColumnas = switchCols
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir la función
--    multFilaPor :: Num a => Int -> a -> Matrix a -> Matrix a
-- tal que (multFilaPor k x p) es la matriz obtenida multiplicando la
-- fila k de la matriz p por el número x. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> multFilaPor 2 3 p
--    (  5  1  0 )
--    (  9  6 18 )
--    (  4  6  9 )
-- ---------------------------------------------------------------------

multFilaPor :: Num a => Int -> a -> Matrix a -> Matrix a
multFilaPor k x p = scaleRow x k p

-- ---------------------------------------------------------------------
-- Ejercicio 23. Definir la función
--    sumaFilaFila :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
-- a la fila k de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> sumaFilaFila 2 3 p
--    (  5  1  0 )
--    (  7  8 15 )
--    (  4  6  9 )
-- ---------------------------------------------------------------------

sumaFilaFila :: Num a => Int -> Int -> Matrix a -> Matrix a
sumaFilaFila k l p = combineRows k 1 l p

-- ---------------------------------------------------------------------
-- Ejercicio 24. Definir la función
--    sumaFilaPor :: Num a => Int -> Int -> a -> Matrix a -> Matrix a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> sumaFilaPor 2 3 10 p
--    (  5  1  0 )
--    ( 43 62 96 )
--    (  4  6  9 )
```

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

sumaFilaPor :: Num a => Int -> Int -> a -> Matrix a -> Matrix a
sumaFilaPor k l x p = combineRows k x l p

-- ---------------------------------------------------------------------
-- Triangularización de matrices                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 25. Definir la función
--    buscaIndiceDesde :: (Num a, Eq a) =>
--                        Matrix a -> Int -> Int -> Maybe Int
-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaIndiceDesde p 3 2
--    Just 2
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaIndiceDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

-- 1ª definición
buscaIndiceDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
  | null xs   = Nothing
  | otherwise = Just (head xs)
  where xs = [k | k <- [i..nrows p], p!(k,j) /= 0]

-- 2ª definición (con listToMaybe http://bit.ly/212iSgl)
buscaIndiceDesde2 :: (Num a, Eq a) => Matrix a -> Int -> Int -> Maybe Int
buscaIndiceDesde2 p j i =
  listToMaybe [k | k <- [i..nrows p], p!(k,j) /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 26. Definir la función
--    buscaPivoteDesde :: (Num a, Eq a) =>
--                        Matrix a -> Int -> Int -> Maybe a
```

```
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaPivoteDesde p 3 2
--    Just 6
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaPivoteDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

-- 1ª definición
buscaPivoteDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
  | null xs   = Nothing
  | otherwise = Just (head xs)
  where xs = [y | k <- [i..nrows p], let y = p!(k,j), y /= 0]

-- 2ª definición (con listToMaybe http://bit.ly/212iSgl)
buscaPivoteDesde2 :: (Num a, Eq a) => Matrix a -> Int -> Int -> Maybe a
buscaPivoteDesde2 p j i =
  listToMaybe [y | k <- [i..nrows p], let y = p!(k,j), y /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 27. Definir la función
--    anuladaColumnaDesde :: (Num a, Eq a) =>
--                           Int -> Int -> Matrix a -> Bool
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> anuladaColumnaDesde q 3 2
--    True
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> anuladaColumnaDesde p 3 2
--    False
-- ---------------------------------------------------------------------

anuladaColumnaDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
  buscaIndiceDesde p j (i+1) == Nothing
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 28. Definir la función
--    anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
--                             Matrix a -> Int -> Int -> Matrix a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
--    ghci> let p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matrix Double
--    ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
--    [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]
-- ---------------------------------------------------------------------

anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
                         Matrix a -> Int -> Int -> Matrix a
anulaEltoColumnaDesde p j i =
  sumaFilaPor l i (-(p!(l,j)/a)) p
  where Just l = buscaIndiceDesde p j (i+1)
        a      = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 29. Definir la función
--    anulaColumnaDesde :: (Fractional a, Eq a) =>
--                         Matrix a -> Int -> Int -> Matrix a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando
-- todos los elementos de la columna j de la matriz p por debajo del la
-- posición (i,j) (se supone que el elemnto p_(i,j) es no nulo). Por
-- ejemplo,
--    ghci> let p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matrix Double
--    ghci> matrizLista (anulaColumnaDesde p 2 1)
--    [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
--    ghci> let p = listaMatriz [[4,5],[2,7%2],[6,10]]
--    ghci> matrizLista (anulaColumnaDesde p 1 1)
--    [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
-- ---------------------------------------------------------------------

anulaColumnaDesde :: (Fractional a, Eq a) =>
                     Matrix a -> Int -> Int -> Matrix a
anulaColumnaDesde p j i
  | anuladaColumnaDesde p j i = p
```

```
  | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i

-- ---------------------------------------------------------------------
-- Algoritmo de Gauss para triangularizar matrices                     --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 30. Definir la función
--    elementosNoNulosColDesde :: (Num a, Eq a) =>
--                                Matrix a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
--    ghci> let p = listaMatriz [[3,2],[5,1],[0,4]]
--    ghci> elementosNoNulosColDesde p 1 2
--    [5]
-- ---------------------------------------------------------------------

elementosNoNulosColDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> [a]
elementosNoNulosColDesde p j i =
  [y | k <- [i..nrows p], let y = p!(k,j), y /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 31. Definir la función
--    existeColNoNulaDesde :: (Num a, Eq a) =>
--                            Matrix a -> Int -> Int -> Bool
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la j; es decir, si la submatriz de p obtenida eliminando
-- las i-1 primeras filas y las j-1 primeras columnas es no nula. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> existeColNoNulaDesde p 2 2
--    False
--    ghci> let q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> existeColNoNulaDesde q 2 2
--    True
-- ---------------------------------------------------------------------

existeColNoNulaDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Bool
existeColNoNulaDesde p j i =
```

```
  or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
  where n = numColumnas p

-- 2ª solución
existeColNoNulaDesde2 :: (Num a, Eq a) => Matrix a -> Int -> Int -> Bool
existeColNoNulaDesde2 p j i =
  submatrix i m j n p /= zero (m-i+1) (n-j+1)
  where (m,n) = dimension p

-- ---------------------------------------------------------------------
-- Ejercicio 32. Definir la función
--    menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
--                                 Matrix a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
--    ghci> let p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde p 2 2
--    Just 2
--    ghci> let q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
--    ghci> menorIndiceColNoNulaDesde q 2 2
--    Just 3
--    ghci> let r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde r 2 2
--    Nothing
-- ---------------------------------------------------------------------

-- 1ª definición
menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
                             Matrix a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
  | null js   = Nothing
  | otherwise = Just (head js)
  where n  = numColumnas p
        js = [j' | j' <- [j..n],
                   not (null (elementosNoNulosColDesde p j' i))]

-- 2ª definición (con listToMaybe http://bit.ly/212iSgl)
menorIndiceColNoNulaDesde2 :: (Num a, Eq a) =>
                              Matrix a -> Int -> Int -> Maybe Int
```

```haskell
menorIndiceColNoNulaDesde2 p j i =
    listToMaybe [j' | j' <- [j..n],
                      not (null (elementosNoNulosColDesde p j' i))]
    where n  = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 33. Definir la función
--    gaussAux :: (Fractional a, Eq a) =>
--                Matrix a -> Int -> Int -> Matrix a
-- tal que (gaussAux p i j) es la matriz que en el que las i-1 primeras
-- filas y las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
--    1. Si la dimensión de p es (i,j), entonces p.
--    2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--       primeras columnas es nulas, entonces p.
--    3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
--    3.1. j' la primera columna a partir de la j donde p tiene
--         algún elemento no nulo a partir de la fila i,
--    3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--         de p,
--    3.3. i' la primera fila a partir de la i donde la columna j de
--         p1 tiene un elemento no nulo,
--    3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--         la matriz p1 y
--    3.5. p' la matriz obtenida anulando todos los elementos de la
--         columna j de p2 por debajo de la fila i.
-- Por ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
--    ghci> gaussAux p 2 2
--    ( 1.0 2.0 3.0 )
--    ( 1.0 2.0 4.0 )
--    ( 2.0 0.0 1.0 )
-- ---------------------------------------------------------------------

gaussAux :: (Fractional a, Eq a) => Matrix a -> Int -> Int -> Matrix a
gaussAux p i j
    | dimension p == (i,j)             = p                          -- 1
    | not (existeColNoNulaDesde p j i) = p                          -- 2
    | otherwise                        = gaussAux p' (i+1) (j+1)    -- 3
    where Just j' = menorIndiceColNoNulaDesde p j i                 -- 3.1
```

```
          p1      = intercambiaColumnas j j' p                    -- 3.2
          Just i' = buscaIndiceDesde p1 j i                       -- 3.3
          p2      = intercambiaFilas i i' p1                      -- 3.4
          p'      = anulaColumnaDesde p2 j i                      -- 3.5

-- ---------------------------------------------------------------------
-- Ejercicio 34. Definir la función
--    gauss :: (Fractional a, Eq a) => Matrix a -> Matrix a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--    ghci> gauss p
--    ( 1.0 3.0 2.0 )
--    ( 0.0 1.0 0.0 )
--    ( 0.0 0.0 0.0 )
--    ghci> let p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
--    ghci> gauss p
--    ( 3 % 1 2 % 1 3 % 1 )
--    ( 0 % 1 4 % 3 3 % 1 )
--    ( 0 % 1 0 % 1 1 % 1 )
--    ghci> let p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
--    ghci> gauss p
--    ( 1.0 3.0 0.0 )
--    ( 0.0 1.0 0.0 )
--    ( 0.0 0.0 0.0 )
-- ---------------------------------------------------------------------

gauss :: (Fractional a, Eq a) => Matrix a -> Matrix a
gauss p = gaussAux p 1 1

-- ---------------------------------------------------------------------
-- Determinante                                                      --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 35. Definir la función
--    gaussCAux :: (Fractional a, Eq a) =>
--                 Matriz a -> Int -> Int -> Int -> Matriz a
-- tal que (gaussCAux p i j c) es el par (n,q) donde q es la matriz que
-- en el que las i-1 primeras filas y las j-1 primeras columnas son las
```

```haskell
-- de p y las restantes están triangularizadas por el método de Gauss;
-- es decir,
--    1. Si la dimensión de p es (i,j), entonces p.
--    2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--       primeras columnas es nulas, entonces p.
--    3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
--    3.1. j' la primera columna a partir de la j donde p tiene
--         algún elemento no nulo a partir de la fila i,
--    3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--         de p,
--    3.3. i' la primera fila a partir de la i donde la columna j de
--         p1 tiene un elemento no nulo,
--    3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--         la matriz p1 y
--    3.5. p' la matriz obtenida anulando todos los elementos de la
--         columna j de p2 por debajo de la fila i.
-- y n es c más el número de intercambios de columnas y filas que se han
-- producido durante el cálculo. Por ejemplo,
--    ghci> gaussCAux (fromLists [[1.0,2,3],[1,2,4],[1,2,5]]) 1 1 0
--    (1,( 1.0 3.0 2.0 )
--       ( 0.0 1.0 0.0 )
--       ( 0.0 0.0 0.0 ))
-- ---------------------------------------------------------------------

gaussCAux :: (Fractional a, Eq a) =>
             Matrix a -> Int -> Int -> Int -> (Int,Matrix a)
gaussCAux p i j c
    | dimension p == (i,j)             = (c,p)                          -- 1
    | not (existeColNoNulaDesde p j i) = (c,p)                          -- 2
    | otherwise                        = gaussCAux p' (i+1) (j+1) c'    -- 3
    where Just j' = menorIndiceColNoNulaDesde p j i                     -- 3.1
          p1      = switchCols j j' p                                   -- 3.2
          Just i' = buscaIndiceDesde p1 j i                             -- 3.3
          p2      = switchRows i i' p1                                  -- 3.4
          p'      = anulaColumnaDesde p2 j i                            -- 3.5
          c'      = c + signum (abs (j-j')) + signum (abs (i-i'))

-- ---------------------------------------------------------------------
-- Ejercicio 36. Definir la función
--    gaussC :: (Fractional a, Eq a) => Matriz a -> Matriz a
```

```haskell
-- tal que (gaussC p) es el par (n,q), donde q es la triangularización
-- de la matriz p por el método de Gauss y n es el número de
-- intercambios de columnas y filas que se han producido durante el
-- cálculo. Por ejemplo,
--    ghci> gaussC (fromLists [[1.0,2,3],[1,2,4],[1,2,5]])
--    (1, ( 1.0 3.0 2.0 )
--        ( 0.0 1.0 0.0 )
--        ( 0.0 0.0 0.0 )
-- ------------------------------------------------------------------

gaussC :: (Fractional a, Eq a) => Matrix a -> (Int,Matrix a)
gaussC p = gaussCAux p 1 1 0

-- ------------------------------------------------------------------
-- Ejercicio 37. Definir la función
--    determinante :: (Fractional a, Eq a) => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> determinante (fromLists [[1.0,2,3],[1,3,4],[1,2,5]])
--    2.0
-- ------------------------------------------------------------------

determinante :: (Fractional a, Eq a) => Matrix a -> a
determinante p = (-1)^c * V.product (getDiag p')
    where (c,p') = gaussC p
```

# Relación 17

# Cálculo numérico: Diferenciación y métodos de Herón y de Newton

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

-- En esta relación se definen funciones para resolver los siguientes
-- problemas de cálculo numérico:
-- + diferenciación numérica,
-- + cálculo de la raíz cuadrada mediante el método de Herón,
-- + cálculo de los ceros de una función por el método de Newton y
-- + cálculo de funciones inversas.

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

import Test.QuickCheck

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    derivada :: Double -> (Double -> Double) -> Double -> Double
```

```haskell
-- tal que (derivada a f x) es el valor de la derivada de la función f
-- en el punto x con aproximación a. Por ejemplo,
--    derivada 0.001 sin pi  ==  -0.9999998333332315
--    derivada 0.001 cos pi  ==  4.999999583255033e-4
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir las funciones
--    derivadaBurda :: (Double -> Double) -> Double -> Double
--    derivadaFina  :: (Double -> Double) -> Double -> Double
--    derivadaSuper :: (Double -> Double) -> Double -> Double
-- tales que
--    * (derivadaBurda f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.01,
--    * (derivadaFina f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.0001.
--    * (derivadaSuper f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.000001.
-- Por ejemplo,
--    derivadaBurda cos pi  ==  4.999958333473664e-3
--    derivadaFina  cos pi  ==  4.999999969612645e-5
--    derivadaSuper cos pi  ==  5.000444502911705e-7
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    derivadaFinaDelSeno :: Double -> Double
-- tal que (derivadaFinaDelSeno x) es el valor de la derivada fina del
```

```haskell
-- seno en x. Por ejemplo,
--    derivadaFinaDelSeno pi  ==  -0.9999999983354436
-- ---------------------------------------------------------------------

derivadaFinaDelSeno :: Double -> Double
derivadaFinaDelSeno = derivadaFina sin

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. En los siguientes apartados de este ejercicio se va a
-- calcular la raíz cuadrada de un número basándose en las siguientes
-- propiedades:
-- + Si y es una aproximación de la raíz cuadrada de x, entonces
--   (y+x/y)/2 es una aproximación mejor.
-- + El límite de la sucesión definida por
--       x_0     = 1
--       x_{n+1} = (x_n+x/x_n)/2
--   es la raíz cuadrada de x.
--
-- Definir, por recursión, la función
--    raiz :: Double -> Double
-- tal que (raiz x) es la raíz cuadrada de x calculada usando la
-- propiedad anterior con una aproximación de 0.00001 y tomando como
-- valor inicial 1. Por ejemplo,
--    raiz 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raiz :: Double -> Double
raiz x = raizAux 1
  where raizAux y | aceptable y = y
                  | otherwise   = raizAux (mejora y)
        aceptable y = abs(y*y-x) < 0.00001
        mejora y    = 0.5*(y+x/y)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir el operador
--    (~=) :: Double -> Double -> Bool
```

```haskell
-- tal que (x ~= y) si |x-y| < 0.001. Por ejemplo,
--    3.05 ~= 3.07        ==  False
--    3.00005 ~= 3.00007  == True
-- ---------------------------------------------------------------------

infix 5 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < 0.001

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raiz x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raiz :: Double -> Bool
prop_raiz x =
  raiz x' ^ 2 ~= x'
  where x' = abs x

-- La comprobación es
--    ghci> 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)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir, por iteración con until, la función
--    raizI :: Double -> Double
-- tal que (raizI x) es la raíz cuadrada de x calculada usando la
-- propiedad anterior. Por ejemplo,
--    raizI 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raizI :: Double -> Double
raizI x = until aceptable mejora 1
  where aceptable y = abs(y*y-x) < 0.00001
        mejora y    = 0.5*(y+x/y)

-- ---------------------------------------------------------------------
-- Ejercicio 3.6. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizI x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizI :: Double -> Bool
prop_raizI x =
  raizI x' ^ (2::Int) ~= x'
  where x' = abs x

-- La comprobación es
--    ghci> 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 :: (Double -> Double) -> Double
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCero cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCeroAux f 1
  where puntoCeroAux f' x | aceptable x = x
                          | otherwise   = puntoCeroAux f' (mejora x)
        aceptable b = abs (f b) < 0.00001
        mejora b    = b - f b / derivadaFina f b

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por iteración con until, la función
--    puntoCeroI :: (Double -> Double) -> Double
-- tal que (puntoCeroI f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCeroI cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

puntoCeroI :: (Double -> Double) -> Double
puntoCeroI f = until aceptable mejora 1
  where aceptable b = abs (f b) < 0.00001
        mejora b    = b - f b / derivadaFina f b

-- ---------------------------------------------------------------------
-- Funciones inversas                                                  --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5. En este ejercicio se usará la función puntoCero para
-- definir la inversa de distintas funciones.
-- ---------------------------------------------------------------------
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, usando puntoCero, la función
--    raizCuadrada :: Double -> Double
-- tal que (raizCuadrada x) es la raíz cuadrada de x. Por ejemplo,
--    raizCuadrada 9  ==  3.000000002941184
-- ---------------------------------------------------------------------

raizCuadrada :: Double -> Double
raizCuadrada a = puntoCero f
  where f x = x*x-a

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizCuadrada x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizCuadrada :: Double -> Bool
prop_raizCuadrada x =
  raizCuadrada x' ^ (2::Int) ~= x'
  where x' = abs x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir, usando puntoCero, la función
--    raizCubica :: Double -> Double
-- tal que (raizCubica x) es la raíz cúbica de x. Por ejemplo,
--    raizCubica 27  ==  3.0000000000196048
-- ---------------------------------------------------------------------

raizCubica :: Double -> Double
raizCubica a = puntoCero f
  where f x = x*x*x-a

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

```haskell
-- Ejercicio 5.4. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizCubica x)^3 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizCubica :: Double -> Bool
prop_raizCubica x =
  raizCubica x ^ (3::Int) ~= x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.5. Definir, usando puntoCero, la función
--    arcoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcoseno 1  == 1.5665489428306574
-- ---------------------------------------------------------------------

arcoseno :: Double -> Double
arcoseno a = puntoCero f
  where f x = sin x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.6. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--    sin (arcoseno x) ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_arcoseno :: Double -> Bool
prop_arcoseno x =
  sin (arcoseno x') ~= x'
  where x' = abs (x - fromIntegral (truncate x))

-- La comprobación es
--    ghci> quickCheck prop_arcoseno
--    OK, passed 100 tests.
```

```haskell
-- Otra forma de expresar la propiedad es
prop_arcoseno2 :: Property
prop_arcoseno2 =
  forAll (choose (0,1)) $ \x -> sin (arcoseno x) ~= x

-- La comprobación es
--    ghci> quickCheck prop_arcoseno2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--    arcocoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcocoseno 0  == 1.5707963267949576
-- ---------------------------------------------------------------------

arcocoseno :: Double -> Double
arcocoseno a = puntoCero f
  where f x = cos x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.8. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--    cos (arcocoseno x) ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_arcocoseno :: Double -> Bool
prop_arcocoseno x =
  cos (arcocoseno x') ~= x'
  where x' = abs (x - fromIntegral (truncate x))

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

-- Otra forma de expresar la propiedad es
prop_arcocoseno2 :: Property
prop_arcocoseno2 =
```

```
  forAll (choose (0,1)) $ \x -> cos (arcocoseno x) ~= x

-- La comprobación es
--    ghci> quickCheck prop_arcocoseno2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--    inversa :: (Double -> Double) -> Double -> Double
-- tal que (inversa g x) es el valor de la inversa de g en x. Por
-- ejemplo,
--    inversa (^2) 9  ==  3.000000002941184
-- ---------------------------------------------------------------------

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

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

raizCuadrada', raizCubica', arcoseno', arcocoseno' :: Double -> Double
raizCuadrada' = inversa (^2)
raizCubica'   = inversa (^3)
arcoseno'     = inversa sin
arcocoseno'   = inversa cos
```

# Relación 18

# Cálculo numérico (2): límites, bisección, integrales y bajada

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

-- En esta relación se definen funciones para resolver los siguientes
-- problemas de cálculo numérico:
-- + Cálculo de límites.
-- + Cálculo de los ceros de una función por el método de la bisección.
-- + Cálculo de raíces enteras.
-- + Cálculo de integrales por el método de los rectángulos.
-- + Algoritmo de bajada para resolver un sistema triangular inferior.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.Matrix

-- ---------------------------------------------------------------------
-- § Cálculo de límites                                                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    limite :: (Double -> Double) -> Double -> Double
```

```haskell
-- tal que (limite f a) es el valor de f en el primer término x tal que,
-- para todo y entre x+1 y x+100, el valor absoluto de la diferencia
-- entre f(y) y f(x) es menor que a. Por ejemplo,
--    limite (\n -> (2*n+1)/(n+5)) 0.001  ==  1.9900110987791344
--    limite (\n -> (1+1/n)**n) 0.001     ==  2.714072874546881
-- ---------------------------------------------------------------------

limite :: (Double -> Double) -> Double -> Double
limite f a =
  head [f x | x <- [1..],
              maximum [abs (f y - f x) | y <- [x+1..x+100]] < a]

-- ---------------------------------------------------------------------
-- § Ceros de una función por el método de la bisección                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 2. El método de bisección para calcular un cero de una
-- función en el intervalo [a,b] se basa en el teorema de Bolzano:
--    "Si f(x) es una función continua en el intervalo [a, b], y si,
--    además, en los extremos del intervalo la función f(x) toma valores
--    de signo opuesto (f(a) * f(b) < 0), entonces existe al menos un
--    valor c en (a, b) para el que f(c) = 0".
--
-- El método para calcular un cero de la función f en el intervalo [a,b]
-- con un error menor que e consiste en tomar el punto medio del
-- intervalo c = (a+b)/2 y considerar los siguientes casos:
-- (*) Si |f(c)| < e, hemos encontrado una aproximación del punto que
--     anula f en el intervalo con un error aceptable.
-- (*) Si f(c) tiene signo distinto de f(a), repetir el proceso en el
--     intervalo [a,c].
-- (*) Si no, repetir el proceso en el intervalo [c,b].
--
-- Definir la función
--    biseccion :: (Double -> Double) -> Double -> Double -> Double -> Double
-- tal que (biseccion f a b e) es una aproximación del punto del
-- intervalo [a,b] en el que se anula la función f, con un error menor
-- que e, calculada mediante el método de la bisección. Por ejemplo,
--    biseccion (\x -> x^2 - 3) 0 5 0.01             ==  1.7333984375
--    biseccion (\x -> x^3 - x - 2) 0 4 0.01         ==  1.521484375
```

```haskell
--    biseccion cos 0 2 0.01                          ==  1.5625
--    biseccion (\x -> log (50-x) - 4) (-10) 3 0.01   ==  -5.125
-- ---------------------------------------------------------------------

-- 1ª solución
biseccion :: (Double -> Double) -> Double -> Double -> Double -> Double
biseccion f a b e
  | abs (f c) < e   = c
  | (f a)*(f c) < 0 = biseccion f a c e
  | otherwise       = biseccion f c b e
  where c = (a+b)/2

-- 2ª solución
biseccion2 :: (Double -> Double) -> Double -> Double -> Double -> Double
biseccion2 f a b e = aux a b
  where aux a b | abs (f c) < e   = c
                | (f a)*(f c) < 0 = aux a c
                | otherwise       = aux c b
          where c = (a+b)/2

-- ---------------------------------------------------------------------
-- § Cálculo de raíces enteras                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    raizEnt :: Integer -> Integer -> Integer
-- tal que (raizEnt x n) es la raíz entera n-ésima de x; es decir, el
-- mayor número entero y tal que y^n <= x. Por ejemplo,
--    raizEnt  8 3      ==  2
--    raizEnt  9 3      ==  2
--    raizEnt 26 3      ==  2
--    raizEnt 27 3      ==  3
--    raizEnt (10^50) 2 ==  100000000000000000000000
--
-- Comprobar con QuickCheck que para todo número natural n,
--     raizEnt (10^(2*n)) 2 == 10^n
-- ---------------------------------------------------------------------

-- 1ª definición
```

```haskell
raizEnt1 :: Integer -> Integer -> Integer
raizEnt1 x n =
    last (takeWhile (\y -> y^n <= x) [0..])

-- 2ª definición
raizEnt2 :: Integer -> Integer -> Integer
raizEnt2 x n =
    floor ((fromIntegral x)**(1 / fromIntegral n))

-- Nota. La definición anterior falla para números grandes. Por ejemplo,
--    λ> raizEnt2 (10^50) 2 == 10^25
--    False

-- 3ª definición
raizEnt3 :: Integer -> Integer -> Integer
raizEnt3 x n = aux (1,x)
  where aux (a,b) | d == x    = c
                  | c == a    = c
                  | d < x     = aux (c,b)
                  | otherwise = aux (a,c)
          where c = (a+b) `div` 2
                d = c^n

-- Comparación de eficiencia
--    λ> raizEnt1 (10^14) 2
--    10000000
--    (6.15 secs, 6,539,367,976 bytes)
--    λ> raizEnt2 (10^14) 2
--    10000000
--    (0.00 secs, 0 bytes)
--    λ> raizEnt3 (10^14) 2
--    10000000
--    (0.00 secs, 25,871,944 bytes)
--
--    λ> raizEnt2 (10^50) 2
--    99999999999999998758486016
--    (0.00 secs, 0 bytes)
--    λ> raizEnt3 (10^50) 2
--    100000000000000000000000000
--    (0.00 secs, 0 bytes)
```

```haskell
-- La propiedad es
prop_raizEnt :: Integer -> Bool
prop_raizEnt n =
    raizEnt3 (10^(2*m)) 2 == 10^m
    where m = abs n

-- La comprobación es
--    λ> quickCheck prop_raizEnt
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- § Integración por el método de los rectángulos                      --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4. La integral definida de una función f entre los límites
-- a y b puede calcularse mediante la regla del rectángulo
-- (ver en http://bit.ly/1FDhZ1z) usando la fórmula
--    h * (f(a+h/2) + f(a+h+h/2) + f(a+2h+h/2) + ... + f(a+n*h+h/2))
-- con a+n*h+h/2 <= b < a+(n+1)*h+h/2 y usando valores pequeños para h.
--
-- Definir la función
--    integral :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
-- tal que (integral a b f h) es el valor de dicha expresión. Por
-- ejemplo, el cálculo de la integral de f(x) = x^3 entre 0 y 1, con
-- paso 0.01, es
--    integral 0 1 (^3) 0.01  ==  0.24998750000000042
-- Otros ejemplos son
--    integral 0 1 (^4) 0.01                        ==  0.19998333362500048
--    integral 0 1 (\x -> 3*x^2 + 4*x^3) 0.01       ==  1.9999250000000026
--    log 2 - integral 1 2 (\x -> 1/x) 0.01         ==  3.124931644782336e-6
--    pi - 4 * integral 0 1 (\x -> 1/(x^2+1)) 0.01  ==  -8.333333331389525e-6
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

integral :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
integral a b f h = h * suma (a+h/2) b (+h) f
```

```haskell
-- (suma a b s f) es l valor de
--    f(a) + f(s(a)) + f(s(s(a)) + ... + f(s(...(s(a))...))
-- hasta que s(s(...(s(a))...)) > b. Por ejemplo,
--    suma 2 5 (1+) (^3)  ==  224
suma :: (Ord t, Num a) => t -> t -> (t -> t) -> (t -> a) -> a
suma a b s f = sum [f x | x <- sucesion a b s]

-- (sucesion x y s) es la lista
--    [a, s(a), s(s(a), ..., s(...(s(a))...)]
-- hasta que s(s(...(s(a))...)) > b. Por ejemplo,
--    sucesion 3 20 (+2)  ==  [3,5,7,9,11,13,15,17,19]
sucesion :: Ord a => a -> a -> (a -> a) -> [a]
sucesion a b s = takeWhile (<=b) (iterate s a)

-- 2ª solución
-- ===========

integral2 :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
integral2 a b f h
    | a+h/2 > b = 0
    | otherwise = h * f (a+h/2) + integral2 (a+h) b f h

-- 3ª solución
-- ===========

integral3 :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
integral3 a b f h = aux a where
  aux x | x+h/2 > b = 0
        | otherwise = h * f (x+h/2) + aux (x+h)

-- Comparación de eficiencia
--    ghci> integral 0 10 (^3) 0.00001
--    2499.9999998811422
--    (4.62 secs, 1084774336 bytes)
--    ghci> integral2 0 10 (^3) 0.00001
--    2499.999999881125
--    (7.90 secs, 1833360768 bytes)
--    ghci> integral3 0 10 (^3) 0.00001
--    2499.999999881125
```

```
--    (7.27 secs, 1686056080 bytes)

-- ---------------------------------------------------------------------
-- § Algoritmo de bajada para resolver un sistema triangular inferior --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5. Un sistema de ecuaciones lineales Ax = b es triangular
-- inferior si todos los elementos de la matriz A que están por encima
-- de la diagonal principal son nulos; es decir, es de la forma
--    a(1,1)*x(1)                                              = b(1)
--    a(2,1)*x(1) + a(2,2)*x(2)                                = b(2)
--    a(3,1)*x(1) + a(3,2)*x(2) + a(3,3)*x(3)                  = b(3)
--    ...
--    a(n,1)*x(1) + a(n,2)*x(2) + a(n,3)*x(3) +...+ a(x,x)*x(n) = b(n)
--
-- El sistema es compatible si, y sólo si, el producto de los elementos
-- de la diagonal principal es distinto de cero. En este caso, la
-- solución se puede calcular mediante el algoritmo de bajada:
--    x(1) = b(1) / a(1,1)
--    x(2) = (b(2) - a(2,1)*x(1)) / a(2,2)
--    x(3) = (b(3) - a(3,1)*x(1) - a(3,2)*x(2)) / a(3,3)
--    ...
--    x(n) = (b(n) - a(n,1)*x(1) - a(n,2)*x(2) -...- a(n,n-1)*x(n-1)) / a(n,n)
--
-- Definir la función
--    bajada :: Matrix Double -> Matrix Double -> Matrix Double
-- tal que (bajada a b) es la solución, mediante el algoritmo de bajada,
-- del sistema compatible triangular superior ax = b. Por ejemplo,
--    ghci> let a = fromLists [[2,0,0],[3,1,0],[4,2,5.0]]
--    ghci> let b = fromLists [[3],[6.5],[10]]
--    ghci> bajada a b
--    ( 1.5 )
--    ( 2.0 )
--    ( 0.0 )
-- Es decir, la solución del sistema
--    2x            = 3
--    3x + y        = 6.5
--    4x + 2y + 5 z = 10
-- es x=1.5, y=2 y z=0.
```

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

bajada :: Matrix Double -> Matrix Double -> Matrix Double
bajada a b = fromLists [[x i] | i <- [1..m]]
    where m = nrows a
          x k = (b!(k,1) - sum [a!(k,j) * x j | j <- [1..k-1]]) / a!(k,k)
```

# Relación 19

# Cálculo numérico en Maxima: Diferenciación y métodos de Herón y de Newton

```
/* ----------------------------------------------------------------------
   Introducción
   ----------------------------------------------------------------------

   En esta relación se programan en Maxima funciones para resolver los
   siguientes problemas de cálculo numérico:
   + diferenciación numérica,
   + cálculo de la raíz cuadrada mediante el método de Herón,
   + cálculo de los ceros de una función por el método de Newton y
   + cálculo de funciones inversas.                                    */

/* ----------------------------------------------------------------------
   Diferenciación numérica
   ---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
   Ejercicio 1.1. Definir la función derivada 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,
      (%i2) derivada (0.001,sin,%pi);
      (%o2) - 0.99999983333334
      (%i3) derivada (0.001,cos,%pi);
      (%o3) 4.999999583255033E-4
```

```
   ------------------------------------------------------------------ */

derivada (a,f,x) := (f(x+a)-f(x))/a$

/* ------------------------------------------------------------------
   Ejercicio 1.2. Definir las funciones derivadaBurda, derivadaFina y
   derivadaSuper 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.
      + derivadaSuper (f,x) es el valor de la derivada de la función f
        en el punto x con aproximación 0.000001.
   Por ejemplo,
      (%i4) derivadaBurda (cos,%pi);
      (%o4) 0.0049999583334736
      (%i5) derivadaFina (cos,%pi);
      (%o5) 4.999999969612645E-5
      (%i6) derivadaSuper (cos,%pi);
      (%o6) 5.000444502911705E-7
   ------------------------------------------------------------------ */

derivadaBurda (f,x) := derivada (0.01,     f,x)$
derivadaFina  (f,x) := derivada (0.0001,   f,x)$
derivadaSuper (f,x) := derivada (0.000001,f,x)$

/* ------------------------------------------------------------------
   Ejercicio 1.3. Definir la función derivadaFinaDelSeno tal que
   (derivadaFinaDelSeno x) es el valor de la derivada fina del seno en
   x. Por ejemplo,
      (%i7) derivadaFinaDelSeno (%pi);
      (%o7) - 0.99999999833333
   ------------------------------------------------------------------ */

derivadaFinaDelSeno (x) := derivadaFina (sin,x)$

/* ------------------------------------------------------------------
   Cálculo de la raíz cuadrada
   ------------------------------------------------------------------ */
```

```
/* ----------------------------------------------------------------------
   Ejercicio 2. El método de Herón para calcular la raíz cuadrada de
   un número se basa en las siguientes propiedades:
   + Si y es una aproximación de la raíz cuadrada de x, entonces
     (y+x/y)/2 es una aproximación mejor.
   + El límite de la sucesión definida por
         x(0)   = 1
         x(n+1) = (x(n)+x/x(n))/2
     es la raíz cuadrada de x.

   Definir, por iteración con unless, la función raiz tal que raiz(x) es
   la raíz cuadrada de x calculada usando la  propiedad anterior. Por
   ejemplo,
      (%i8) raiz (9);
      (%o8) 3.000000001396984
   ---------------------------------------------------------------------- */

raiz (x) := block ([y:1],
  unless abs (y*y-x) < 0.00001 do
    y : 0.5*(y+x/y),
  y)$

/* ----------------------------------------------------------------------
   Ceros de una función
   ---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
   Ejercicio 3. 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.

   Definir, por iteración con unless, la función puntoCero tal que
   puntoCero(f) es un cero de la función f calculado usando el
   método anterior. Por ejemplo,
      (%i9) puntoCero (cos);
```

```
      (%o9) 1.570796326794957
   ---------------------------------------------------------------------- */

puntoCero (f) := block ([b:1],
  unless abs (f(b)) < 0.00001 do
     b : b - f(b) / derivadaFina (f,b),
  float (b))$

/* ----------------------------------------------------------------------
   Funciones inversas
   ---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
   Ejercicio 4.1. En este ejercicio se usará la función puntoCero para
   definir la inversa de distintas funciones.

   Definir, usando puntoCero, la función raizCuadrada tal que
   raizCuadrada x es la raíz cuadrada de x. Por ejemplo,
      (%i10) raizCuadrada (9);
      (%o10) 3.000000002941184
   ---------------------------------------------------------------------- */

raizCuadrada (a) := block (
    local (f),
    f (x) := x^2-a,
    puntoCero (f))$

/* ----------------------------------------------------------------------
   Ejercicio 4.2. Definir, usando puntoCero, la función raizCubica tal
   que raizCubica (x) es la raíz cúbica de x. Por ejemplo,
      (%i11) raizCubica (27);
      (%o11) 3.000000000019604
   ---------------------------------------------------------------------- */

raizCubica (a) := block (
  local (f),
  f (x) := x^3-a,
  puntoCero (f))$

/* ----------------------------------------------------------------------
```

```
   Ejercicio 4.3. Definir, usando puntoCero, la función arcoseno tal que
   arcoseno(x) es el arcoseno de x. Por ejemplo,
      (%i12) arcoseno (1);
      (%o12) 1.566548942830657
   ---------------------------------------------------------------------- */

arcoseno (a) := block (
  local (f),
  f (x) := sin (x) - a,
  puntoCero (f))$

/* ---------------------------------------------------------------------
   Ejercicio 4.4. Definir, usando puntoCero, la función arcocoseno tal
   que arcocoseno(x) es el arcoseno de x. Por ejemplo,
      (%i13) arcocoseno (0);
      (%o13) 1.570796326794957
   ---------------------------------------------------------------------- */

arcocoseno (a) := block (
  local (f),
  f (x) := cos (x) - a,
  puntoCero (f))$

/* ---------------------------------------------------------------------
   Ejercicio 4.5. Definir, usando puntoCero, la función inversa tal que
   inversa(g,a) es el valor de la inversa de g en a Por ejemplo,
      (%i14) cuadrado (x) := x^2$
      (%i15) inversa (cuadrado,9);
      (%o15) 3.000000002941184
      (%i16) inversa (lambda ([x], x^2), 9);
      (%o16) 3.000000002941184
   ---------------------------------------------------------------------- */

inversa (g,a) := block (
  local (f),
  f (x) := g (x) - a,
  puntoCero (f))$

/* ---------------------------------------------------------------------
   Ejercicio 4.6. Redefinir, usando inversa, las funciones raizCuadrada,
```

```
   raizCubica, arcoseno y arcocoseno. Por ejemplo,
      (%i17) raizCuadradaI (9);
      (%o17) 3.000000002941184
      (%i18) raizCubicaI (9);
      (%o18) 2.080083824207376
      (%i19) arcosenoI (1);
      (%o19) 1.566548942830657
      (%i20) arcocosenoI (0);
      (%o20) 1.570796326794957
   ------------------------------------------------------------------- */

raizCuadradaI (a) := inversa (lambda ([x], x^2), a)$
raizCubicaI   (a) := inversa (lambda ([x], x^3), a)$
arcosenoI     (a) := inversa (sin, a)$
arcocosenoI   (a) := inversa (cos, a)$
```

# Relación 20

# Estadística descriptiva

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es definir las principales medidas
-- estadísticas de centralización (medias, mediana y modas) y de
-- dispersión (rango, desviación media, varianza y desviación típica)
-- que se estudian en 3º de ESO (como en http://bit.ly/2FbXOQm ).
--
-- En las soluciones de los ejercicios se pueden usar las siguientes
-- funciones de la librería Data.List fromIntegral, genericLength, sort,
-- y group (cuya descripción se puede consultar en el "Manual de
-- funciones básicas de Haskell" http://bit.ly/2VICngx ).

-- ---------------------------------------------------------------------
-- Librerías auxiliares                                                --
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck
import Graphics.Gnuplot.Simple

-- ---------------------------------------------------------------------
-- Medidas de centralización                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
```

```haskell
--    media :: Floating a => [a] -> a
-- tal que (media xs) es la media aritmética de los números de la lista
-- xs. Por ejemplo,
--    media [4,8,4,5,9]  ==  6.0
-- ---------------------------------------------------------------------

media :: Floating a => [a] -> a
media xs = sum xs / genericLength xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. La mediana de una lista de valores es el valor de
-- la lista que ocupa el lugar central de los valores ordenados de menor
-- a mayor. Si el número de datos es impar se toma como valor de la
-- mediana el valor central. Si el número de datos es par se toma como
-- valor de la mediana la media aritmética de los dos valores
-- centrales.
--
-- Definir la función
--    mediana :: (Floating a, Ord a) => [a] -> a
-- tal que (mediana xs) es la mediana de la lista xs. Por ejemplo,
--    mediana [2,3,6,8,9]    ==  6.0
--    mediana [2,3,4,6,8,9]  ==  5.0
--    mediana [9,6,8,4,3,2]  ==  5.0
-- ---------------------------------------------------------------------

mediana :: (Floating a, Ord a) => [a] -> a
mediana xs | odd n     = ys !! i
           | otherwise = media [ys !! (i-1), ys !! i]
  where ys = sort xs
        n  = length xs
        i  = n `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comprobar con QuickCheck que para cualquier lista no
-- vacía xs el número de elementos de xs menores que su mediana es menor
-- o igual que la mitad de los elementos de xs y lo mismo pasa con los
-- mayores o iguales que la mediana.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_mediana :: (Floating a, Ord a) => [a] -> Property
prop_mediana xs =
  not (null xs) ==>
  genericLength [x | x <- xs, x < m] <= n/2 &&
  genericLength [x | x <- xs, x > m] <= n/2
  where m = mediana xs
        n = genericLength xs

-- La comprobación es
--    ghci> quickCheck prop_mediana
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    frecuencias :: Ord a => [a] -> [(a,Int)]
-- tal que (frecuencias xs) es la lista formada por los elementos de xs
-- junto con el número de veces que aparecen en xs. Por ejemplo,
--    frecuencias "sosos" ==  [('o',2),('s',3)]
--
-- Nota: El orden de los pares no importa
-- ---------------------------------------------------------------------

-- 1ª solución
frecuencias :: Ord a => [a] -> [(a,Int)]
frecuencias xs = [(y,ocurrencias y xs) | y <- sort (nub xs)]
  where ocurrencias y zs = length [1 | x <- zs, x == y]

-- 2ª solución
frecuencias2 :: Ord a => [a] -> [(a,Int)]
frecuencias2 xs = [(y,1 + length ys) | (y:ys) <- group (sort xs)]

-- 3ª solución
frecuencias3 :: Ord a => [a] -> [(a,Int)]
frecuencias3 = map (\ys@(y:_) -> (y,length ys)) . group . sort

-- ---------------------------------------------------------------------
-- Ejercicio 5. Las modas de una lista son los elementos de la lista
-- que más se repiten.
--
-- Definir la función
```

```haskell
--    modas :: Ord a => [a] -> [a]
-- tal que (modas xs) es la lista ordenada de las modas de xs. Por
-- ejemplo
--    modas [7,3,7,5,3,1,6,9,6]  ==  [3,6,7]
-- ---------------------------------------------------------------------

modas :: Ord a => [a] -> [a]
modas xs = [y | (y,n) <- ys, n == m]
  where ys = frecuencias xs
        m  = maximum (map snd ys)

-- ---------------------------------------------------------------------
-- Ejercicio 6. La media geométrica de una lista de n números es la
-- raíz n-ésima del producto de todos los números.
--
-- Definir la función
--    mediaGeometrica :: Floating a => [a] -> a
-- tal que (mediaGeometrica xs) es la media geométrica de xs. Por
-- ejemplo,
--    mediaGeometrica [2,18]   ==  6.0
--    mediaGeometrica [3,1,9]  ==  3.0
-- ---------------------------------------------------------------------

mediaGeometrica :: Floating a => [a] -> a
mediaGeometrica xs = product xs ** (1 / genericLength xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck que la media geométrica de
-- cualquier lista no vacía de números no negativos es siempre menor o
-- igual que la media aritmética.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mediaGeometrica :: (Floating a, Ord a) => [a] -> Property
prop_mediaGeometrica xs =
    not (null xs) ==>
    mediaGeometrica ys <= media ys
    where ys = map abs xs

-- La comprobación es
```

```
--    ghci> quickCheck prop_mediaGeometrica
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Medidas de dispersión                                             --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 8. El recorrido (o rango) de una lista de valores es la
-- diferencia entre el mayor y el menor.
--
-- Definir la función
--    rango :: (Num a, Ord a) => [a] -> a
-- tal que (rango xs) es el rango de xs. Por ejemplo,
--    rango [4,2,4,7,3]  ==  5
-- ---------------------------------------------------------------------

-- 1ª solución
rango :: (Num a, Ord a) => [a] -> a
rango xs = maximum xs - minimum xs

-- 2ª solución
rango2 :: (Num a, Ord a) => [a] -> a
rango2 xs = maximo - minimo
  where (y:ys)           = xs
        (minimo,maximo)  = aux ys (y,y)
        aux [] (a,b)     = (a,b)
        aux (z:zs) (a,b) = aux zs (min a z, max z b)

-- ---------------------------------------------------------------------
-- Ejercicio 9. La desviación media de una lista de datos xs es la
-- media de las distancias de los datos a la media xs, donde la
-- distancia entre dos elementos es el valor absoluto de su
-- diferencia. Por ejemplo, la desviación media de [4,8,4,5,9] es 2 ya
-- que la media de [4,8,4,5,9] es 6 y
--      (|4-6| + |8-6| + |4-6| + |5-6| + |9-6|) / 5
--    = (2 + 2 + 2 + 1 + 3) / 5
--    = 2
--
-- Definir la función
```

```haskell
--    desviacionMedia :: Floating a => [a] -> a
-- tal que (desviacionMedia xs) es la desviación media de xs. Por
-- ejemplo,
--    desviacionMedia [4,8,4,5,9]       ==  2.0
--    desviacionMedia (replicate 10 3)  ==  0.0
-- ---------------------------------------------------------------------

desviacionMedia :: Floating a => [a] -> a
desviacionMedia xs = media [abs (x - m) | x <- xs]
  where m = media xs

-- ---------------------------------------------------------------------
-- Ejercicio 10. La varianza de una lista datos es la media de los
-- cuadrados de las distancias de los datos a la media. Por ejemplo, la
-- varianza de [4,8,4,5,9] es 4.4 ya que la media de [4,8,4,5,9] es 6 y
--      ((4-6)^2 + (8-6)^2 + (4-6)^2 + (5-6)^2 + (9-6)^2) / 5
--    = (4 + 4 + 4 + 1 + 9) / 5
--    = 4.4
--
-- Definir la función
--    varianza :: Floating a => [a] -> a
-- tal que (desviacionMedia xs) es la varianza de xs. Por ejemplo,
--    varianza [4,8,4,5,9]       ==  4.4
--    varianza (replicate 10 3)  ==  0.0
-- ---------------------------------------------------------------------

varianza :: Floating a => [a] -> a
varianza xs = media [(x-m)^2 | x <- xs]
  where m = media xs

-- ---------------------------------------------------------------------
-- Ejercicio 11. La desviación típica de una lista de datos es la raíz
-- cuadrada de su varianza.
--
-- Definir la función
--    desviacionTipica :: Floating a => [a] -> a
-- tal que (desviacionTipica xs) es la desviación típica de xs. Por
-- ejemplo,
--    desviacionTipica [4,8,4,5,9]       ==  2.0976176963403033
--    desviacionTipica (replicate 10 3)  ==  0.0
```

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

-- 1ª definición
desviacionTipica :: Floating a => [a] -> a
desviacionTipica xs = sqrt (varianza xs)

-- 2ª definición
desviacionTipica2 :: Floating a => [a] -> a
desviacionTipica2 = sqrt . varianza

-- ---------------------------------------------------------------------
-- Regresión lineal                                                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 12. Dadas dos listas de valores
--    xs = [x(1), x(2), ..., x(n)]
--    ys = [y(1), y(2), ..., y(n)]
-- la ecuación de la recta de regresión de ys sobre xs es y = a+bx,
-- donde
--    b = (nΣx(i)y(i) - Σx(i)Σy(i)) / (nΣx(i)² - (Σx(i))²)
--    a = (Σy(i) - bΣx(i)) / n
--
-- Definir la función
--    regresionLineal :: [Double] -> [Double] -> (Double,Double)
-- tal que (regresionLineal xs ys) es el par (a,b) de los coeficientes
-- de la recta de regresión de ys sobre xs. Por ejemplo, para los
-- valores
--    ejX, ejY :: [Double]
--    ejX = [5,  7, 10, 12, 16, 20, 23, 27, 19, 14]
--    ejY = [9, 11, 15, 16, 20, 24, 27, 29, 22, 20]
-- se tiene
--    λ> regresionLineal ejX ejY
--    (5.195045748716805,0.9218924347243919)
-- ---------------------------------------------------------------------

ejX, ejY :: [Double]
ejX = [5,  7, 10, 12, 16, 20, 23, 27, 19, 14]
ejY = [9, 11, 15, 16, 20, 24, 27, 29, 22, 20]
```

```haskell
regresionLineal :: [Double] -> [Double] -> (Double,Double)
regresionLineal xs ys = (a,b)
    where n     = genericLength xs
          sumX  = sum xs
          sumY  = sum ys
          sumX2 = sum (zipWith (*) xs xs)
          sumXY = sum (zipWith (*) xs ys)
          b     = (n*sumXY - sumX*sumY) / (n*sumX2 - sumX^2)
          a     = (sumY - b*sumX) / n

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir el procedimiento
--    grafica :: [Double] -> [Double] -> IO ()
-- tal que (grafica xs ys) pinte los puntos correspondientes a las
-- listas de valores xs e ys y su recta de regresión. Por ejemplo,
-- con (grafica ejX ejY) se obtiene el dibujo de la Figura 1.
-- ---------------------------------------------------------------------

grafica :: [Double] -> [Double] -> IO ()
grafica xs ys =
    plotPathsStyle
      [YRange (0,10+mY)]
      [(defaultStyle {plotType = Points,
                      lineSpec = CustomStyle [LineTitle "Datos",
                                              PointType 2,
                                              PointSize 2.5]},
                     zip xs ys),
       (defaultStyle {plotType = Lines,
                      lineSpec = CustomStyle [LineTitle "Ajuste",
                                              LineWidth 2]},
                     [(x,a+b*x) | x <- [0..mX]])]
    where (a,b) = regresionLineal xs ys
          mX    = maximum xs
          mY    = maximum ys
```

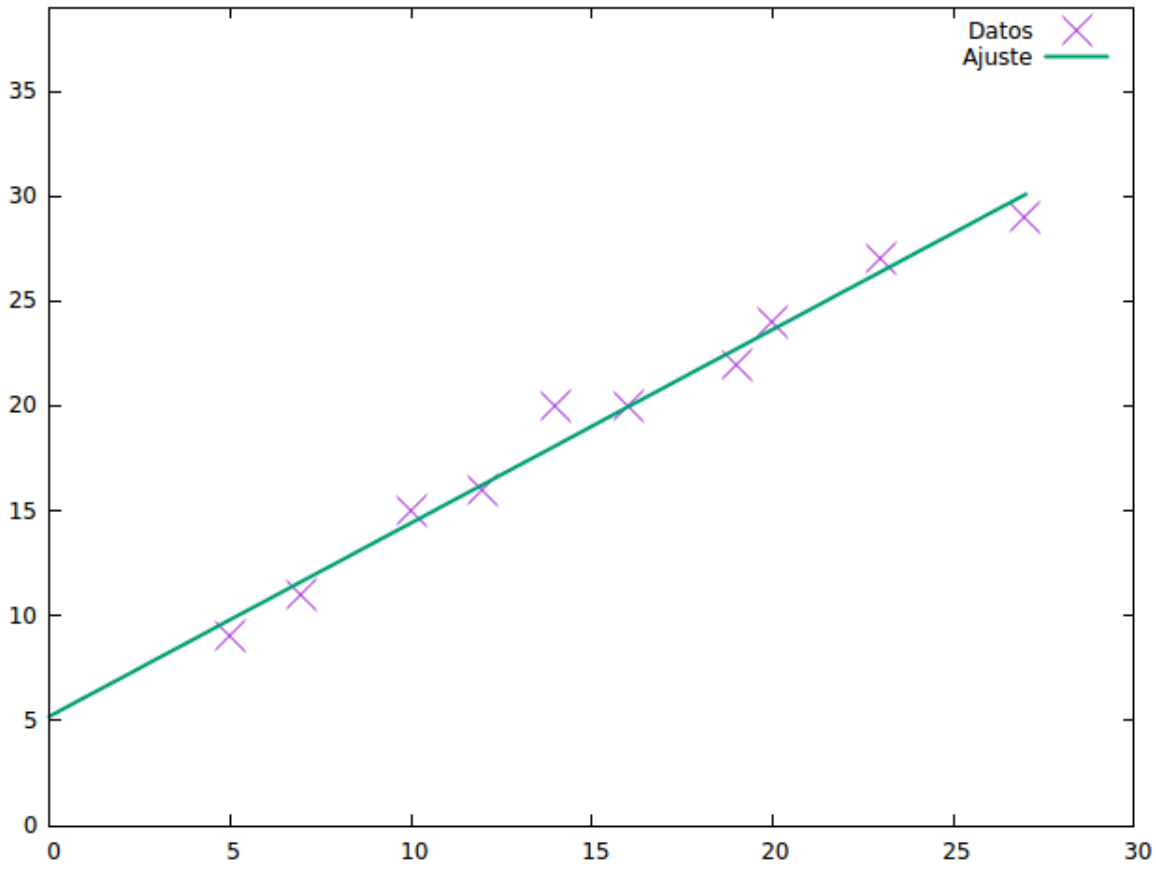


Figura 20.1: Regresión

# Relación 21

# Combinatoria

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es estudiar la generación y el número de
-- las principales operaciones de la combinatoria. En concreto, se
-- estudia
--    * Permutaciones.
--    * Combinaciones sin repetición.
--    * Combinaciones con repetición
--    * Variaciones sin repetición.
--    * Variaciones con repetición.
-- Como referencia se puede usar los apuntes de http://bit.ly/2HyyxAi

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

import Test.QuickCheck
import Data.List (genericLength)

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

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

-- 1ª definición
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs ]

-- 2ª definición
subconjunto2 :: Eq a => [a] -> [a] -> Bool
subconjunto2 []     _  = True
subconjunto2 (x:xs) ys = elem x ys && subconjunto2 xs ys

-- 3ª definición
subconjunto3 :: Eq a => [a] -> [a] -> Bool
subconjunto3 xs ys = foldr f True xs
  where f x z = x `elem` ys && z

-- La propiedad de equivalencia es
prop_equiv_subconjunto :: [Integer] -> [Integer] -> Bool
prop_equiv_subconjunto xs ys =
  subconjunto xs ys == subconjunto2 xs ys &&
  subconjunto xs ys == subconjunto3 xs ys

-- La comprobación es
--    ghci> quickCheck prop_equiv_subconjunto
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 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 3. Comprobar con QuickCheck que las funciones subconjunto
```

```haskell
-- y subconjunto' son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_equivalencia :: [Integer] -> [Integer] -> Bool
prop_equivalencia xs ys =
  subconjunto xs ys == subconjunto' xs ys

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. 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 5. Definir la función
--    subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de las subconjuntos de la lista
-- xs. Por ejemplo,
--    ghci> subconjuntos [2,3,4]
--    [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
--    ghci> subconjuntos [1,2,3,4]
--    [[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
```

```haskell
-- Cambiando la comprensión por map se obtiene
subconjuntos' :: [a] -> [[a]]
subconjuntos' []     = [[]]
subconjuntos' (x:xs) = sub ++ map (x:) sub
  where sub = subconjuntos' xs

-- ---------------------------------------------------------------------
-- § Permutaciones
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    intercala :: a -> [a] -> [[a]]
-- tal que (intercala x ys) es la lista de las listas obtenidas
-- intercalando x entre los elementos de ys. Por ejemplo,
--    intercala 1 [2,3]  ==  [[1,2,3],[2,1,3],[2,3,1]]
-- ---------------------------------------------------------------------

-- Una definición recursiva es
intercala1 :: a -> [a] -> [[a]]
intercala1 x []     = [[x]]
intercala1 x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala1 x ys]

-- Otra definición, más eficiente, es
intercala :: a -> [a] -> [[a]]
intercala y xs =
  [take n xs ++ (y : drop n xs) | n <- [0..length xs]]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    permutaciones :: [a] -> [[a]]
-- tal que (permutaciones xs) es la lista de las permutaciones de la
-- lista xs. Por ejemplo,
--    permutaciones "bc"   ==  ["bc","cb"]
--    permutaciones "abc"  ==  ["abc","bac","bca","acb","cab","cba"]
-- ---------------------------------------------------------------------

permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
```

```haskell
permutaciones (x:xs) =
  concat [intercala x ys | ys <- permutaciones xs]

-- 2ª definición
permutaciones2 :: [a] -> [[a]]
permutaciones2 []     = [[]]
permutaciones2 (x:xs) = concatMap (intercala x) (permutaciones2 xs)

-- 3ª definición
permutaciones3 :: [a] -> [[a]]
permutaciones3 = foldr (concatMap . intercala) [[]]

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    permutacionesN :: Integer -> [[Integer]]
-- tal que (permutacionesN n) es la lista de las permutaciones de los n
-- primeros números. Por ejemplo,
--    ghci> permutacionesN 3
--    [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
-- ---------------------------------------------------------------------

-- 1ª definición
permutacionesN :: Integer -> [[Integer]]
permutacionesN n = permutaciones [1..n]

-- 2ª definición
permutacionesN2 :: Integer -> [[Integer]]
permutacionesN2 = permutaciones . enumFromTo 1

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir, usando permutacionesN, la función
--    numeroPermutacionesN :: Integer -> Integer
-- tal que (numeroPermutacionesN n) es el número de permutaciones de un
-- conjunto con n elementos. Por ejemplo,
--    numeroPermutacionesN 3  ==  6
--    numeroPermutacionesN 4  ==  24
-- ---------------------------------------------------------------------

numeroPermutacionesN :: Integer -> Integer
numeroPermutacionesN = genericLength . permutacionesN
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    fact :: Integer -> Integer
-- tal que (fact n) es el factorial de n. Por ejemplo,
--    fact 3  ==  6
-- ---------------------------------------------------------------------

fact :: Integer -> Integer
fact n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir, usando fact, la función
--    numeroPermutacionesN' :: Integer -> Integer
-- tal que (numeroPermutacionesN' n) es el número de permutaciones de un
-- conjunto con n elementos. Por ejemplo,
--    numeroPermutacionesN' 3  ==  6
--    numeroPermutacionesN' 4  ==  24
-- ---------------------------------------------------------------------

numeroPermutacionesN' :: Integer -> Integer
numeroPermutacionesN' = fact

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    prop_numeroPermutacionesN :: Integer -> Bool
-- tal que (prop_numeroPermutacionesN n) se verifica si las funciones
-- numeroPermutacionesN y numeroPermutacionesN' son equivalentes para
-- los n primeros números. Por ejemplo,
--    prop_numeroPermutacionesN 5  ==  True
-- ---------------------------------------------------------------------

prop_numeroPermutacionesN :: Integer -> Bool
prop_numeroPermutacionesN n =
  and [numeroPermutacionesN x == numeroPermutacionesN' x | x <- [1..n]]

-- ---------------------------------------------------------------------
-- § Combinaciones
-- ---------------------------------------------------------------------
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    combinaciones :: Integer -> [a] -> [[a]]
-- tal que (combinaciones k xs) es la lista de las combinaciones de
-- orden k de los elementos de la lista xs. Por ejemplo,
--    ghci> combinaciones 2 "bcde"
--    ["bc","bd","be","cd","ce","de"]
--    ghci> combinaciones 3 "bcde"
--    ["bcd","bce","bde","cde"]
--    ghci> combinaciones 3 "abcde"
--    ["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
-- ---------------------------------------------------------------------

-- 1ª definición
combinaciones1 :: Integer -> [a] -> [[a]]
combinaciones1 n xs =
  [ys | ys <- subconjuntos xs, genericLength ys == n]

-- 2ª definición
combinaciones2 :: Integer -> [a] -> [[a]]
combinaciones2 0 _          = [[]]
combinaciones2 _ []         = []
combinaciones2 k (x:xs) =
  [x:ys | ys <- combinaciones2 (k-1) xs] ++ combinaciones2 k xs

-- La anterior definición se puede escribir usando map:
combinaciones3 :: Integer -> [a] -> [[a]]
combinaciones3 0 _      = [[]]
combinaciones3 _ []     = []
combinaciones3 k (x:xs) =
    map (x:) (combinaciones3 (k-1) xs) ++ combinaciones3 k xs

-- Nota. La segunda definición es más eficiente como se comprueba en la
-- siguiente sesión
--    ghci> :set +s
--    ghci> length (combinaciones1 2 [1..15])
--    105
--    (0.19 secs, 6373848 bytes)
--    ghci> length (combinaciones2 2 [1..15])
--    105
```

```
--    (0.01 secs, 525360 bytes)
--    ghci> length (combinaciones3 2 [1..15])
--    105
--    (0.02 secs, 528808 bytes)

-- En lo que sigue, usaremos combinaciones como combinaciones2
combinaciones :: Integer -> [a] -> [[a]]
combinaciones = combinaciones2

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    combinacionesN :: Integer -> Integer -> [[Integer]]
-- tal que (combinacionesN n k) es la lista de las combinaciones de
-- orden k de los n primeros números. Por ejemplo,
--    ghci> combinacionesN 4 2
--    [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
--    ghci> combinacionesN 4 3
--    [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
-- ---------------------------------------------------------------------

-- 1ª definición
combinacionesN :: Integer -> Integer -> [[Integer]]
combinacionesN n k = combinaciones k [1..n]

-- 2ª definición
combinacionesN2 :: Integer -> Integer -> [[Integer]]
combinacionesN2 = flip combinaciones . enumFromTo 1

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir, usando combinacionesN, la función
--    numeroCombinaciones :: Integer -> Integer -> Integer
-- tal que (numeroCombinaciones n k) es el número de combinaciones de
-- orden k de un conjunto con n elementos. Por ejemplo,
--    numeroCombinaciones 4 2  ==  6
--    numeroCombinaciones 4 3  ==  4
-- ---------------------------------------------------------------------

numeroCombinaciones :: Integer -> Integer -> Integer
numeroCombinaciones n k = genericLength (combinacionesN n k)
```

```haskell
-- Puede definirse por composición
numeroCombinaciones2 :: Integer -> Integer -> Integer
numeroCombinaciones2 = (genericLength .) . combinacionesN

-- Para facilitar la escritura de las definiciones por composición de
-- funciones con dos argumentos, se puede definir
(.:) :: (c -> d) -> (a -> b -> c) -> a -> b -> d
(.:) = (.) . (.)

-- con lo que la definición anterior se simplifica a
numeroCombinaciones3 :: Integer -> Integer -> Integer
numeroCombinaciones3 = genericLength .: combinacionesN

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    comb :: Integer -> Integer -> Integer
-- tal que (comb n k) es el número combinatorio n sobre k; es decir, .
--    (comb n k) = n! / (k!(n-k)!).
-- Por ejemplo,
--    comb 4 2  ==  6
--    comb 4 3  ==  4
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir, usando comb, la función
--    numeroCombinaciones' :: Integer -> Integer -> Integer
-- tal que (numeroCombinaciones' n k) es el número de combinaciones de
-- orden k de un conjunto con n elementos. Por ejemplo,
--    numeroCombinaciones' 4 2  ==  6
--    numeroCombinaciones' 4 3  ==  4
-- ---------------------------------------------------------------------

numeroCombinaciones' :: Integer -> Integer -> Integer
numeroCombinaciones' = comb

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
```

```haskell
--    prop_numeroCombinaciones :: Integer -> Bool
-- tal que (prop_numeroCombinaciones n) se verifica si las funciones
-- numeroCombinaciones y numeroCombinaciones' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--    prop_numeroCombinaciones 5  ==  True
-- ---------------------------------------------------------------------

prop_numeroCombinaciones :: Integer -> Bool
prop_numeroCombinaciones n =
  and [numeroCombinaciones n k == numeroCombinaciones' n k | k <- [1..n]]

-- ---------------------------------------------------------------------
-- § Combinaciones con repetición
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    combinacionesR :: Integer -> [a] -> [[a]]
-- tal que (combinacionesR k xs) es la lista de las combinaciones orden
-- k de los elementos de xs con repeticiones. Por ejemplo,
--    ghci> combinacionesR 2 "abc"
--    ["aa","ab","ac","bb","bc","cc"]
--    ghci> combinacionesR 3 "bc"
--    ["bbb","bbc","bcc","ccc"]
--    ghci> combinacionesR 3 "abc"
--    ["aaa","aab","aac","abb","abc","acc","bbb","bbc","bcc","ccc"]
-- ---------------------------------------------------------------------

combinacionesR :: Integer -> [a] -> [[a]]
combinacionesR _ [] = []
combinacionesR 0 _  = [[]]
combinacionesR k (x:xs) =
  [x:ys | ys <- combinacionesR (k-1) (x:xs)] ++ combinacionesR k xs

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    combinacionesRN :: Integer -> Integer -> [[Integer]]
-- tal que (combinacionesRN n k) es la lista de las combinaciones orden
-- k de los primeros n números naturales. Por ejemplo,
--    ghci> combinacionesRN 3 2
```

```
--    [[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]
--    ghci> combinacionesRN 2 3
--    [[1,1,1],[1,1,2],[1,2,2],[2,2,2]]
-- ---------------------------------------------------------------------

-- 1ª definición
combinacionesRN :: Integer -> Integer -> [[Integer]]
combinacionesRN n k = combinacionesR k [1..n]

-- 2ª definición
combinacionesRN2 :: Integer -> Integer -> [[Integer]]
combinacionesRN2 = flip combinacionesR . enumFromTo 1

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir, usando combinacionesRN, la función
--    numeroCombinacionesR :: Integer -> Integer -> Integer
-- tal que (numeroCombinacionesR n k) es el número de combinaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
--    numeroCombinacionesR 3 2  ==  6
--    numeroCombinacionesR 2 3  ==  4
-- ---------------------------------------------------------------------

numeroCombinacionesR :: Integer -> Integer -> Integer
numeroCombinacionesR n k = genericLength (combinacionesRN n k)

-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir, usando comb, la función
--    numeroCombinacionesR' :: Integer -> Integer -> Integer
-- tal que (numeroCombinacionesR' n k) es el número de combinaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
--    numeroCombinacionesR' 3 2  ==  6
--    numeroCombinacionesR' 2 3  ==  4
-- ---------------------------------------------------------------------

numeroCombinacionesR' :: Integer -> Integer -> Integer
numeroCombinacionesR' n k = comb (n+k-1) k

-- ---------------------------------------------------------------------
-- Ejercicio 23. Definir la función
--    prop_numeroCombinacionesR :: Integer -> Bool
```

```haskell
-- tal que (prop_numeroCombinacionesR n) se verifica si las funciones
-- numeroCombinacionesR y numeroCombinacionesR' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--    prop_numeroCombinacionesR 5  ==  True
-- ---------------------------------------------------------------------

prop_numeroCombinacionesR :: Integer -> Bool
prop_numeroCombinacionesR n =
  and [numeroCombinacionesR n k == numeroCombinacionesR' n k |
       k <- [1..n]]

-- ---------------------------------------------------------------------
-- § Variaciones
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 24. Definir la función
--    variaciones :: Integer -> [a] -> [[a]]
-- tal que (variaciones n xs) es la lista de las variaciones n-arias
-- de la lista xs. Por ejemplo,
--    variaciones 2 "abc"  ==  ["ab","ba","ac","ca","bc","cb"]
-- ---------------------------------------------------------------------

variaciones :: Integer -> [a] -> [[a]]
variaciones k xs = concatMap permutaciones (combinaciones k xs)

-- ---------------------------------------------------------------------
-- Ejercicio 25. Definir la función
--    variacionesN :: Integer -> Integer -> [[Integer]]
-- tal que (variacionesN n k) es la lista de las variaciones de orden k
-- de los n primeros números. Por ejemplo,
--    variacionesN 3 2  ==  [[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]]
-- ---------------------------------------------------------------------

variacionesN :: Integer -> Integer -> [[Integer]]
variacionesN n k = variaciones k [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 26. Definir, usando variacionesN, la función
--    numeroVariaciones :: Integer -> Integer -> Integer
```

```haskell
-- tal que (numeroVariaciones n k) es el número de variaciones de orden
-- k de un conjunto con n elementos. Por ejemplo,
--    numeroVariaciones 4 2  ==  12
--    numeroVariaciones 4 3  ==  24
-- ---------------------------------------------------------------------

-- 1ª definición
numeroVariaciones :: Integer -> Integer -> Integer
numeroVariaciones n k = genericLength (variacionesN n k)

-- 2ª definición
numeroVariaciones2 :: Integer -> Integer -> Integer
numeroVariaciones2 = (genericLength .) . variacionesN

-- ---------------------------------------------------------------------
-- Ejercicio 27. Definir, usando product, la función
--    numeroVariaciones' :: Integer -> Integer -> Integer
-- tal que (numeroVariaciones' n k) es el número de variaciones de orden
-- k de un conjunto con n elementos. Por ejemplo,
--    numeroVariaciones' 4 2  ==  12
--    numeroVariaciones' 4 3  ==  24
-- ---------------------------------------------------------------------

numeroVariaciones' :: Integer -> Integer -> Integer
numeroVariaciones' n k = product [n-k+1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 28. Definir la función
--    prop_numeroVariaciones :: Integer -> Bool
-- tal que (prop_numeroVariaciones n) se verifica si las funciones
-- numeroVariaciones y numeroVariaciones' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--    prop_numeroVariaciones 5  ==  True
-- ---------------------------------------------------------------------

prop_numeroVariaciones :: Integer -> Bool
prop_numeroVariaciones n =
  and [numeroVariaciones n k == numeroVariaciones' n k | k <- [1..n]]

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

```haskell
-- § Variaciones con repetición
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 28. Definir la función
--    variacionesR :: Integer -> [a] -> [[a]]
-- tal que (variacionesR k xs) es la lista de las variaciones de orden
-- k de los elementos de xs con repeticiones. Por ejemplo,
--    ghci> variacionesR 1 "ab"
--    ["a","b"]
--    ghci> variacionesR 2 "ab"
--    ["aa","ab","ba","bb"]
--    ghci> variacionesR 3 "ab"
--    ["aaa","aab","aba","abb","baa","bab","bba","bbb"]
-- ---------------------------------------------------------------------

variacionesR :: Integer -> [a] -> [[a]]
variacionesR 0 _  = [[]]
variacionesR k xs =
  [z:ys | z <- xs, ys <- variacionesR (k-1) xs]

-- ---------------------------------------------------------------------
-- Ejercicio 30. Definir la función
--    variacionesRN :: Integer -> Integer -> [[Integer]]
-- tal que (variacionesRN n k) es la lista de las variaciones orden
-- k de los primeros n números naturales. Por ejemplo,
--    ghci> variacionesRN 3 2
--    [[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
--    ghci> variacionesRN 2 3
--    [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
-- ---------------------------------------------------------------------

variacionesRN :: Integer -> Integer -> [[Integer]]
variacionesRN n k = variacionesR k [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 31. Definir, usando variacionesR, la función
--    numeroVariacionesR :: Integer -> Integer -> Integer
-- tal que (numeroVariacionesR n k) es el número de variaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
```

```
--    numeroVariacionesR 3 2  ==  9
--    numeroVariacionesR 2 3  ==  8
-- ---------------------------------------------------------------------

numeroVariacionesR :: Integer -> Integer -> Integer
numeroVariacionesR n k = genericLength (variacionesRN n k)

-- ---------------------------------------------------------------------
-- Ejercicio 32. Definir, usando (^), la función
--    numeroVariacionesR' :: Integer -> Integer -> Integer
-- tal que (numeroVariacionesR' n k) es el número de variaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
--    numeroVariacionesR' 3 2  ==  9
--    numeroVariacionesR' 2 3  ==  8
-- ---------------------------------------------------------------------

numeroVariacionesR' :: Integer -> Integer -> Integer
numeroVariacionesR' n k = n^k

-- ---------------------------------------------------------------------
-- Ejercicio 33. Definir la función
--    prop_numeroVariacionesR :: Integer -> Bool
-- tal que (prop_numeroVariacionesR n) se verifica si las funciones
-- numeroVariacionesR y numeroVariacionesR' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--    prop_numeroVariacionesR 5  ==  True
-- ---------------------------------------------------------------------

prop_numeroVariacionesR :: Integer -> Bool
prop_numeroVariacionesR n =
  and [numeroVariacionesR n k == numeroVariacionesR' n k |
       k <- [1..n]]
```

# Relación 22

# Cálculo del número pi mediante el método de Montecarlo

```
-- ---------------------------------------------------------------------
-- § Introducción                                                      --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es el uso de los números
-- aleatorios para calcular el número pi mediante el método de
-- Montecarlo. Un ejemplo del método se puede leer en el artículo de
-- Pablo Rodríguez "Calculando pi con gotas de lluvia" que se encuentra
-- en http://bit.ly/1cNfSR0

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import System.Random
import Graphics.Gnuplot.Simple

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    puntosDelCuadrado :: Int -> IO [(Double,Double)]
-- tal que (puntosDelCuadrado n) es una lista aleatoria de n puntos del
-- cuadrado de vértices opuestos (-1,-1) y (1,1). Por ejemplo,
--    λ> puntosDelCuadrado 2
--    [(0.7071212580055017,0.5333728820632873),
--     (-0.18430740317151528,-0.9996319726105287)]
--    λ> puntosDelCuadrado 2
```

```
--    [(-0.45032646341358595,0.30614607738929633),
--     (0.4402992058238284,0.5810531167431172)]
-- ---------------------------------------------------------------------

puntosDelCuadrado :: Int -> IO [(Double,Double)]
puntosDelCuadrado n = do
  gen <- newStdGen
  let xs = randomRs (-1,1) gen
      (as, ys) = splitAt n xs
      (bs, _)  = splitAt n ys
  return (zip as bs)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    puntosEnElCirculo :: [(Double,Double)] -> Int
-- tal que (puntosEnElCirculo xs) es el número de puntos de la lista xs
-- que están en el círculo de centro (0,0) y radio 1.
--    puntosEnElCirculo [(1,0), (0.5,0.9), (0.2,-0.3)]  ==  2
-- ---------------------------------------------------------------------

puntosEnElCirculo :: [(Double,Double)] -> Int
puntosEnElCirculo xs =
  length [(x,y) | (x,y) <- xs
                , x^2+y^2 <= 1]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    calculoDePi :: Int -> Double
-- tal que (calculoDePi n) es el cálculo del número pi usando n puntos
-- aleatorios (la probabilidad de que estén en el círculo es pi/4). Por
-- ejemplo,
--    λ> calculoDePi 1000
--    3.088
--    λ> calculoDePi 1000
--    3.184
--    λ> calculoDePi 10000
--    3.1356
--    λ> calculoDePi 100000
--    3.13348
-- ---------------------------------------------------------------------
```

```haskell
calculoDePi :: Int -> IO Double
calculoDePi n = do
  xs <- puntosDelCuadrado n
  let enCirculo = fromIntegral (puntosEnElCirculo xs)
      total     = fromIntegral n
  return (4 * enCirculo / total)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    graficaPi :: [Int] -> IO ()
-- tal que (graficaPi xs) dibuja la grafica del valor de pi usando el
-- número de putos indicados por los elementos de xs. Por ejemplo,
-- (graficaPi [0,10..4000]) dibuja la Figura 1.
-- ---------------------------------------------------------------------

graficaPi :: [Int] -> IO ()
graficaPi xs = do
  ys <- mapM calculoDePi xs
  plotLists [Key Nothing]
            [ zip xs ys
            , zip xs (repeat pi) ]
```

# Relación 23

# Ecuaciones con factoriales

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es resolver la ecuación
--    a! * b! = a! + b! + c!
-- donde a, b y c son números naturales.

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    factorial :: Integer -> Integer
-- tal que (factorial n) es el factorial de n. Por ejemplo,
--    factorial 5  ==  120
-- ---------------------------------------------------------------------

-- 1ª definición
factorial1 :: Integer -> Integer
factorial1 n = product [1..n]

-- 2ª definición
factorial2 :: Integer -> Integer
```

```haskell
factorial2 n = foldl' (*) 1 [1..n]

-- Comparación de eficiencia
--    λ> length (show (factoria1 (10^5)))
--    456574
--    λ> length (show (factorial2 (10^5)))
--    456574
--    (1.33 secs, 11,315,759,904 bytes)

-- En lo que sigue, usaremos la 2ª
factorial :: Integer -> Integer
factorial = factorial2

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

-- 1ª definición
factoriales1 :: [Integer]
factoriales1 = map factorial [0..]

-- 2ª definición
factoriales2 :: [Integer]
factoriales2 = 1 : scanl1 (*) [1..]

-- 3ª definición
factoriales3 :: [Integer]
factoriales3 = scanl' (*) 1 [1..]

-- Comparación de eficiencia
--    λ> length (show (factoriales1 !! (5*10^4)))
--    213237
--    (0.33 secs, 2,618,502,664 bytes)
--    λ> length (show (factoriales2 !! (5*10^4)))
--    213237
--    (1.00 secs, 2,617,341,392 bytes)
```

```
--    λ> length (show (factoriales3 !! (5*10^4)))
--    213237
--    (1.19 secs, 2,613,701,520 bytes)

-- Usaremos la 1ª
factoriales :: [Integer]
factoriales = factoriales1

-- ---------------------------------------------------------------------
-- 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,
--    ghci> take 7 posicionesFactoriales
--    [(0,1),(1,1),(2,2),(3,6),(4,24),(5,120),(6,720)]
-- ---------------------------------------------------------------------

posicionesFactoriales :: [(Integer,Integer)]
posicionesFactoriales = zip [0..] factoriales

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    invFactorial :: Integer -> Maybe Integer
-- tal que (invFactorial x) es (Just n) si el factorial de n es x y es
-- Nothing, en caso contrario. Por ejemplo,
--    invFactorial 120  == Just 5
--    invFactorial 20   == Nothing
-- ---------------------------------------------------------------------
```

```haskell
invFactorial :: Integer -> Maybe Integer
invFactorial x
  | esFactorial x = Just (head [n | (n,y) <- posicionesFactoriales
                                  , y == x])
  | otherwise     = Nothing

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la constante
--    pares :: [(Integer,Integer)]
-- tal que pares es la lista de todos los pares de números naturales. Por
-- ejemplo,
--    ghci> 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
--    ghci> solucionFactoriales
--    (3,3,4)

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

```haskell
-- Ejercicio 8. Comprobar con QuickCheck que solucionFactoriales es la
-- única solución de la ecuación
--    a! * b! = a! + b! + c!
-- con a, b y c números naturales
-- ------------------------------------------------------------------

prop_solucionFactoriales :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales x y z =
    x >= 0 && y >= 0 && z >= 0 && (x,y,z) /= solucionFactoriales
    ==> (f x * f y /= f x + f y + f z)
    where f = factorial

-- La comprobación es
--    ghci> quickCheck prop_solucionFactoriales
--    *** Gave up! Passed only 86 tests.

-- También se puede expresar como
prop_solucionFactoriales' :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales' x y z =
    x >= 0 && y >= 0 && z >= 0 &&
    f x * f y == f x + f y + f z
    ==> (x,y,z) == solucionFactoriales
    where f = factorial

-- La comprobación es
--    ghci> quickCheck prop_solucionFactoriales
--    *** Gave up! Passed only 0 tests.

-- ------------------------------------------------------------------
-- Nota: El ejercicio se basa en el artículo "Ecuación con factoriales"
-- del blog Gaussianos publicado en
--    http://gaussianos.com/ecuacion-con-factoriales
-- ------------------------------------------------------------------
```

# Relación 24

# Algoritmos de ordenación y complejidad

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es presentar una recopilación de los
-- algoritmos de ordenación y el estudio de su complejidad.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- § Ordenación por selección                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación por selección se selecciona el menor elemento de xs y se
-- le añade a la ordenación por selección de los restantes. Por ejemplo,
-- para ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
--       ordenaPorSeleccion [3,1,4,1,5,9,2]
--     = 1 : ordenaPorSeleccion [3,4,1,5,9,2]
--     = 1 : 1 : ordenaPorSeleccion [3,4,5,9,2]
--     = 1 : 1 : 2 : ordenaPorSeleccion [3,4,5,9]
```

```
--     = 1 : 1 : 2 : 3 : ordenaPorSeleccion [4,5,9]
--     = 1 : 1 : 2 : 3 : 4 : ordenaPorSeleccion [5,9]
--     = 1 : 1 : 2 : 3 : 4 : 5 : ordenaPorSeleccion [9]
--     = 1 : 1 : 2 : 3 : 4 : 5 : 9 : ordenaPorSeleccion []
--     = 1 : 1 : 2 : 3 : 4 : 5 : 9 : []
--     = [1,1,2,3,4,5,9]
--
-- Definir la función
--    ordenaPorSeleccion :: Ord a => [a] -> [a]
-- tal que (ordenaPorSeleccion xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--    ordenaPorSeleccion [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaPorSeleccion :: Ord a => [a] -> [a]
ordenaPorSeleccion [] = []
ordenaPorSeleccion xs = m : ordenaPorSeleccion (delete m xs)
  where m = minimum xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaPorSeleccion [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000].
--
-- ¿Cuál es el orden de complejidad de ordenaPorSeleccion?
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+-----
--    1000 | 0.05
--    2000 | 0.25
--    3000 | 0.58
--    4000 | 1.13

-- La complejidad de ordenaPorSeleccion es O(n^2).
--
-- Las ecuaciones de recurrencia del coste de ordenaPorSeleccion son
--    T(0) = 1
--    T(n) = 1 + T(n-1) + 2n
```

```haskell
-- Luego, T(n) = (n+1)^2 (ver http://bit.ly/1DGsMeW )

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    ordenaPorSeleccion2 :: Ord a => [a] -> [a]
-- tal que (ordenaPorSeleccion2 xs) es la lista xs ordenada por el
-- algoritmo de selección, pero usando un acumulador. Por ejemplo,
--    ordenaPorSeleccion2 [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaPorSeleccion2 :: Ord a => [a] -> [a]
ordenaPorSeleccion2 [] = []
ordenaPorSeleccion2 (x:xs) = aux xs x []
  where aux [] m r = m : ordenaPorSeleccion2 r
        aux (y:ys) m r | y < m     = aux ys y (m:r)
                       | otherwise = aux ys m (y:r)

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Calcular los tiempos necesarios para calcular
--    let n = k in length (ordenaPorSeleccion2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+-----
--    1000 | 0.39
--    2000 | 1.53
--    3000 | 3.48
--    4000 | 6.35

-- ---------------------------------------------------------------------
-- § Ordenación rápida (Quicksort)                                     --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación rápida se selecciona el primer elemento x de xs, se divide
-- los restantes en los menores o iguales que x y en los mayores que x,
-- se ordena cada una de las dos partes y se unen los resultados. Por
```

```haskell
-- ejemplo, para ordenar la lista [3,1,4,1,5,9,2] el proceso es el
-- siguiente:
--       or [3,1,4,1,5,9,2]
--     = or [1,1,2] ++ [3] ++ or [4,5,9]
--     = (or [1] ++ [1] ++ or [2]) ++ [3] ++ (or [] ++ [4] ++ or [5,9])
--     = ((or [] ++ [1] ++ or []) ++ [1] ++ (or [] ++ [2] ++ or []))
--       ++ [3] ++ ([] ++ [4] ++ (or [] ++ [5] ++ or [9]))
--     = (([] ++ [1] ++ []) ++ [1] ++ ([] ++ [2] ++ []))
--       ++ [3] ++ ([4] ++ ([] ++ [5] ++ (or [] ++ [9] ++ or [])))
--     = ([1] ++ [1] ++ [2] ++
--       ++ [3] ++ ([4] ++ ([5] ++ (or [] ++ [9] ++ or [])))
--     = ([1] ++ [1] ++ [2] ++
--       ++ [3] ++ ([4] ++ ([5] ++ ([] ++ [9] ++ [])))
--     = ([1] ++ [1] ++ [2] ++
--       ++ [3] ++ ([4] ++ ([5] ++ [9]))
--     = [1,1,2,3,4,5,9]
--
-- Definir la función
--    ordenaRapida :: Ord a => [a] -> [a]
-- tal que (ordenaRapida xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--    ordenaRapida [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaRapida :: Ord a => [a] -> [a]
ordenaRapida [] = []
ordenaRapida (x:xs) =
  ordenaRapida menores ++ [x] ++ ordenaRapida mayores
  where menores = [y | y <- xs, y <= x]
        mayores = [y | y <- xs, y >  x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaRapida [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
--
-- ¿Cuál es el orden de complejidad de ordenaRapida?
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
```

```
--    k    | segs.
--    -----+------
--    1000 |  0.64
--    2000 |  2.57
--    3000 |  6.64
--    4000 | 12.33

-- La complejidad de ordenaRapida es O(n log(n)).

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir, usando un acumulador, la función
--    ordenaRapida2 :: Ord a => [a] -> [a]
-- tal que (ordenaRapida2 xs) es la lista obtenida ordenando xs
-- por el procedimiento de ordenación rápida. Por ejemplo,
--    ordenaRapida2 [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaRapida2 :: Ord a => [a] -> [a]
ordenaRapida2 xs = aux xs []
  where aux [] s     = s
        aux (x:ys) s = aux menores (x : aux mayores s)
          where menores = [y | y <- ys, y <= x]
                mayores = [y | y <- ys, y >  x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaRapida2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+------
--    1000 |  0.56
--    2000 |  2.42
--    3000 |  5.87
--    4000 | 10.93

-- ---------------------------------------------------------------------
-- § Ordenación por inserción                                        --
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación por inserción se selecciona el primer elemento x de xs, se
-- ordena el resto de xs y se inserta x en su lugar. Por ejemplo, para
-- ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
--      ordenaPorInsercion [3,1,4,1,5,9,2]
--    = 3 : ordenaPorInsercion [1,4,1,5,9,2]
--    = 3 : 1 : ordenaPorInsercion [4,1,5,9,2]
--    = 3 : 1 : 4 : ordenaPorInsercion [1,5,9,2]
--    = 3 : 1 : 4 : 1 : ordenaPorInsercion [5,9,2]
--    = 3 : 1 : 4 : 1 : 5 : ordenaPorInsercion [9,2]
--    = 3 : 1 : 4 : 1 : 5 : 9 : ordenaPorInsercion [2]
--    = 3 : 1 : 4 : 1 : 5 : 9 : 2 : ordenaPorInsercion []
--    = 3 : 1 : 4 : 1 : 5 : 9 : 2 : []
--    = 3 : 1 : 4 : 1 : 5 : 9 : [2]
--    = 3 : 1 : 4 : 1 : 5 : [2,9]
--    = 3 : 1 : 4 : 1 : [2,5,9]
--    = 3 : 1 : 4 : [1,2,5,9]
--    = 3 : 1 : [1,2,4,5,9]
--    = 3 : [1,1,2,4,5,9]
--    = [1,1,2,3,4,5,9]
--
-- Definir la función
--    ordenaPorInsercion :: Ord a => [a] -> [a]
-- tal que (ordenaPorInsercion xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--    ordenaPorInsercion [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaPorInsercion :: Ord a => [a] -> [a]
ordenaPorInsercion []     = []
ordenaPorInsercion (x:xs) = inserta x (ordenaPorInsercion xs)

-- (inserta x xs) inserta el elemento x después de los elementos de xs
-- que son menores o iguales que x. Por ejemplo,
--    inserta 5 [3,2,6,4]  ==  [3,2,5,6,4]
inserta :: Ord a => a -> [a] -> [a]
inserta y []                 = [y]
```

```haskell
inserta y l@(x:xs) | y <= x    = y : l
                   | otherwise = x : inserta y xs

-- 2ª definición de inserta:
inserta2 :: Ord a => a -> [a] -> [a]
inserta2 x xs = takeWhile (<= x) xs ++ [x] ++ dropWhile (<=x) xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaPorInsercion [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
--
-- ¿Cuál es la complejidad de ordenaPorInsercion?
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+-----
--    1000 | 0.39
--    2000 | 1.53
--    3000 | 3.49
--    4000 | 6.32

-- La complejidad de ordenaPorInsercion es O(n^2)
--
-- Las ecuaciones de recurrencia del coste de ordenaPorInsercion son
--    T(0) = 1
--    T(n) = n + T(n-1)
-- Luego, T(n) = 2n(n+1)+1 (ver https://bit.ly/2WWMP85 )

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, por plegados, la función
--    ordenaPorInsercion2 :: Ord a => [a] -> [a]
-- tal que (ordenaPorInsercion2 xs) es la lista obtenida ordenando xs
-- por el procedimiento de ordenación por inserción. Por ejemplo,
--    ordenaPorInsercion2 [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaPorInsercion2 :: Ord a => [a] -> [a]
ordenaPorInsercion2 = foldr inserta []
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaPorInsercion2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+------
--    1000 | 0.38
--    2000 | 1.54
--    3000 | 3.46
--    4000 | 6.29

-- ---------------------------------------------------------------------
-- § Ordenación por mezcla ("Mergesort")                               --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación por mezcla se divide xs por la mitad, se ordena cada una
-- de las partes y se mezclan los resultados. Por ejemplo, para
-- ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
--      om [3,1,4,1,5,9,2]
--    = m (om [3,1,4]) (om 1,5,9,2])
--    = m (m (om [3]) (om [1,4])) (m (om [1,5]) (om [9,2]))
--    = m (m [3] (m (om [1]) (om [4])))
--        (m (m (om [1]) (om [5])) (m (om [9]) (om [2])))
--    = m (m [3] (m [1] [4]))
--        (m (m [1] [5]) (m [9] [2]))
--    = m (m [3] [1,4]) (m [1,5] [2,9])
--    = m [1,3,4] [1,2,5,9]
--    = [1,1,2,3,4,5,9]
-- donde om es ordenaPorMezcla y m es mezcla.
--
-- Definir la función
--    ordenaPorMezcla :: Ord a => [a] -> [a]
-- tal que (ordenaPorMezcla xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
```

```haskell
--    ordenaPorMezcla [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ----------------------------------------------------------------------

ordenaPorMezcla :: Ord a => [a] -> [a]
ordenaPorMezcla []  = []
ordenaPorMezcla [x] = [x]
ordenaPorMezcla l   = mezcla (ordenaPorMezcla l1) (ordenaPorMezcla l2)
    where l1 = take k l
          l2 = drop k l
          k  = length l `div` 2

-- (mezcla xs ys) es la lista obtenida mezclando xs e ys. Por ejemplo,
--    mezcla [1,3] [2,4,6]  ==  [1,2,3,4,6]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] b = b
mezcla a [] = a
mezcla a@(x:xs) b@(y:ys) | x <= y    = x : mezcla xs b
                         | otherwise = y : mezcla a ys

-- ----------------------------------------------------------------------
-- Ejercicio 4.2. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaPorMezcla [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
--
-- ¿Cuál es la complejidad de ordenaPorMezcla?
-- ----------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+-----
--    1000 | 0.02
--    2000 | 0.03
--    3000 | 0.05
--    4000 | 0.06

-- La complejidad de ordenaPorMezcla es O(n log(n)).
--
-- Las ecuaciones de recurrencia del coste de ordenaPorMezcla son
--    T(0) = 1
--    T(1) = 1
```

```
--    T(n) = n + 2*T(n/2)
-- Luego, T(n) = (c*n)/2+(n log(n))/(log(2)) (ver http://bit.ly/1EyUTYG )

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Otra forma de ordenar una lista xs mediante el
-- algoritmo de ordenación por mezcla consiste en dividir xs en listas
-- unitarias y mezclar los resultados. Por ejemplo, para
-- ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
--      om [3,1,4,1,5,9,2]
--    = mp [[3],[1],[4],[1],[5],[9],[2]]
--    = mp [[1,3],[1,4],[5,9],[2]]
--    = mp [[1,1,3,4],[2,5,9]]
--    = [1,1,2,3,4,5,9]
-- donde om es ordenaPorMezcla y mp es mezclaPares.
--
-- Definir la función
--    ordenaPorMezcla2 :: Ord a => [a] -> [a]
-- tal que (ordenaPorMezcla2 xs) es la lista obtenida ordenando por
-- mezcla la lista xs. Por ejemplo,
--    ordenaPorMezcla2 [3,1,4,1,5,9,2]  ==  [1,1,2,3,4,5,9]
-- ---------------------------------------------------------------------

ordenaPorMezcla2 :: Ord a => [a] -> [a]
ordenaPorMezcla2 xs = aux (divide xs)
  where aux [r] = r
        aux ys  = aux (mezclaPares ys)

-- (divide xs) es la lista de de las listas unitarias formadas por los
-- elementos de xs. Por ejemplo,
--    divide [3,1,4,1,5,9,2,8]  ==  [[3],[1],[4],[1],[5],[9],[2],[8]]
divide :: Ord a => [a] -> [[a]]
divide xs = [[x] | x <- xs]

-- También se puede definir por recursión
divide2 :: Ord a => [a] -> [[a]]
divide2 []     = []
divide2 (x:xs) = [x] : divide2 xs

-- (mezclaPares xs) es la lista obtenida mezclando los pares de
-- elementos consecutivos de xs. Por ejemplo,
```

```
--    ghci> mezclaPares [[3],[1],[4],[1],[5],[9],[2],[8]]
--    [[1,3],[1,4],[5,9],[2,8]]
--    ghci> mezclaPares [[1,3],[1,4],[5,9],[2,8]]
--    [[1,1,3,4],[2,5,8,9]]
--    ghci> mezclaPares [[1,1,3,4],[2,5,8,9]]
--    [[1,1,2,3,4,5,8,9]]
--    ghci> mezclaPares [[1],[3],[2]]
--    [[1,3],[2]]
mezclaPares :: (Ord a) => [[a]] -> [[a]]
mezclaPares []           = []
mezclaPares [x]          = [x]
mezclaPares (xs:ys:zss)  = mezcla xs ys : mezclaPares zss

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Calcular los tiempos necesarios para calcular
--     let n = k in length (ordenaPorMezcla2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-- ---------------------------------------------------------------------

-- El resumen de los tiempos es
--    k    | segs.
--    -----+-----
--    1000 | 0.02
--    2000 | 0.03
--    3000 | 0.03
--    4000 | 0.05

-- ---------------------------------------------------------------------
-- § Comparaciones con listas aleatorias                              --
-- ---------------------------------------------------------------------

--    λ> import System.Random (randomRIO)
--    λ> import Control.Monad (replicateM)
--    λ> ej10000 <- replicateM 10000 (randomRIO (0,10000))
--    λ> :set +s
--    λ> maximum (ordenaPorSeleccion ej10000)
--    9998
--    (2.69 secs, 6,757,883,928 bytes)
--    λ> maximum (ordenaRapida ej10000)
--    9998
```

```
--    (0.11 secs, 40,701,576 bytes)
--    λ> maximum (ordenaPorInsercion ej10000)
--    9998
--    (6.57 secs, 6,305,208,920 bytes)
--    λ> maximum (ordenaPorMezcla ej10000)
--    9998
--    (0.09 secs, 36,797,672 bytes)
--    λ> ej20000 <- replicateM 20000 (randomRIO (0,20000))
--    (0.01 secs, 11,782,488 bytes)
--    λ> maximum (ordenaRapida ej20000)
--    20000
--    (0.18 secs, 86,766,376 bytes)
--    λ> maximum (ordenaPorMezcla ej20000)
--    20000
--    (0.14 secs, 78,188,176 bytes)
--    λ> ej50000 <- replicateM 50000 (randomRIO (0,50000))
--    (0.03 secs, 28,855,672 bytes)
--    λ> maximum (ordenaRapida ej50000)
--    50000
--    (0.45 secs, 240,099,608 bytes)
--    λ> maximum (ordenaPorMezcla ej50000)
--    50000
--    (0.38 secs, 211,648,672 bytes)
```

# Relación 25

# El TAD de las pilas

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

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las pilas, utilizando las implementaciones estudiadas en el
-- tema 14 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-14.html
--
-- Para realizar los ejercicios hay que descargar (en el mismo
-- directorio donde está el ejercicio) las implementaciones de las
-- implementaciones de las pilas:
-- + PilaConTipoDeDatoAlgebraico.hs que está en http://bit.ly/2Of5PbG
-- + PilaConListas.hs               que está en http://bit.ly/2OihhDw
--
-- Además, hay que instalar las librerías de I1M que se encuentra en
-- http://hackage.haskell.org/package/I1M
--
-- Para instalar la librería de I1M, basta ejecutar en una consola
--    cabal update
--    cabal install I1M

{-# OPTIONS_GHC -fno-warn-unused-matches
                -fno-warn-unused-imports
                -fno-warn-orphans
#-}

module Rel_25_sol where

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

```haskell
-- Introducción                                                --
-- ------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las pilas, utilizando las implementaciones estudiadas en el
-- tema 14 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-14.html
--
-- Para realizar los ejercicios hay que descargar (en el mismo
-- directorio donde está el ejercicio) las implementaciones de las
-- implementaciones de las pilas:
-- + PilaConTipoDeDatoAlgebraico.hs que está en http://bit.ly/2Of5PbG
-- + PilaConListas.hs               que está en http://bit.ly/2OihhDw

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

import Data.List
import Test.QuickCheck

-- Hay que elegir una implementación del TAD pilas.
-- import PilaConTipoDeDatoAlgebraico
-- import PilaConListas
import I1M.Pila

-- ------------------------------------------------------------
-- Ejemplos
-- ------------------------------------------------------------

-- A lo largo de esta relación de ejercicios usaremos los siguientes
-- ejemplos de pila
ejP1, ejP2, ejP3, ejP4, ejP5 :: Pila Int
ejP1 = foldr apila vacia [1..20]
ejP2 = foldr apila vacia [2,5..18]
ejP3 = foldr apila vacia [3..10]
ejP4 = foldr apila vacia [4,-1,7,3,8,10,0,3,3,4]
ejP5 = foldr apila vacia [1..5]

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

```
-- Ejercicio 1: Definir la función
--    filtraPila :: (a -> Bool) -> Pila a -> Pila a
-- tal que (filtraPila p q) es la pila obtenida con los elementos de
-- pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
--    ghci> ejP1
--    1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|-
--    ghci> filtraPila even ejP1
--    2|4|6|8|10|12|14|16|18|20|-
-- ---------------------------------------------------------------------

filtraPila :: (a -> Bool) -> Pila a -> Pila a
filtraPila p q
  | esVacia q = vacia
  | p cq      = apila cq (filtraPila p dq)
  | otherwise = filtraPila p dq
  where cq = cima q
        dq = desapila q

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    mapPila :: (a -> a) -> Pila a -> Pila a
-- tal que (mapPila f p) es la pila formada con las imágenes por f de
-- los elementos de pila p, en el mismo orden. Por ejemplo,
--    ghci> mapPila (+7) ejP1
--    8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|-
-- ---------------------------------------------------------------------

mapPila :: (a -> a) -> Pila a -> Pila a
mapPila f p
  | esVacia p = p
  | otherwise = apila (f cp) (mapPila f dp)
  where cp = cima p
        dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 3: Definir la función
--    pertenecePila :: Eq a => a -> Pila a -> Bool
-- tal que (pertenecePila y p) se verifica si y es un elemento de la
-- pila p. Por ejemplo,
--    pertenecePila 7 ejP1  == True
```

```haskell
--    perteneciaPila 70 ejP1 == False
-- ---------------------------------------------------------------------

perteneciaPila :: Eq a => a -> Pila a -> Bool
perteneciaPila x p
  | esVacia p  = False
  | otherwise  = x == cp || perteneciaPila x dp
  where cp = cima p
        dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 4: Definir la función
--    contenidaPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (contenidaPila p1 p2) se verifica si todos los elementos de
-- de la pila p1 son elementos de la pila p2. Por ejemplo,
--    contenidaPila ejP2 ejP1  == True
--    contenidaPila ejP1 ejP2  == False
-- ---------------------------------------------------------------------

contenidaPila :: Eq a => Pila a -> Pila a -> Bool
contenidaPila p1 p2
  | esVacia p1 = True
  | otherwise  = perteneciaPila cp1 p2 && contenidaPila dp1 p2
  where cp1 = cima p1
        dp1 = desapila p1

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    prefijoPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (prefijoPila p1 p2) se verifica si la pila p1 es justamente
-- un prefijo de la pila p2. Por ejemplo,
--    prefijoPila ejP3 ejP2 == False
--    prefijoPila ejP5 ejP1 == True
-- ---------------------------------------------------------------------

prefijoPila :: Eq a => Pila a -> Pila a -> Bool
prefijoPila p1 p2
    | esVacia p1 = True
    | esVacia p2 = False
    | otherwise  = cp1 == cp2 && prefijoPila dp1 dp2
```

```
    where cp1 = cima p1
          dp1 = desapila p1
          cp2 = cima p2
          dp2 = desapila p2

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    subPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (subPila p1 p2) se verifica si p1 es una subpila de p2. Por
-- ejemplo,
--    subPila ejP2 ejP1 == False
--    subPila ejP3 ejP1 == True
-- ---------------------------------------------------------------------

subPila :: Eq a => Pila a -> Pila a -> Bool
subPila p1 p2
    | esVacia p1 = True
    | esVacia p2 = False
    | cp1 == cp2 = prefijoPila dp1 dp2 || subPila p1 dp2
    | otherwise  = subPila p1 dp2
    where cp1 = cima p1
          dp1 = desapila p1
          cp2 = cima p2
          dp2 = desapila p2

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    ordenadaPila :: Ord a => Pila a -> Bool
-- tal que (ordenadaPila p) se verifica si los elementos de la pila p
-- están ordenados en orden creciente. Por ejemplo,
--    ordenadaPila ejP1 == True
--    ordenadaPila ejP4 == False
-- ---------------------------------------------------------------------

ordenadaPila :: Ord a => Pila a -> Bool
ordenadaPila p
  | esVacia p  = True
  | esVacia dp = True
  | otherwise  = cp <= cdp && ordenadaPila dp
  where cp  = cima p
```

```
        dp  = desapila p
        cdp = cima dp

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir la función
--    lista2Pila :: [a] -> Pila a
-- tal que (lista2Pila xs) es la pila formada por los elementos de
-- xs. Por ejemplo,
--    lista2Pila [1..6] == 1|2|3|4|5|6|-
-- ---------------------------------------------------------------------

lista2Pila :: [a] -> Pila a
lista2Pila = foldr apila vacia

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Definir la función
--    pila2Lista :: Pila a -> [a]
-- tal que (pila2Lista p) es la lista formada por los elementos de la
-- lista p. Por ejemplo,
--    pila2Lista ejP2 == [2,5,8,11,14,17]
-- ---------------------------------------------------------------------

pila2Lista :: Pila a -> [a]
pila2Lista p
    | esVacia p = []
    | otherwise = cp : pila2Lista dp
    where cp = cima p
          dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Comprobar con QuickCheck que la función pila2Lista es
-- la inversa de lista2Pila, y recíprocamente.
-- ---------------------------------------------------------------------

prop_pila2Lista :: Pila Int -> Bool
prop_pila2Lista p =
  lista2Pila (pila2Lista p) == p

-- ghci> quickCheck prop_pila2Lista
-- +++ OK, passed 100 tests.
```

```haskell
prop_lista2Pila :: [Int] -> Bool
prop_lista2Pila xs =
  pila2Lista (lista2Pila xs) == xs

-- ghci> quickCheck prop_lista2Pila
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir la función
--    ordenaInserPila :: Ord a => Pila a -> Pila a
-- tal que (ordenaInserPila p) es la pila obtenida ordenando por
-- inserción los los elementos de la pila p. Por ejemplo,
--    ghci> ordenaInserPila ejP4
--    -1|0|3|3|3|4|4|7|8|10|-
-- ---------------------------------------------------------------------

ordenaInserPila :: Ord a => Pila a -> Pila a
ordenaInserPila p
  | esVacia p = p
  | otherwise = insertaPila cp (ordenaInserPila dp)
  where cp = cima p
        dp = desapila p

insertaPila :: Ord a => a -> Pila a -> Pila a
insertaPila x p
  | esVacia p = apila x p
  | x < cp    = apila x p
  | otherwise = apila cp (insertaPila x dp)
  where cp = cima p
        dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Comprobar con QuickCheck que la pila
--    (ordenaInserPila p)
-- está ordenada correctamente.
-- ---------------------------------------------------------------------

prop_ordenaInserPila :: Pila Int -> Bool
prop_ordenaInserPila p =
```

```
  pila2Lista (ordenaInserPila p) == sort (pila2Lista p)

-- ghci> quickCheck prop_ordenaInserPila
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Definir la función
--    nubPila :: Eq a => Pila a -> Pila a
-- tal que (nubPila p) es la pila con los elementos de p sin repeticiones.
-- Por ejemplo,
--    ghci> ejP4
--    4|-1|7|3|8|10|0|3|3|4|-
--    ghci> nubPila ejP4
--    -1|7|8|10|0|3|4|-
-- ---------------------------------------------------------------------

nubPila :: Eq a => Pila a -> Pila a
nubPila p
  | esVacia p           = vacia
  | pertenecePila cp dp = nubPila dp
  | otherwise           = apila cp (nubPila dp)
  where cp = cima p
        dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Definir la propiedad siguiente: "la composición de
-- las funciones nub y pila2Lista coincide con la composición de las
-- funciones pila2Lista y nubPila", y comprobarla con QuickCheck.
-- En caso de ser falsa, redefinir la función nubPila para que se
-- verifique la propiedad.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_nubPila :: Pila Int -> Bool
prop_nubPila p =
  nub (pila2Lista p) == pila2Lista (nubPila p)

-- La comprobación es
--    ghci> quickCheck prop_nubPila
--    *** Failed! Falsifiable (after 8 tests):
```

```
--    -7|-2|0|-5|-7|-
--    ghci> let p = foldr apila vacia [-7,-2,0,-5,-7]
--    ghci> p
--    -7|-2|0|-5|-7|-
--    ghci> pila2Lista p
--    [-7,-2,0,-5,-7]
--    ghci> nub (pila2Lista p)
--    [-7,-2,0,-5]
--    ghci> nubPila p
--    -2|0|-5|-7|-
--    ghci> pila2Lista (nubPila p)
--    [-2,0,-5,-7]

-- Falla porque nub quita el último de los elementos repetidos de la
-- lista, mientras que nubPila quita el primero de ellos.

-- La redefinimos
nubPila' :: Eq a => Pila a -> Pila a
nubPila' p
  | esVacia p           = p
  | pertenecePila cp dp = apila cp (nubPila' (eliminaPila cp dp))
  | otherwise           = apila cp (nubPila' dp)
  where cp = cima p
        dp = desapila p

eliminaPila :: Eq a => a -> Pila a -> Pila a
eliminaPila x p
    | esVacia p = p
    | x == cp    = eliminaPila x dp
    | otherwise = apila cp (eliminaPila x dp)
    where cp = cima p
          dp = desapila p

-- La propiedad es
prop_nubPila' :: Pila Int -> Bool
prop_nubPila' p =
    nub (pila2Lista p) == pila2Lista (nubPila' p)

-- La comprobación es
--    ghci> quickCheck prop_nubPila'
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    maxPila :: Ord a => Pila a -> a
-- tal que (maxPila p) sea el mayor de los elementos de la pila p. Por
-- ejemplo,
--    ghci> ejP4
--    4|-1|7|3|8|10|0|3|3|4|-
--    ghci> maxPila ejP4
--    10
-- ---------------------------------------------------------------------

maxPila :: Ord a => Pila a -> a
maxPila p
  | esVacia p  = error "pila vacia"
  | esVacia dp = cp
  | otherwise  = max cp (maxPila dp)
  where cp = cima p
        dp = desapila p

-- ---------------------------------------------------------------------
-- Generador de pilas                                                --
-- ---------------------------------------------------------------------

-- genPila es un generador de pilas. Por ejemplo,
--    ghci> sample genPila
--    -
--    0|0|-
--    -
--    -6|4|-3|3|0|-
--    -
--    9|5|-1|-3|0|-8|-5|-7|2|-
--    -3|-10|-3|-12|11|6|1|-2|0|-12|-6|-
--    2|-14|-5|2|-
--    5|9|-
--    -1|-14|5|-
--    6|13|0|17|-12|-7|-8|-19|-14|-5|10|14|3|-18|2|-14|-11|-6|-
genPila :: (Num a, Arbitrary a) => Gen (Pila a)
genPila = do xs <- listOf arbitrary
```

```
            return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
    arbitrary = genPila
```

# Relación 26

# El TAD de las colas

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

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las colas, utilizando las implementaciones estudiadas en el
-- tema 15 transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-15.html
--
-- Para realizar los ejercicios hay que tener instalada la librería de
-- I1M. Para instalarla basta ejecutar en una consola
--    cabal update
--    cabal install I1M
--
-- Otra forma es descargar las implementaciones de las implementaciones
-- de las colas:
-- + ColaConListas.hs    que está en http://bit.ly/1oNxWQq
-- + ColaConDosListas.hs que está en http://bit.ly/1oNxZMe
{-# OPTIONS_GHC -fno-warn-unused-matches
                -fno-warn-unused-imports
                -fno-warn-orphans
#-}

module Rel_26_sol where

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

import Data.List
```

```haskell
import Test.QuickCheck

-- Hay que elegir una implementación del TAD colas:
import I1M.Cola
-- import ColaConListas
-- import ColaConDosListas

-- ---------------------------------------------------------------------
-- Nota. A lo largo de la relación de ejercicios usaremos los siguientes
-- ejemplos de colas:
c1, c2, c3, c4, c5, c6 :: Cola Int
c1 = foldr inserta vacia [1..20]
c2 = foldr inserta vacia [2,5..18]
c3 = foldr inserta vacia [3..10]
c4 = foldr inserta vacia [4,-1,7,3,8,10,0,3,3,4]
c5 = foldr inserta vacia [15..20]
c6 = foldr inserta vacia (reverse [1..20])
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1: Definir la función
--    ultimoCola :: Cola a -> a
-- tal que (ultimoCola c) es el último elemento de la cola c. Por
-- ejemplo:
--    ultimoCola c4 == 4
--    ultimoCola c5 == 15
-- ---------------------------------------------------------------------

ultimoCola :: Cola a -> a
ultimoCola c
    | esVacia c  = error "cola vacia"
    | esVacia rc = pc
    | otherwise  = ultimoCola rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    longitudCola :: Cola a -> Int
-- tal que (longitudCola c) es el número de elementos de la cola c. Por
```

```haskell
-- ejemplo,
--    longitudCola c2 == 6
-- ---------------------------------------------------------------------

longitudCola :: Cola a -> Int
longitudCola c
    | esVacia c = 0
    | otherwise = 1 + longitudCola rc
    where rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 3: Definir la función
--    todosVerifican :: (a -> Bool) -> Cola a -> Bool
-- tal que (todosVerifican p c) se verifica si todos los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--    todosVerifican (>0) c1 == True
--    todosVerifican (>0) c4 == False
-- ---------------------------------------------------------------------

todosVerifican :: (a -> Bool) -> Cola a -> Bool
todosVerifican p c
    | esVacia c = True
    | otherwise = p pc && todosVerifican p rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 4: Definir la función
--    algunoVerifica :: (a -> Bool) -> Cola a -> Bool
-- tal que (algunoVerifica p c) se verifica si algún elemento de la cola
-- c cumple la propiedad p. Por ejemplo,
--    algunoVerifica (<0) c1 == False
--    algunoVerifica (<0) c4 == True
-- ---------------------------------------------------------------------

algunoVerifica :: (a -> Bool) -> Cola a -> Bool
algunoVerifica p c
    | esVacia c = False
    | otherwise = p pc || algunoVerifica p rc
    where pc = primero c
```

```
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 5: Definir la función
--    ponAlaCola :: Cola a -> Cola a -> Cola a
-- tal que (ponAlaCola c1 c2) es la cola que resulta de poner los
-- elementos de c2 a la cola de c1. Por ejemplo,
--    ponAlaCola c2 c3 == C [17,14,11,8,5,2,10,9,8,7,6,5,4,3]
-- ---------------------------------------------------------------------

ponAlaCola :: Cola a -> Cola a -> Cola a
ponAlaCola c1 c2
    | esVacia c2 = c1
    | otherwise  = ponAlaCola (inserta pc2 c1) rq2
    where pc2 = primero c2
          rq2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 6: Definir la función
--    mezclaColas :: Cola a -> Cola a -> Cola a
-- tal que (mezclaColas c1 c2) es la cola formada por los elementos de
-- c1 y c2 colocados en una cola, de forma alternativa, empezando por
-- los elementos de c1. Por ejemplo,
--    mezclaColas c2 c4 == C [17,4,14,3,11,3,8,0,5,10,2,8,3,7,-1,4]
-- ---------------------------------------------------------------------

mezclaColas :: Cola a -> Cola a -> Cola a
mezclaColas c1 c2 = aux c1 c2 vacia
    where aux c1 c2 c
              | esVacia c1 = ponAlaCola c c2
              | esVacia c2 = ponAlaCola c c1
              | otherwise = aux rc1 rc2 (inserta pc2 (inserta pc1 c))
              where pc1 = primero c1
                    rc1 = resto c1
                    pc2 = primero c2
                    rc2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 7: Definir la función
--    agrupaColas :: [Cola a] -> Cola a
```

```haskell
-- tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
-- las colas de la lista como sigue: mezcla c1 con c2, el resultado con
-- c3, el resultado con c4, y así sucesivamente. Por ejemplo,
--    ghci> agrupaColas [c3,c3,c4]
--    C [10,4,10,3,9,3,9,0,8,10,8,8,7,3,7,7,6,-1,6,4,5,5,4,4,3,3]
-- ---------------------------------------------------------------------

agrupaColas :: [Cola a] -> Cola a
agrupaColas []           = vacia
agrupaColas [c]          = c
agrupaColas (c1:c2:colas) = agrupaColas (mezclaColas c1 c2 : colas)

-- 2ª solución
agrupaColas2 :: [Cola a] -> Cola a
agrupaColas2 = foldl mezclaColas vacia

-- ---------------------------------------------------------------------
-- Ejercicio 8: Definir la función
--    perteneceCola :: Eq a => a -> Cola a -> Bool
-- tal que (perteneceCola x c) se verifica si x es un elemento de la
-- cola c. Por ejemplo,
--    perteneceCola 7 c1  == True
--    perteneceCola 70 c1 == False
-- ---------------------------------------------------------------------

perteneceCola :: Eq a => a -> Cola a -> Bool
perteneceCola x c
    | esVacia c  = False
    | otherwise  = pc == x || perteneceCola x rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 9: Definir la función
--    contenidaCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (contenidaCola c1 c2) se verifica si todos los elementos de
-- c1 son elementos de c2. Por ejemplo,
--    contenidaCola c2 c1 == True
--    contenidaCola c1 c2 == False
-- ---------------------------------------------------------------------
```

```haskell
contenidaCola :: Eq a => Cola a -> Cola a -> Bool
contenidaCola c1 c2
    | esVacia c1 = True
    | esVacia c2 = False
    | otherwise  = perteneceCola pc1 c2 && contenidaCola rc1 c2
    where pc1 = primero c1
          rc1 = resto c1

-- ---------------------------------------------------------------------
-- Ejercicio 10: Definir la función
--    prefijoCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (prefijoCola c1 c2) se verifica si la cola c1 es un prefijo
-- de la cola c2. Por ejemplo,
--    prefijoCola c3 c2 == False
--    prefijoCola c5 c1 == True
-- ---------------------------------------------------------------------

prefijoCola :: Eq a => Cola a -> Cola a -> Bool
prefijoCola c1 c2
    | esVacia c1 = True
    | esVacia c2 = False
    | otherwise  = pc1 == pc2 && prefijoCola rc1 rc2
    where pc1 = primero c1
          rc1 = resto c1
          pc2 = primero c2
          rc2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 11: Definir la función
--    subCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (subCola c1 c2) se verifica si c1 es una subcola de c2. Por
-- ejemplo,
--    subCola c2 c1 == False
--    subCola c3 c1 == True
-- ---------------------------------------------------------------------

subCola :: Eq a => Cola a -> Cola a -> Bool
subCola c1 c2
    | esVacia c1 = True
```

```
    | esVacia c2 = False
    | pc1 == pc2 = prefijoCola rc1 rc2 || subCola c1 rc2
    | otherwise  = subCola c1 rc2
    where pc1 = primero c1
          rc1 = resto c1
          pc2 = primero c2
          rc2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 12: Definir la función
--    ordenadaCola :: Ord a => Cola a -> Bool
-- tal que (ordenadaCola c) se verifica si los elementos de la cola c
-- están ordenados en orden creciente. Por ejemplo,
--    ordenadaCola c6 == True
--    ordenadaCola c4 == False
-- ---------------------------------------------------------------------

ordenadaCola :: Ord a => Cola a -> Bool
ordenadaCola c
    | esVacia c = True
    | esVacia rc = True
    | otherwise = pc <= prc && ordenadaCola rc
    where pc  = primero c
          rc  = resto c
          prc = primero rc

-- ---------------------------------------------------------------------
-- Ejercicio 13.1: Definir una función
--    lista2Cola :: [a] -> Cola a
-- tal que (lista2Cola xs) es una cola formada por los elementos de xs.
-- Por ejemplo,
--    lista2Cola [1..6] == C [1,2,3,4,5,6]
-- ---------------------------------------------------------------------

lista2Cola :: [a] -> Cola a
lista2Cola xs = foldr inserta vacia (reverse xs)

-- ---------------------------------------------------------------------
-- Ejercicio 13.2: Definir una función
--    cola2Lista :: Cola a -> [a]
```

```haskell
-- tal que (cola2Lista c) es la lista formada por los elementos de p.
-- Por ejemplo,
--    cola2Lista c2 == [17,14,11,8,5,2]
-- ---------------------------------------------------------------------

cola2Lista :: Cola a -> [a]
cola2Lista c
    | esVacia c = []
    | otherwise = pc : cola2Lista rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 13.3. Comprobar con QuickCheck que la función cola2Lista es
-- la inversa de  lista2Cola, y recíprocamente.
-- ---------------------------------------------------------------------

prop_cola2Lista :: Cola Int -> Bool
prop_cola2Lista c =
    lista2Cola (cola2Lista c) == c

-- ghci> quickCheck prop_cola2Lista
-- +++ OK, passed 100 tests.

prop_lista2Cola :: [Int] -> Bool
prop_lista2Cola xs =
    cola2Lista (lista2Cola xs) == xs

-- ghci> quickCheck prop_lista2Cola
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 14: Definir la función
--    maxCola :: Ord a => Cola a -> a
-- tal que (maxCola c) es el mayor de los elementos de la cola c. Por
-- ejemplo,
--    maxCola c4 == 10
-- ---------------------------------------------------------------------

maxCola :: Ord a => Cola a -> a
```

```haskell
maxCola c
    | esVacia c = error "cola vacia"
    | esVacia rc = pc
    | otherwise = max pc (maxCola rc)
    where pc = primero c
          rc = resto c

prop_maxCola c =
    not (esVacia c) ==>
        maxCola c == maximum (cola2Lista c)

-- ghci> quickCheck prop_maxCola
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Generador de colas                                                --
-- ---------------------------------------------------------------------

-- genCola es un generador de colas de enteros. Por ejemplo,
--    ghci> sample genCola
--    C ([],[])
--    C ([],[])
--    C ([],[])
--    C ([],[])
--    C ([7,8,4,3,7],[5,3,3])
--    C ([],[])
--    C ([1],[13])
--    C ([18,28],[12,21,28,28,3,18,14])
--    C ([47],[64,45,7])
--    C ([8],[])
--    C ([42,112,178,175,107],[])
genCola :: (Num a, Arbitrary a) => Gen (Cola a)
genCola = frequency [(1, return vacia),
                     (30, do n <- choose (10,100)
                             xs <- vectorOf n arbitrary
                             return (creaCola xs))]
          where creaCola = foldr inserta vacia

-- El tipo cola es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
```

```
  arbitrary = genCola
```

# Relación 27

# Operaciones con conjuntos

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

-- El objetivo de esta relación de ejercicios es definir operaciones
-- entre conjuntos, representados mediante listas ordenadas sin
-- repeticiones, explicado en el tema 17 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-17.html

{-# LANGUAGE FlexibleInstances #-}

module Rel_27_sol where

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- § Representación de conjuntos y operaciones básicas                --
-- ---------------------------------------------------------------------

-- Los conjuntos como listas ordenadas sin repeticiones.
newtype Conj a = Cj [a]
  deriving Eq

-- Ejemplo de conjunto:
--    λ> c1
```

```haskell
--    Cj [0,1,2,3,5,7,9]
c1 :: Conj Int
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]

-- Procedimiento de escritura de los conjuntos.
instance Show a => Show (Conj a) where
  show = escribeConj

-- (escribeConj c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--    λ> c1
--    Cj [0,1,2,3,5,7,9]
--    λ> escribeConj c1
--    "{0,1,2,3,5,7,9}"
escribeConj :: Show a => Conj a -> String
escribeConj (Cj [])     = "{}"
escribeConj (Cj (x:xs)) = "{" ++ show x ++ aux xs
  where aux []     = "}"
        aux (x:xs) = "," ++ show x ++ aux xs

-- vacio es el conjunto vacío. Por ejemplo,
--    λ> vacio
--    Cj []
--    λ> escribeConj vacio
--    "{}"
vacio :: Conj a
vacio = Cj []

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--    esVacio c1     ==  False
--    esVacio vacio  ==  True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--    λ> c1
--    Cj [0,1,2,3,5,7,9]
--    λ> pertenece 3 c1
--    True
--    λ> pertenece 4 c1
```

```
--    False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = elem x (takeWhile (<= x) s)

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--    λ> c1
--    Cj [0,1,2,3,5,7,9]
--    λ> inserta 5 c1
--    Cj [0,1,2,3,5,7,9]
--    λ> inserta 4 c1
--    Cj [0,1,2,3,4,5,7,9]
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj ys) = Cj (insertaL x ys)

-- (insertaL x ys) es la lista obtenida añadiendo el elemento x a la
-- lista ordenada ys. Por ejemplo,
--    λ> insertaL 5 [0,1,2,3,5,7,9]
--    [0,1,2,3,5,7,9]
--    λ> insertaL 4 [0,1,2,3,5,7,9]
--    [0,1,2,3,4,5,7,9]
insertaL :: Ord a => a -> [a] -> [a]
insertaL x []                 = [x]
insertaL x (y:ys) | x > y     = y : insertaL x ys
                  | x < y     = x : y : ys
                  | otherwise = y : ys

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--    λ> c1
--    Cj [0,1,2,3,5,7,9]
--    λ> elimina 3 c1
--    Cj [0,1,2,5,7,9]
--    λ> elimina 4 c1
--    Cj [0,1,2,3,5,7,9]
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj ys) = Cj (eliminaL x ys)

-- (eliminaL x ys) es la lista obtenida eliminando el elemento x
-- de la lista ordenada ys. Por ejemplo,
```

```
--    λ> eliminaL 3 [0,1,2,3,5,7,9]
--    [0,1,2,5,7,9]
--    λ> eliminaL 4 [0,1,2,3,5,7,9]
--    [0,1,2,3,5,7,9]
eliminaL :: Ord a => a -> [a] -> [a]
eliminaL x []                 = []
eliminaL x (y:ys) | x > y     = y : eliminaL x ys
                  | x < y     = y : ys
                  | otherwise = ys

-- Ejemplos de conjunto:
c2, c3, c4 :: Conj Int
c2 = foldr inserta vacio [2,6,8,6,1,2,1,9,6]
c3 = Cj [2..100000]
c4 = Cj [1..100000]

-- ---------------------------------------------------------------------
-- § Ejercicios                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    subconjunto :: Ord a => Conj a -> Conj a -> Bool
-- tal que (subconjunto c1 c2) se verifica si todos los elementos de c1
-- pertenecen a c2. Por ejemplo,
--    subconjunto (Cj [2..100000]) (Cj [1..100000]) == True
--    subconjunto (Cj [1..100000]) (Cj [2..100000]) == False
-- ---------------------------------------------------------------------

-- 1ª definición
subconjunto1 :: Ord a => Conj a -> Conj a -> Bool
subconjunto1 (Cj xs) (Cj ys) = sublista xs ys
  where sublista [] _      = True
        sublista (x:xs) ys = elem x ys && sublista xs ys

-- 2ª definición
subconjunto2 :: Ord a => Conj a -> Conj a -> Bool
subconjunto2 (Cj xs) c =
  and [pertenece x c | x <- xs]
```

```haskell
-- 3ª definición
subconjunto3 :: Ord a => Conj a -> Conj a -> Bool
subconjunto3 (Cj xs) (Cj ys) = sublista' xs ys
  where sublista' [] _      = True
        sublista' _ []      = False
        sublista' (x:xs) ys@(y:zs) = x >= y && elem x ys &&
                                     sublista' xs zs

-- 4ª definición
subconjunto4 :: Ord a => Conj a -> Conj a -> Bool
subconjunto4 (Cj xs) (Cj ys) = sublista' xs ys
  where sublista' [] _      = True
        sublista' _ []      = False
        sublista' (x:xs) ys@(y:zs)
            | x < y  = False
            | x == y = sublista' xs zs
            | x > y  = elem x zs && sublista' xs zs

-- Comparación de la eficiencia:
--    λ> subconjunto1 (Cj [2..100000]) (Cj [1..1000000])
--      C-c C-cInterrupted.
--    λ> subconjunto2 (Cj [2..100000]) (Cj [1..1000000])
--      C-c C-cInterrupted.
--    λ> subconjunto3 (Cj [2..100000]) (Cj [1..1000000])
--    True
--    (0.52 secs, 26097076 bytes)
--    λ> subconjunto4 (Cj [2..100000]) (Cj [1..1000000])
--    True
--    (0.66 secs, 32236700 bytes)
--    λ> subconjunto1 (Cj [2..100000]) (Cj [1..10000])
--    False
--    (0.54 secs, 3679024 bytes)
--    λ> subconjunto2 (Cj [2..100000]) (Cj [1..10000])
--    False
--    (38.19 secs, 1415562032 bytes)
--    λ> subconjunto3 (Cj [2..100000]) (Cj [1..10000])
--    False
--    (0.08 secs, 3201112 bytes)
--    λ> subconjunto4 (Cj [2..100000]) (Cj [1..10000])
--    False
```

```
--    (0.09 secs, 3708988 bytes)

-- En lo que sigue, se usará la 3ª definición:
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto = subconjunto3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
-- tal (subconjuntoPropio c1 c2) se verifica si c1 es un subconjunto
-- propio de c2. Por ejemplo,
--   subconjuntoPropio (Cj [2..5]) (Cj [1..7]) == True
--   subconjuntoPropio (Cj [2..5]) (Cj [1..4]) == False
--   subconjuntoPropio (Cj [2..5]) (Cj [2..5]) == False
-- ---------------------------------------------------------------------

subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
subconjuntoPropio c1 c2 =
  subconjunto c1 c2 && c1 /= c2

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    unitario :: Ord a => a -> Conj a
-- tal que (unitario x) es el conjunto {x}. Por ejemplo,
--   unitario 5 == {5}
-- ---------------------------------------------------------------------

unitario :: Ord a => a -> Conj a
unitario x = inserta x vacio

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    cardinal :: Conj a -> Int
-- tal que (cardinal c) es el número de elementos del conjunto c. Por
-- ejemplo,
--    cardinal c1 == 7
--    cardinal c2 == 5
-- ---------------------------------------------------------------------

cardinal :: Conj a -> Int
```

```haskell
cardinal (Cj xs) = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    union :: Ord a => Conj a -> Conj a -> Conj a
-- tal (union c1 c2) es la unión de ambos conjuntos. Por ejemplo,
--    union c1 c2             == {0,1,2,3,5,6,7,8,9}
--    cardinal (union2 c3 c4) == 100000
-- ---------------------------------------------------------------------

-- 1ª definición:
union1 :: Ord a => Conj a -> Conj a -> Conj a
union1 (Cj xs) (Cj ys) = foldr inserta (Cj ys) xs

-- Otra definión es
union2 :: Ord a => Conj a -> Conj a -> Conj a
union2 (Cj xs) (Cj ys) = Cj (unionL xs ys)
  where unionL [] ys = ys
        unionL xs [] = xs
        unionL l1@(x:xs) l2@(y:ys)
            | x < y  = x : unionL xs l2
            | x == y = x : unionL xs ys
            | x > y  = y : unionL l1 ys

-- Comparación de eficiencia
--    λ> :set +s
--    λ> let c = Cj [1..1000]
--    λ> cardinal (union1 c c)
--    1000
--    (1.04 secs, 56914332 bytes)
--    λ> cardinal (union2 c c)
--    1000
--    (0.01 secs, 549596 bytes)

-- En lo que sigue se usará la 2ª definición
union :: Ord a => Conj a -> Conj a -> Conj a
union = union2

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
```

```haskell
--    unionG:: Ord a => [Conj a] -> Conj a
-- tal (unionG cs) calcule la unión de la lista de conjuntos cd. Por
-- ejemplo,
--    unionG [c1, c2] == {0,1,2,3,5,6,7,8,9}
-- ---------------------------------------------------------------------

unionG :: Ord a => [Conj a] -> Conj a
unionG []          = vacio
unionG (Cj xs:css) = Cj xs `union` unionG css

-- Se puede definir por plegados
unionG2 :: Ord a => [Conj a] -> Conj a
unionG2 = foldr union vacio

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    interseccion :: Eq a => Conj a -> Conj a -> Conj a
-- tal que (interseccion c1 c2) es la intersección de los conjuntos c1 y
-- c2. Por ejemplo,
--    interseccion (Cj [1..7]) (Cj [4..9])    == {4,5,6,7}
--    interseccion (Cj [2..1000000]) (Cj [1]) == {}
-- ---------------------------------------------------------------------

-- 1ª definición
interseccion1 :: Eq a => Conj a -> Conj a -> Conj a
interseccion1 (Cj xs) (Cj ys) = Cj [x | x <- xs, x `elem` ys]

-- 2ª definición
interseccion2 :: Ord a => Conj a -> Conj a -> Conj a
interseccion2 (Cj xs) (Cj ys) = Cj (interseccionL xs ys)
  where interseccionL l1@(x:xs) l2@(y:ys)
          | x > y   = interseccionL l1 ys
          | x == y  = x : interseccionL xs ys
          | x < y   = interseccionL xs l2
        interseccionL _ _ = []

-- La comparación de eficiencia es
-- λ> interseccion1 (Cj [2..1000000]) (Cj [1])
-- {}
-- (0.32 secs, 80396188 bytes)
```

```
-- λ> interseccion2 (Cj [2..1000000]) (Cj [1])
-- {}
-- (0.00 secs, 2108848 bytes)

-- En lo que sigue se usa la 2ª definición:
interseccion :: Ord a => Conj a -> Conj a -> Conj a
interseccion = interseccion2

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    interseccionG:: Ord a => [Conj a] -> Conj a
-- tal que (interseccionG cs) es la intersección de la lista de
-- conjuntos cs. Por ejemplo,
--    interseccionG [c1, c2] == {1,2,9}
-- ---------------------------------------------------------------------

interseccionG :: Ord a => [Conj a] -> Conj a
interseccionG [c]      = c
interseccionG (cs:css) = interseccion cs (interseccionG css)

-- Se puede definir por plegado
interseccionG2 :: Ord a => [Conj a] -> Conj a
interseccionG2 = foldr1 interseccion

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    disjuntos :: Ord a => Conj a -> Conj a -> Bool
-- tal que (disjuntos c1 c2) se verifica si los conjuntos c1 y c2 son
-- disjuntos. Por ejemplo,
--   disjuntos (Cj [2..5]) (Cj [6..9]) == True
--   disjuntos (Cj [2..5]) (Cj [1..9]) == False
-- ---------------------------------------------------------------------

disjuntos :: Ord a => Conj a -> Conj a -> Bool
disjuntos c1 c2 = esVacio (interseccion c1 c2)

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    diferencia :: Eq a => Conj a -> Conj a -> Conj a
-- tal que (diferencia c1 c2) es el conjunto de los elementos de c1 que
```

```haskell
-- no son elementos de c2. Por ejemplo,
--    diferencia c1 c2 == {0,3,5,7}
--    diferencia c2 c1 == {6,8}
-- ---------------------------------------------------------------------

diferencia :: Eq a => Conj a -> Conj a -> Conj a
diferencia (Cj xs) (Cj ys) = Cj zs
  where zs = [x | x <- xs, x `notElem` ys]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
-- tal que (diferenciaSimetrica c1 c2) es la diferencia simétrica de los
-- conjuntos c1 y c2. Por ejemplo,
--   diferenciaSimetrica c1 c2 == {0,3,5,6,7,8}
--   diferenciaSimetrica c2 c1 == {0,3,5,6,7,8}
-- ---------------------------------------------------------------------

diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
diferenciaSimetrica c1 c2 =
  diferencia (union c1 c2) (interseccion c1 c2)

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    filtra :: (a -> Bool) -> Conj a -> Conj a
-- tal (filtra p c) es el conjunto de elementos de c que verifican el
-- predicado p. Por ejemplo,
--    filtra even c1 == {0,2}
--    filtra odd  c1  == {1,3,5,7,9}
-- ---------------------------------------------------------------------

filtra :: (a -> Bool) -> Conj a -> Conj a
filtra p (Cj xs) = Cj (filter p xs)

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    particion :: (a -> Bool) -> Conj a -> (Conj a, Conj a)
-- tal que (particion c) es el par formado por dos conjuntos: el de sus
-- elementos que verifican p y el de los elementos que no lo
-- verifica. Por ejemplo,
```

```
--    particion even c1 == ({0,2},{1,3,5,7,9})
-- ---------------------------------------------------------------------

particion :: (a -> Bool) -> Conj a -> (Conj a, Conj a)
particion p c = (filtra p c, filtra (not . p) c)

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    divide :: (Ord a) => a-> Conj a -> (Conj a, Conj a)
-- tal que (divide x c) es el par formado por dos subconjuntos de c: el
-- de los elementos menores o iguales que x y el de los mayores que x.
-- Por ejemplo,
--    divide 5 c1 == ({0,1,2,3,5},{7,9})
-- ---------------------------------------------------------------------

divide :: Ord a => a-> Conj a -> (Conj a, Conj a)
divide x = particion (<= x)

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    mapC :: (a -> b) -> Conj a -> Conj b
-- tal que (map f c) es el conjunto formado por las imágenes de los
-- elementos de c, mediante f. Por ejemplo,
--   mapC (*2) (Cj [1..4]) == {2,4,6,8}
-- ---------------------------------------------------------------------

mapC :: (a -> b) -> Conj a -> Conj b
mapC f (Cj xs) = Cj (map f xs)

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    everyC :: (a -> Bool) -> Conj a -> Bool
-- tal que (everyC p c) se verifica si todos los elemsntos de c
-- verifican el predicado p.  Por ejmplo,
--   everyC even (Cj [2,4..10]) == True
--   everyC even (Cj [2..10])   == False
-- ---------------------------------------------------------------------

everyC :: (a -> Bool) -> Conj a -> Bool
everyC p (Cj xs) = all p xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    someC :: (a -> Bool) -> Conj a -> Bool
-- tal que (someC p c) se verifica si algún elemento de c verifica el
-- predicado p. Por ejemplo,
--   someC even (Cj [1,4,7]) == True
--   someC even (Cj [1,3,7]) == False
-- ---------------------------------------------------------------------

someC :: (a -> Bool) -> Conj a -> Bool
someC p (Cj xs) = any p xs

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
-- tal que (productoC c1 c2) es el producto cartesiano de los
-- conjuntos c1 y c2. Por ejemplo,
--   productoC (Cj [1,3]) (Cj [2,4])== {(1,2),(1,4),(3,2),(3,4)}
-- ---------------------------------------------------------------------

productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
productoC (Cj xs) (Cj ys) =
  foldr inserta vacio [(x,y) | x <- xs, y <- ys]

-- ---------------------------------------------------------------------
-- Ejercicio. Especificar que, dado un tipo ordenado a, el orden entre
-- los conjuntos con elementos en a es el orden inducido por el orden
-- existente entre las listas con elementos en a.
-- ---------------------------------------------------------------------

instance Ord a => Ord (Conj a) where
  (Cj xs) <= (Cj ys) = xs <= ys

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    potencia :: Ord a => Conj a -> Conj (Conj a)
-- tal que (potencia c) es el conjunto potencia de c; es decir, el
-- conjunto de todos los subconjuntos de c. Por ejemplo,
--    potencia (Cj [1,2])  == {{},{1},{1,2},{2}}
```

```
--    potencia (Cj [1..3]) == {{},{1},{1,2},{1,2,3},{1,3},{2},{2,3},{3}}
-- ---------------------------------------------------------------------

potencia :: Ord a => Conj a -> Conj (Conj a)
potencia (Cj []) = unitario vacio
potencia (Cj (x:xs)) = mapC (inserta x) pr `union` pr
  where pr = potencia (Cj xs)

-- ---------------------------------------------------------------------
-- Ejercicio 20. Comprobar con QuickCheck que la relación de subconjunto
-- es un orden parcial. Es decir, es una relación reflexiva,
-- antisimétrica y transitiva.
-- ---------------------------------------------------------------------

propSubconjuntoReflexiva :: Conj Int -> Bool
propSubconjuntoReflexiva c = subconjunto c c

-- La comprobación es
--    λ> quickCheck propSubconjuntoReflexiva
--    +++ OK, passed 100 tests.

propSubconjuntoAntisimetrica :: Conj Int -> Conj Int -> Property
propSubconjuntoAntisimetrica c1 c2 =
  subconjunto c1 c2 && subconjunto c2 c1 ==> c1 == c2

-- La comprobación es
--    λ> quickCheck propSubconjuntoAntisimetrica
--    *** Gave up! Passed only 13 tests.

propSubconjuntoTransitiva :: Conj Int -> Conj Int -> Conj Int -> Property
propSubconjuntoTransitiva c1 c2 c3 =
  subconjunto c1 c2 && subconjunto c2 c3 ==> subconjunto c1 c3

-- La comprobación es
--    λ> quickCheck propSubconjuntoTransitiva
--    *** Gave up! Passed only 7 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 21. Comprobar con QuickCheck que el conjunto vacío está
-- contenido en cualquier conjunto.
```

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

propSubconjuntoVacio :: Conj Int -> Bool
propSubconjuntoVacio c = subconjunto vacio c

-- La comprobación es
--    λ> quickCheck propSubconjuntoVacio
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 22. Comprobar con QuickCheck las siguientes propiedades de
-- la unión de conjuntos:
--    Idempotente:      A U A = A
--    Neutro:           A U {} = A
--    Commutativa:      A U B = B U A
--    Asociativa:       A U (B U C) = (A U B) U C
--    UnionSubconjunto: A y B son subconjuntos de (A U B)
--    UnionDiferencia:  A U B = A U (B \ A)
-- ---------------------------------------------------------------------

propUnionIdempotente :: Conj Int -> Bool
propUnionIdempotente c =
  union c c == c

-- La comprobación es
--    λ> quickCheck propUnionIdempotente
--    +++ OK, passed 100 tests.

propVacioNeutroUnion :: Conj Int -> Bool
propVacioNeutroUnion c =
  union c vacio == c

-- La comprobación es
--    λ> quickCheck propVacioNeutroUnion
--    +++ OK, passed 100 tests.

propUnionCommutativa :: Conj Int -> Conj Int -> Bool
propUnionCommutativa c1 c2 =
  union c1 c2 == union c2 c1
```

```haskell
-- La comprobación es
--    λ> quickCheck propUnionCommutativa
--    +++ OK, passed 100 tests.

propUnionAsociativa :: Conj Int -> Conj Int -> Conj Int -> Bool
propUnionAsociativa c1 c2 c3 =
  union c1 (union c2 c3) == union (union c1 c2) c3

-- La comprobación es
--    λ> quickCheck propUnionAsociativa
--    +++ OK, passed 100 tests.

propUnionSubconjunto :: Conj Int -> Conj Int -> Bool
propUnionSubconjunto c1 c2 =
  subconjunto c1 c3 && subconjunto c2 c3
  where c3 = union c1 c2

-- La comprobación es
--    λ> quickCheck propUnionSubconjunto
--    +++ OK, passed 100 tests.

propUnionDiferencia :: Conj Int -> Conj Int -> Bool
propUnionDiferencia c1 c2 =
  union c1 c2 == union c1 (diferencia c2 c1)

-- La comprobación es
--    λ> quickCheck propUnionDiferencia
--    +++ OK, passed 100 tests.

-- ----------------------------------------------------------------------
-- Ejercicio 23. Comprobar con QuickCheck las siguientes propiedades de
-- la intersección de conjuntos:
--    Idempotente:             A n A = A
--    VacioInterseccion:       A n {} = {}
--    Commutativa:             A n B = B n A
--    Asociativa:              A n (B n C) = (A n B) n C
--    InterseccionSubconjunto: (A n B) es subconjunto de A y B
--    DistributivaIU:          A n (B U C) = (A n B) U (A n C)
--    DistributivaUI:          A U (B n C) = (A U B) n (A U C)
-- ----------------------------------------------------------------------
```

```haskell
propInterseccionIdempotente :: Conj Int -> Bool
propInterseccionIdempotente c =
  interseccion c c == c

-- La comprobación es
--    λ> quickCheck propInterseccionIdempotente
--    +++ OK, passed 100 tests.

propVacioInterseccion :: Conj Int -> Bool
propVacioInterseccion c =
  interseccion c vacio == vacio

-- La comprobación es
--    λ> quickCheck propVacioInterseccion
--    +++ OK, passed 100 tests.

propInterseccionCommutativa :: Conj Int -> Conj Int -> Bool
propInterseccionCommutativa c1 c2 =
  interseccion c1 c2 == interseccion c2 c1

-- La comprobación es
--    λ> quickCheck propInterseccionCommutativa
--    +++ OK, passed 100 tests.

propInterseccionAsociativa :: Conj Int -> Conj Int -> Conj Int -> Bool
propInterseccionAsociativa c1 c2 c3 =
  interseccion c1 (interseccion c2 c3) == interseccion (interseccion c1 c2) c3

-- La comprobación es
--    λ> quickCheck propInterseccionAsociativa
--    +++ OK, passed 100 tests.

propInterseccionSubconjunto :: Conj Int -> Conj Int -> Bool
propInterseccionSubconjunto c1 c2 =
  subconjunto c3 c1 && subconjunto c3 c2
  where c3 = interseccion c1 c2

-- La comprobación es
```

```
--    λ> quickCheck propInterseccionSubconjunto
--    +++ OK, passed 100 tests.

propDistributivaIU :: Conj Int -> Conj Int -> Conj Int -> Bool
propDistributivaIU c1 c2 c3 =
  interseccion c1 (union c2 c3) == union (interseccion c1 c2)
                                         (interseccion c1 c3)

-- La comprobación es
--    λ> quickCheck propDistributivaIU
--    +++ OK, passed 100 tests.

propDistributivaUI :: Conj Int -> Conj Int -> Conj Int -> Bool
propDistributivaUI c1 c2 c3 =
  union c1 (interseccion c2 c3) == interseccion (union c1 c2)
                                                (union c1 c3)
-- La comprobación es
--    λ> quickCheck propDistributivaUI
--    +++ OK, passed 100 tests.

-- ----------------------------------------------------------------------
-- Ejercicio 24. Comprobar con QuickCheck las siguientes propiedades de
-- la diferencia de conjuntos:
--    DiferenciaVacio1: A \ {} = A
--    DiferenciaVacio2: {} \ A = {}
--    DiferenciaDif1:   (A \ B) \ C = A \ (B U C)
--    DiferenciaDif2:   A \ (B \ C) = (A \ B) U (A n C)
--    DiferenciaSubc:   (A \ B) es subconjunto de A
--    DiferenciaDisj:   A y (B \ A) son disjuntos
--    DiferenciaUI:     (A U B) \ A = B \ (A n B)
-- ----------------------------------------------------------------------

propDiferenciaVacio1 :: Conj Int -> Bool
propDiferenciaVacio1 c = diferencia c vacio == c

-- La comprobación es
--    λ> quickCheck propDiferenciaVacio2
--    +++ OK, passed 100 tests.

propDiferenciaVacio2 :: Conj Int -> Bool
```

```
propDiferenciaVacio2 c = diferencia vacio c == vacio

-- La comprobación es
--    λ> quickCheck propDiferenciaVacio2
--    +++ OK, passed 100 tests.

propDiferenciaDif1 :: Conj Int -> Conj Int -> Conj Int -> Bool
propDiferenciaDif1 c1 c2 c3 =
  diferencia (diferencia c1 c2) c3 == diferencia c1 (union c2 c3)

-- La comprobación es
--    λ> quickCheck propDiferenciaDif1
--    +++ OK, passed 100 tests.

propDiferenciaDif2 :: Conj Int -> Conj Int -> Conj Int -> Bool
propDiferenciaDif2 c1 c2 c3 =
  diferencia c1 (diferencia c2 c3) == union (diferencia c1 c2)
                                            (interseccion c1 c3)

-- La comprobación es
--    λ> quickCheck propDiferenciaDif2
--    +++ OK, passed 100 tests.

propDiferenciaSubc :: Conj Int -> Conj Int -> Bool
propDiferenciaSubc c1 c2 =
  subconjunto (diferencia c1 c2) c1

-- La comprobación es
--    λ> quickCheck propDiferenciaSubc
--    +++ OK, passed 100 tests.

propDiferenciaDisj :: Conj Int -> Conj Int -> Bool
propDiferenciaDisj c1 c2 =
  disjuntos c1 (diferencia c2 c1)

-- La comprobación es
--    λ> quickCheck propDiferenciaDisj
--    +++ OK, passed 100 tests.

propDiferenciaUI :: Conj Int -> Conj Int -> Bool
```

```haskell
propDiferenciaUI c1 c2 =
  diferencia (union c1 c2) c1 == diferencia c2 (interseccion c1 c2)

-- La comprobación es
--    λ> quickCheck propDiferenciaUI
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Generador de conjuntos                                              --
-- ---------------------------------------------------------------------

-- genConjunto es un generador de conjuntos. Por ejemplo,
--    λ> sample genConjunto
--    {}
--    {}
--    {}
--    {3,-2,-2,-3,-2,4}
--    {-8,0,4,6,-5,-2}
--    {12,-2,-1,-10,-2,2,15,15}
--    {2}
--    {}
--    {-42,55,55,-11,23,23,-11,27,-17,-48,16,-15,-7,5,41,43}
--    {-124,-66,-5,-47,58,-88,-32,-125}
--    {49,-38,-231,-117,-32,-3,45,227,-41,54,169,-160,19}
genConjunto :: Gen (Conj Int)
genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance Arbitrary (Conj Int) where
  arbitrary = genConjunto
```

# Relación 28

# Operaciones con conjuntos con la librería Data.Set.

```haskell
-- ----------------------------------------------------------------------
-- Introducción                                                         --
-- ----------------------------------------------------------------------

-- El objetivo de esta relación es hacer los ejercicios de la relación 27
-- sobre operaciones con conjuntos usando la librería Data.Set

-- ----------------------------------------------------------------------
-- § Librerías auxiliares                                               --
-- ----------------------------------------------------------------------

import Data.Set as S

-- ----------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    subconjunto :: Ord a => Set a -> Set a -> Bool
-- tal que (subconjunto c1 c2) se verifica si todos los elementos de c1
-- pertenecen a c2. Por ejemplo,
--    subconjunto (fromList [2..100000]) (fromList [1..100000]) == True
--    subconjunto (fromList [1..100000]) (fromList [2..100000]) == False
-- ----------------------------------------------------------------------

subconjunto :: Ord a => Set a -> Set a -> Bool
subconjunto = isSubsetOf

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

```haskell
-- Ejercicio 2. Definir la función
--    subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
-- tal (subconjuntoPropio c1 c2) se verifica si c1 es un subconjunto
-- propio de c2. Por ejemplo,
--   subconjuntoPropio (fromList [2..5]) (fromList [1..7]) == True
--   subconjuntoPropio (fromList [2..5]) (fromList [1..4]) == False
--   subconjuntoPropio (fromList [2..5]) (fromList [2..5]) == False
-- ---------------------------------------------------------------------

subconjuntoPropio :: Ord a => Set a -> Set a -> Bool
subconjuntoPropio = isProperSubsetOf

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    unitario :: Ord a => a -> Set a
-- tal que (unitario x) es el conjunto {x}. Por ejemplo,
--   unitario 5 == fromList [5]
-- ---------------------------------------------------------------------

unitario :: Ord a => a -> Set a
unitario = singleton

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    cardinal :: Set a -> Int
-- tal que (cardinal c) es el número de elementos del conjunto c. Por
-- ejemplo,
--    cardinal (fromList [3,2,5,1,2,3])  ==  4
-- ---------------------------------------------------------------------

cardinal :: Set a -> Int
cardinal = size

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    union' :: Ord a => Set a -> Set a -> Set a
-- tal (union' c1 c2) es la unión de ambos conjuntos. Por ejemplo,
--    ghci> union' (fromList [3,2,5]) (fromList [2,7,5])
--    fromList [2,3,5,7]
-- ---------------------------------------------------------------------
```

```haskell
union' :: Ord a => Set a -> Set a -> Set a
union' = union

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    unionG:: Ord a => [Set a] -> Set a
-- tal (unionG cs) calcule la unión de la lista de conjuntos cd. Por
-- ejemplo,
--    ghci> unionG [fromList [3,2], fromList [2,5], fromList [3,5,7]]
--    fromList [2,3,5,7]
-- ---------------------------------------------------------------------

unionG :: Ord a => [Set a] -> Set a
unionG = unions

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    interseccion :: Ord a => Set a -> Set a -> Set a
-- tal que (interseccion c1 c2) es la intersección de los conjuntos c1 y
-- c2. Por ejemplo,
--    ghci> interseccion (fromList [1..7]) (fromList [4..9])
--    fromList [4,5,6,7]
--    ghci> interseccion (fromList [2..1000000]) (fromList [1])
--    fromList []
-- ---------------------------------------------------------------------

interseccion :: Ord a => Set a -> Set a -> Set a
interseccion = intersection

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    interseccionG:: Ord a => [Set a] -> Set a
-- tal que (interseccionG cs) es la intersección de la lista de
-- conjuntos cs. Por ejemplo,
--    ghci> interseccionG [fromList [3,2], fromList [2,5,3], fromList [3,5,7]]
--    fromList [3]
-- ---------------------------------------------------------------------

interseccionG :: Ord a => [Set a] -> Set a
```

```haskell
interseccionG [c]      = c
interseccionG (cs:css) = intersection cs (interseccionG css)

-- Se puede definir por plegado
interseccionG2 :: Ord a => [Set a] -> Set a
interseccionG2 = foldr1 interseccion

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    disjuntos :: Ord a => Set a -> Set a -> Bool
-- tal que (disjuntos c1 c2) se verifica si los conjuntos c1 y c2 son
-- disjuntos. Por ejemplo,
--   disjuntos (fromList [2..5]) (fromList [6..9]) == True
--   disjuntos (fromList [2..5]) (fromList [1..9]) == False
-- ---------------------------------------------------------------------

disjuntos :: Ord a => Set a -> Set a -> Bool
disjuntos c1 c2 = S.null (intersection c1 c2)

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    diferencia :: Ord a => Set a -> Set a -> Set a
-- tal que (diferencia c1 c2) es el conjunto de los elementos de c1 que
-- no son elementos de c2. Por ejemplo,
--    ghci> diferencia (fromList [2,5,3]) (fromList [1,4,5])
--    fromList [2,3]
-- ---------------------------------------------------------------------

diferencia :: Ord a => Set a -> Set a -> Set a
diferencia = difference

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    diferenciaSimetrica :: Ord a => Set a -> Set a -> Set a
-- tal que (diferenciaSimetrica c1 c2) es la diferencia simétrica de los
-- conjuntos c1 y c2. Por ejemplo,
--    ghci> diferenciaSimetrica (fromList [3,2,5]) (fromList [1,5])
--    fromList [1,2,3]
-- ---------------------------------------------------------------------
```

```haskell
diferenciaSimetrica :: Ord a => Set a -> Set a -> Set a
diferenciaSimetrica c1 c2 =
  (c1 `union` c2) \\ (c1 `intersection` c2)

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    filtra :: (a -> Bool) -> Set a -> Set a
-- tal (filtra p c) es el conjunto de elementos de c que verifican el
-- predicado p. Por ejemplo,
--    filtra even (fromList [3,2,5,6,8,9])  ==  fromList [2,6,8]
--    filtra odd  (fromList [3,2,5,6,8,9])  ==  fromList [3,5,9]
-- ---------------------------------------------------------------------

filtra :: (a -> Bool) -> Set a -> Set a
filtra = S.filter

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    particion :: (a -> Bool) -> Set a -> (Set a, Set a)
-- tal que (particion c) es el par formado por dos conjuntos: el de sus
-- elementos que verifican p y el de los elementos que no lo verifica.
-- Por ejemplo,
--    ghci> particion even (fromList [3,2,5,6,8,9])
--    (fromList [2,6,8],fromList [3,5,9])
-- ---------------------------------------------------------------------

particion :: (a -> Bool) -> Set a -> (Set a, Set a)
particion = partition

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    divide :: (Ord a) => a-> Set a -> (Set a, Set a)
-- tal que (divide x c) es el par formado por dos subconjuntos de c: el
-- de los elementos menores que x y el de los mayores que x. Por ejemplo,
--    ghci> divide 5 (fromList [3,2,9,5,8,6])
--    (fromList [2,3],fromList [6,8,9])
-- ---------------------------------------------------------------------

divide :: Ord a => a-> Set a -> (Set a, Set a)
divide = split
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    mapC :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
-- tal que (map f c) es el conjunto formado por las imágenes de los
-- elementos de c, mediante f. Por ejemplo,
--    mapC (*2) (fromList [1..4])  ==  fromList [2,4,6,8]
-- ---------------------------------------------------------------------

mapC :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
mapC = S.map

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    everyC :: Ord a => (a -> Bool) -> Set a -> Bool
-- tal que (everyC p c) se verifica si todos los elementos de c
-- verifican el predicado p.  Por ejemplo,
--   everyC even (fromList [2,4..10]) == True
--   everyC even (fromList [2..10])   == False
-- ---------------------------------------------------------------------

-- 1ª definición
everyC :: Ord a => (a -> Bool) -> Set a -> Bool
everyC p c | S.null c  = True
           | otherwise = p x && everyC p c1
  where (x,c1) = deleteFindMin c

-- 2ª definición
everyC2 :: Ord a => (a -> Bool) -> Set a -> Bool
everyC2 p = S.foldr (\x r -> p x && r) True

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    someC :: Ord a => (a -> Bool) -> Set a -> Bool
-- tal que (someC p c) se verifica si algún elemento de c verifica el
-- predicado p. Por ejemplo,
--   someC even (fromList [1,4,7]) == True
--   someC even (fromList [1,3,7]) == False
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
someC :: Ord a => (a -> Bool) -> Set a -> Bool
someC p c | S.null c  = False
          | otherwise = p x || someC p c1
  where (x,c1) = deleteFindMin c

-- 2ª definición
someC2 :: Ord a => (a -> Bool) -> Set a -> Bool
someC2 p = S.foldr (\x r -> p x || r) False

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    productoC :: (Ord a, Ord b) => Set a -> Set b -> Set (a,b)
-- tal que (productoC c1 c2) es el producto cartesiano de los
-- conjuntos c1 y c2. Por ejemplo,
--    ghci> productoC (fromList [1,3]) (fromList [2,4])
--    fromList [(1,2),(1,4),(3,2),(3,4)]
-- ---------------------------------------------------------------------

productoC :: (Ord a, Ord b) => Set a -> Set b -> Set (a,b)
productoC c1 c2 =
  fromList [(x,y) | x <- elems c1, y <- elems c2]

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    potencia :: Ord a => Set a -> Set (Set a)
-- tal que (potencia c) es el conjunto potencia de c; es decir, el
-- conjunto de todos los subconjuntos de c. Por ejemplo,
--    ghci> potencia (fromList [1..3])
--    fromList [fromList [],fromList [1],fromList [1,2],fromList [1,2,3],
--              fromList [1,3],fromList [2],fromList [2,3],fromList [3]]
-- ---------------------------------------------------------------------

potencia :: Ord a => Set a -> Set (Set a)
potencia c | S.null c  = singleton empty
           | otherwise = S.map (insert x) pr `union` pr
    where (x,rc) = deleteFindMin c
          pr     = potencia rc
```

# Relación 29

# Relaciones binarias homogéneas con la librería Data.Set

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

-- El objetivo de esta relación de ejercicios es definir propiedades y
-- operaciones sobre las relaciones binarias (homogéneas) usando la
-- librería Data.Set.
--
-- Como referencia se puede usar el artículo de la wikipedia
-- http://bit.ly/HVHOPS

-- ----------------------------------------------------------------------
-- § Librerías auxiliares                                               --
-- ----------------------------------------------------------------------

import Test.QuickCheck
import Data.Set as S

-- ----------------------------------------------------------------------
-- Ejercicio 1. Una relación binaria S sobre un conjunto A se puede
-- representar mediante la expresión (R xs ps) donde xs es el conjunto
-- de los elementos de A (el universo de S) y ps es el conjunto de pares
-- de S (el grafo de S). Definir el tipo de dato (Rel a) para
-- representar las relaciones binarias sobre a.
-- ----------------------------------------------------------------------
```

```haskell
data Rel a = R (Set a) (Set (a,a))
  deriving Show

-- ---------------------------------------------------------------------
-- Nota. En los ejemplos usaremos las siguientes relaciones binarias:
--    r1, r2, r3 :: Rel Int
--    r1 = R (fromList [1..9]) (fromList [(1,3), (2,6), (8,9), (2,7)])
--    r2 = R (fromList [1..9]) (fromList [(1,3), (2,6), (8,9), (3,7)])
--    r3 = R (fromList [1..9]) (fromList [(1,3), (2,6), (8,9), (3,6)])
-- ---------------------------------------------------------------------

r1, r2, r3 :: Rel Int
r1 = R (fromList [1..9]) (fromList [(1,3), (2,6), (8,9), (2,7)])
r2 = R (fromList [1..9]) (fromList [(1,3), (2,6), (8,9), (3,7)])
r3 = R (fromList [1..9]) (fromList [(1,3), (2,6), (8,9), (3,6)])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    universo :: Ord a => Rel a -> Set a
-- tal que (universo r) es el universo de la relación r. Por ejemplo,
--    universo r1  ==  fromList [1,2,3,4,5,6,7,8,9]
-- ---------------------------------------------------------------------

universo :: Ord a => Rel a -> Set a
universo (R u _) = u

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    grafo :: Ord a => Rel a -> [(a,a)]
-- tal que (grafo r) es el grafo de la relación r. Por ejemplo,
--    grafo r1  ==  fromList [(1,3),(2,6),(2,7),(8,9)]
-- ---------------------------------------------------------------------

grafo :: Ord a => Rel a -> Set (a,a)
grafo (R _ g) = g

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    reflexiva :: Ord a => Rel a -> Bool
-- tal que (reflexiva r) se verifica si la relación r es reflexiva. Por
```

```
-- ejemplo,
--    ghci> reflexiva (R (fromList [1,3]) (fromList [(1,1),(1,3),(3,3)]))
--    True
--    ghci> reflexiva (R (fromList [1,2,3]) (fromList [(1,1),(1,3),(3,3)]))
--    False
-- ----------------------------------------------------------------------

reflexiva :: Ord a => Rel a -> Bool
reflexiva (R u g) = and [(x,x) `member` g | x <- elems u]

-- ----------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    simetrica :: Ord a => Rel a -> Bool
-- tal que (simetrica r) se verifica si la relación r es simétrica. Por
-- ejemplo,
--    ghci> simetrica (R (fromList [1,3]) (fromList [(1,1),(1,3),(3,1)]))
--    True
--    ghci> simetrica (R (fromList [1,3]) (fromList [(1,1),(1,3),(3,2)]))
--    False
--    ghci> simetrica (R (fromList [1,3]) (fromList []))
--    True
-- ----------------------------------------------------------------------

simetrica :: Ord a => Rel a -> Bool
simetrica (R _ g) = and [(y,x) `member` g | (x,y) <- elems g]

-- ----------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    subconjunto :: Ord a => Set a -> Set a -> Bool
-- tal que (subconjunto c1 c2) se verifica si c1 es un subconjunto de
-- c2. Por ejemplo,
--    subconjunto (fromList [1,3]) (fromList [3,1,5])  ==  True
--    subconjunto (fromList [3,1,5]) (fromList [1,3])  ==  False
-- ----------------------------------------------------------------------

subconjunto :: Ord a => Set a -> Set a -> Bool
subconjunto = isSubsetOf

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

```
--    composicion :: Ord a => Rel a -> Rel a -> Rel a
-- tal que (composicion r s) es la composición de las relaciones r y
-- s. Por ejemplo,
--    ghci> let r1 = (R (fromList [1,2]) (fromList [(1,2),(2,2)]))
--    ghci> let r2 = (R (fromList [1,2]) (fromList [(2,1)]))
--    ghci> let r3 = (R (fromList [1,2]) (fromList [(1,1)]))
--    ghci> composicion r1 r2
--    R (fromList [1,2]) (fromList [(1,1),(2,1)])
--    ghci> composicion r1 r3
--    R (fromList [1,2,3,4,5,6,7,8,9]) (fromList [])
-- ---------------------------------------------------------------------

composicion :: Ord a => Rel a -> Rel a -> Rel a
composicion (R u g1) (R _ g2) =
  R u (fromList [(x,z) | (x,y1) <- elems g1,
                         (y2,z) <- elems g2,
                         y1 == y2])

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    transitiva :: Ord a => Rel a -> Bool
-- tal que (transitiva r) se verifica si la relación r es transitiva.
-- Por ejemplo,
-- ghci> transitiva (R (fromList [1,3,5])
--                     (fromList [(1,1),(1,3),(3,1),(3,3),(5,5)]))
--  True
--  ghci> transitiva (R (fromList [1,3,5])
--                      (fromList [(1,1),(1,3),(3,1),(5,5)]))
--  False
-- ---------------------------------------------------------------------

transitiva :: Ord a => Rel a -> Bool
transitiva r@(R _ g) =
  isSubsetOf (grafo (composicion r r)) g

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    esEquivalencia :: Ord a => Rel a -> Bool
-- tal que (esEquivalencia r) se verifica si la relación r es de
-- equivalencia. Por ejemplo,
```

```
--    ghci> esEquivalencia (R (fromList [1,3,5])
--                            (fromList [(1,1),(1,3),(3,1),(3,3),(5,5)]))
--    True
--    ghci> esEquivalencia (R (fromList [1,2,3,5])
--                            (fromList [(1,1),(1,3),(3,1),(3,3),(5,5)]))
--    False
--    ghci> esEquivalencia (R (fromList [1,3,5])
--                            (fromList [(1,1),(1,3),(3,3),(5,5)]))
--    False
-- ---------------------------------------------------------------------

esEquivalencia :: Ord a => Rel a -> Bool
esEquivalencia r = reflexiva r && simetrica r && transitiva r

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    irreflexiva :: Ord a => Rel a -> Bool
-- tal que (irreflexiva r) se verifica si la relación r es irreflexiva;
-- es decir, si ningún elemento de su universo está relacionado con
-- él mismo. Por ejemplo,
--    ghci> irreflexiva (R (fromList [1,2,3]) (fromList [(1,2),(2,1),(2,3)]))
--    True
--    ghci> irreflexiva (R (fromList [1,2,3]) (fromList [(1,2),(2,1),(3,3)]))
--    False
-- ---------------------------------------------------------------------

irreflexiva :: Ord a => Rel a -> Bool
irreflexiva (R u g) = and [(x,x) `notMember` g | x <- elems u]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    antisimetrica :: Ord a => Rel a -> Bool
-- tal que (antisimetrica r) se verifica si la relación r es
-- antisimétrica; es decir, si (x,y) e (y,x) están relacionado, entonces
-- x=y. Por ejemplo,
--    antisimetrica (R (fromList [1,2]) (fromList [(1,2)]))       == True
--    antisimetrica (R (fromList [1,2]) (fromList [(1,2),(2,1)])) == False
--    antisimetrica (R (fromList [1,2]) (fromList [(1,1),(2,1)])) == True
-- ---------------------------------------------------------------------
```

```haskell
antisimetrica :: Ord a => Rel a -> Bool
antisimetrica (R _ g) =
  [(x,y) | (x,y) <- elems g, x /= y, (y,x) `member` g] == []

-- Otra definición es
antisimetrica2 :: Ord a => Rel a -> Bool
antisimetrica2 (R u g) =
  and [ ((x,y) `member` g && (y,x) `member` g) --> (x == y)
      | x <- elems u, y <- elems u]
  where p --> q = not p || q

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    esTotal :: Ord a => Rel a -> Bool
-- tal que (esTotal r) se verifica si la relación r es total; es decir, si
-- para cualquier par x, y de elementos del universo de r, se tiene que
-- x está relacionado con y ó y está relacionado con x. Por ejemplo,
--    esTotal (fromList [1,3],fromList [(1,1),(3,1),(3,3)])  ==  True
--    esTotal (fromList [1,3],fromList [(1,1),(3,1)])        ==  False
--    esTotal (fromList [1,3],fromList [(1,1),(3,3)])        ==  False
-- ---------------------------------------------------------------------

esTotal :: Ord a => Rel a -> Bool
esTotal (R u g) =
  and [(x,y) `member` g || (y,x) `member` g | x <- xs, y <- xs]
  where xs = elems u

-- ---------------------------------------------------------------------
-- Ejercicio 13. Comprobar con QuickCheck que las relaciones totales son
-- reflexivas.
-- ---------------------------------------------------------------------

prop_total_reflexiva :: Rel Int -> Property
prop_total_reflexiva r =
  esTotal r ==> reflexiva r

-- La comprobación es
--    ghci> quickCheck prop_total_reflexiva
--    *** *** Gave up! Passed only 77 tests.
```

```
-- ---------------------------------------------------------------------
-- § Clausuras                                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    clausuraReflexiva :: Ord a => Rel a -> Rel a
-- tal que (clausuraReflexiva r) es la clausura reflexiva de r; es
-- decir, la menor relación reflexiva que contiene a r. Por ejemplo,
--    ghci> clausuraReflexiva (fromList [1,3], fromList [(1,1),(3,1)])
--    (fromList [1,3],fromList [(1,1),(3,1),(3,3)])
-- ---------------------------------------------------------------------

clausuraReflexiva :: Ord a => Rel a -> Rel a
clausuraReflexiva (R u g) =
  R u (g `union` fromList [(x,x) | x <- elems u])

-- ---------------------------------------------------------------------
-- Ejercicio 15. Comprobar con QuickCheck que clausuraReflexiva es
-- reflexiva.
-- ---------------------------------------------------------------------

prop_ClausuraReflexiva :: Rel Int -> Bool
prop_ClausuraReflexiva r =
  reflexiva (clausuraReflexiva r)

-- La comprobación es
--    ghci> quickCheck prop_ClausuraReflexiva
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    clausuraSimetrica :: Ord a => Rel a -> Rel a
-- tal que (clausuraSimetrica r) es la clausura simétrica de r; es
-- decir, la menor relación simétrica que contiene a r. Por ejemplo,
--    ghci> clausuraSimetrica (R (fromList [1,3,5])
--                               (fromList [(1,1),(3,1),(1,5)]))
--    R (fromList [1,3,5]) (fromList [(1,1),(1,3),(1,5),(3,1),(5,1)])
-- ---------------------------------------------------------------------
```

```haskell
clausuraSimetrica :: Ord a => Rel a -> Rel a
clausuraSimetrica (R u g) =
  R u (g `union` fromList [(y,x) | (x,y) <- elems g])

-- ---------------------------------------------------------------------
-- Ejercicio 17. Comprobar con QuickCheck que clausuraSimetrica es
-- simétrica.
-- ---------------------------------------------------------------------

prop_ClausuraSimetrica :: Rel Int -> Bool
prop_ClausuraSimetrica r =
  simetrica (clausuraSimetrica r)

-- La comprobación es
--    ghci> quickCheck prop_ClausuraSimetrica
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    clausuraTransitiva :: Ord a => Rel a -> Rel a
-- tal que (clausuraTransitiva r) es la clausura transitiva de r; es
-- decir, la menor relación transitiva que contiene a r. Por ejemplo,
--    ghci> clausuraTransitiva (R (fromList [1..6])
--                                (fromList [(1,2),(2,5),(5,6)]))
--    R (fromList [1,2,3,4,5,6])
--      (fromList [(1,2),(1,5),(1,6),(2,5),(2,6),(5,6)])
-- ---------------------------------------------------------------------

clausuraTransitiva :: Ord a => Rel a -> Rel a
clausuraTransitiva (R u g) = R u (aux g)
  where aux r | cerradoTr r = r
              | otherwise   = aux (r `union` comp r r)
        cerradoTr r = isSubsetOf (comp r r) r
        comp r s    = fromList [(x,z) | (x,y1) <- elems r,
                                        (y2,z) <- elems s,
                                        y1 == y2]

-- ---------------------------------------------------------------------
-- Ejercicio 19. Comprobar con QuickCheck que clausuraTransitiva es
-- transitiva.
```

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

prop_ClausuraTransitiva :: Rel Int -> Bool
prop_ClausuraTransitiva r =
  transitiva (clausuraTransitiva r)

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_ClausuraTransitiva
--    +++ OK, passed 100 tests.

-- ----------------------------------------------------------------------
-- § Generador de relaciones                                            --
-- ----------------------------------------------------------------------

-- genSet es un generador de relaciones binarias. Por ejemplo,
--    ghci> sample genRel
--    (fromList [0],fromList [])
--    (fromList [-1,1],fromList [(-1,1)])
--    (fromList [-3,-2],fromList [])
--    (fromList [-2,0,1,6],fromList [(0,0),(6,0)])
--    (fromList [-7,0,2],fromList [(-7,0),(2,0)])
--    (fromList [2,11],fromList [(2,2),(2,11),(11,2),(11,11)])
--    (fromList [-4,-2,1,4,5],fromList [(1,-2),(1,1),(1,5)])
--    (fromList [-4,-3,-2,6,7],fromList [(-3,-4),(7,-3),(7,-2)])
--    (fromList [-9,-7,0,10],fromList [(10,-9)])
--    (fromList [-10,3,8,10],fromList [(3,3),(10,-10)])
--    (fromList [-10,-9,-7,-6,-5,-4,-2,8,12],fromList [])
genRel :: (Arbitrary a, Integral a) => Gen (Rel a)
genRel = do
  xs <- listOf1 arbitrary
  ys <- listOf (elements [(x,y) | x <- xs, y <- xs])
  return (R (fromList xs) (fromList ys))

instance (Arbitrary a, Integral a) => Arbitrary (Rel a) where
  arbitrary = genRel
```

# Relación 30

# Operaciones con el TAD de montículos

```haskell
module Rel_30_sol where

-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las montículos, utilizando las implementaciones estudiadas
-- en el tema 20 que se encuenta en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-20.html
--
-- Además, hay que instalar las librerías de I1M que se encuentra en
-- http://hackage.haskell.org/package/I1M
--
-- Para instalar la librería de I1M, basta ejecutar en una consola
--    cabal update
--    cabal install I1M
--
-- Otra forma es descargar la implementación del TAD de montículos:
-- + Monticulo.hs que está en http://bit.ly/1oNy2HT

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

```haskell
import Test.QuickCheck

-- Hay que elegir una implementación del TAD montículos:
import I1M.Monticulo
-- import Monticulo

-- ---------------------------------------------------------------------
-- Ejemplos                                                            --
-- ---------------------------------------------------------------------

-- Para los ejemplos se usarán los siguientes montículos.
m1, m2, m3 :: Monticulo Int
m1 = foldr inserta vacio [6,1,4,8]
m2 = foldr inserta vacio [7,5]
m3 = foldr inserta vacio [6,1,4,8,7,5]

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    numeroDeNodos :: Ord a => Monticulo a -> Int
-- tal que (numeroDeNodos m) es el número de nodos del montículo m. Por
-- ejemplo,
--    numeroDeNodos m1  ==  4
-- ---------------------------------------------------------------------

numeroDeNodos :: Ord a => Monticulo a -> Int
numeroDeNodos m
  | esVacio m = 0
  | otherwise = 1 + numeroDeNodos (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
-- tal que (filtra p m) es el montículo con los nodos del montículo m
-- que cumplen la propiedad p. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> filtra even m1
--    M 4 1 (M 6 1 (M 8 1 Vacio Vacio) Vacio) Vacio
--    ghci> filtra odd m1
--    M 1 1 Vacio Vacio
```

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

filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
filtra p m
  | esVacio m = vacio
  | p mm      = inserta mm (filtra p rm)
  | otherwise = filtra p rm
  where mm = menor m
        rm = resto m

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    menores :: Ord a => Int -> Monticulo a -> [a]
-- tal que (menores n m) es la lista de los n menores elementos del
-- montículo m. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> menores 3 m1
--    [1,4,6]
--    ghci> menores 10 m1
--    [1,4,6,8]
-- ---------------------------------------------------------------------

menores :: Ord a => Int -> Monticulo a -> [a]
menores 0 _  = []
menores n m
  | esVacio m = []
  | otherwise = menor m : menores (n-1) (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    restantes :: Ord a => Int -> Monticulo a -> Monticulo a
-- tal que (restantes n m) es el montículo obtenido eliminando los n
-- menores elementos del montículo m. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> restantes 3 m1
--    M 8 1 Vacio Vacio
--    ghci> restantes 2 m1
--    M 6 1 (M 8 1 Vacio Vacio) Vacio
```

```haskell
--    ghci> restantes 7 m1
--    Vacio
-- ---------------------------------------------------------------------

restantes :: Ord a => Int -> Monticulo a -> Monticulo a
restantes 0 m  = m
restantes n m
  | esVacio m = vacio
  | otherwise = restantes (n-1) (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    lista2Monticulo :: Ord a => [a] -> Monticulo a
-- tal que (lista2Monticulo xs) es el montículo cuyos nodos son los
-- elementos de la lista xs. Por ejemplo,
--    ghci> lista2Monticulo [2,5,3,7]
--    M 2 1 (M 3 2 (M 7 1 Vacio Vacio) (M 5 1 Vacio Vacio)) Vacio
-- ---------------------------------------------------------------------

lista2Monticulo :: Ord a => [a] -> Monticulo a
lista2Monticulo = foldr inserta vacio

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    monticulo2Lista :: Ord a => Monticulo a -> [a]
-- tal que (monticulo2Lista m) es la lista ordenada de los nodos del
-- montículo m. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> monticulo2Lista m1
--    [1,4,6,8]
-- ---------------------------------------------------------------------

monticulo2Lista :: Ord a => Monticulo a -> [a]
monticulo2Lista m
  | esVacio m = []
  | otherwise = menor m : monticulo2Lista (resto m)

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

```
--    ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada de
-- forma creciente. Por ejemplo,
--    ordenada [3,5,9]  ==  True
--    ordenada [3,5,4]  ==  False
--    ordenada [7,5,4]  ==  False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8. Comprobar con QuickCheck que para todo montículo m,
-- (monticulo2Lista m) es una lista ordenada creciente.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_monticulo2Lista_ordenada :: Monticulo Int -> Bool
prop_monticulo2Lista_ordenada m =
  ordenada (monticulo2Lista m)

-- La comprobación es
--    ghci> quickCheck prop_monticulo2Lista_ordenada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10. Usando monticulo2Lista y lista2Monticulo, definir la
-- función
--    ordena :: Ord a => [a] -> [a]
-- tal que (ordena xs) es la lista obtenida ordenando de forma creciente
-- los elementos de xs. Por ejemplo,
--    ordena [7,5,3,6,5]  ==  [3,5,5,6,7]
-- ---------------------------------------------------------------------

ordena :: Ord a => [a] -> [a]
ordena  = monticulo2Lista . lista2Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 11. Comprobar con QuickCheck que para toda lista xs,
```

```haskell
-- (ordena xs) es una lista ordenada creciente.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordena_ordenada :: [Int] -> Bool
prop_ordena_ordenada xs =
  ordenada (ordena xs)

-- La comprobación es
--    ghci> quickCheck prop_ordena_ordenada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
-- x en la lista xs. Por ejemplo,
--    borra 1 [1,2,1]  ==  [2,1]
--    borra 3 [1,2,1]  ==  [1,2,1]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función esPermutación tal que
-- (esPermutación xs ys) se verifique si xs es una permutación de
-- ys. Por ejemplo,
--    esPermutación [1,2,1] [2,1,1]  ==  True
--    esPermutación [1,2,1] [1,2,2]  ==  False
-- ---------------------------------------------------------------------

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

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

```haskell
-- Ejercicio 15. Comprobar con QuickCheck que para toda lista xs,
-- (ordena xs) es una permutación de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordena_permutacion :: [Int] -> Bool
prop_ordena_permutacion xs =
  esPermutacion (ordena xs) xs

-- La comprobación es
--    ghci> quickCheck prop_ordena_permutacion
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Generador de montículos                                           --
-- ---------------------------------------------------------------------

-- genMonticulo es un generador de montículos. Por ejemplo,
--    ghci> sample genMonticulo
--    VacioM
--    M (-1) 1 (M 1 1 VacioM VacioM) VacioM
--    ...
genMonticulo :: Gen (Monticulo Int)
genMonticulo = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Montículo es una instancia de la clase arbitraria.
instance Arbitrary (Monticulo Int) where
  arbitrary = genMonticulo
```

# Relación 31

# Operaciones con el TAD de polinomios

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

-- El objetivo de esta relación es ampliar el conjunto de operaciones
-- sobre polinomios definidas utilizando las implementaciones del TAD de
-- polinomio estudiadas en el tema 21
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-21.html
--
-- Además, en algunos ejemplos de usan polinomios con coeficientes
-- racionales. En Haskell, el número racional x/y se representa por
-- x%y. El TAD de los números racionales está definido en el módulo
-- Data.Ratio.
--
-- Para realizar los ejercicios hay que tener instalada la librería I1M
-- que se encuentra en http://hackage.haskell.org/package/I1M
--
-- Para instalar la librería de I1M, basta ejecutar en una consola
--    cabal update
--    cabal install I1M
--
-- Otra forma es descargar, en el directorio de ejercicios, la
-- implementación del TAD de polinomios:
-- + PolRepTDA      que está en http://bit.ly/1WJnS93
-- + PolRepDispersa que está en http://bit.ly/1WJnUO8
-- + PolRepDensa    que está en http://bit.ly/1WJnV4E
```

```
-- + PolOperaciones que está en http://bit.ly/1WJnTd7

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

import Data.Ratio

-- Hay que elegir una librería
import I1M.PolOperaciones
-- import PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
-- tal que (creaPolDispersa xs) es el polinomio cuya representación
-- dispersa es xs. Por ejemplo,
--    creaPolDispersa [7,0,0,4,0,3]  ==  7*x^5 + 4*x^2 + 3
-- ---------------------------------------------------------------------

creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
creaPolDispersa []     = polCero
creaPolDispersa (x:xs) = consPol (length xs) x (creaPolDispersa xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    creaPolDensa :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
-- tal que (creaPolDensa xs) es el polinomio cuya representación
-- densa es xs. Por ejemplo,
--    creaPolDensa [(5,7),(4,2),(3,0)]  ==  7*x^5 + 2*x^4
-- ---------------------------------------------------------------------

creaPolDensa :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
creaPolDensa []         = polCero
creaPolDensa ((n,a):ps) = consPol n a (creaPolDensa ps)

-- 2ª definición
creaPolDensa2 :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
creaPolDensa2 = foldr (\(x,y) -> consPol x y) polCero
```

```haskell
-- 3ª definición
creaPolDensa3 :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
creaPolDensa3 = foldr (uncurry consPol) polCero

-- ---------------------------------------------------------------------
-- Nota. En el resto de la relación se usará en los ejemplos los
-- los polinomios que se definen a continuación.
-- ---------------------------------------------------------------------

pol1, pol2, pol3 :: (Num a, Eq a) => Polinomio a
pol1 = creaPolDensa [(5,1),(2,5),(1,4)]
pol2 = creaPolDispersa [2,3]
pol3 = creaPolDensa [(7,2),(4,5),(2,5)]


pol4, pol5, pol6 :: Polinomio Rational
pol4 = creaPolDensa [(4,3%1),(2,5),(0,3)]
pol5 = creaPolDensa [(2,6),(1,2)]
pol6 = creaPolDensa [(2,8),(1,14),(0,3)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    densa :: (Num a, Eq a) => Polinomio a -> [(Int,a)]
-- tal que (densa p) es la representación densa del polinomio p. Por
-- ejemplo,
--    pol1        ==  x^5 + 5*x^2 + 4*x
--    densa pol1  ==  [(5,1),(2,5),(1,4)]
-- ---------------------------------------------------------------------

densa :: (Num a, Eq a) => Polinomio a -> [(Int,a)]
densa p | esPolCero p = []
        | otherwise   = (grado p, coefLider p) : densa (restoPol p)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    densaAdispersa :: Num a => [(Int,a)] -> [a]
-- tal que (densaAdispersa ps) es la representación dispersa del
-- polinomio cuya representación densa es ps. Por ejemplo,
--    densaAdispersa [(5,1),(2,5),(1,4)]  ==  [1,0,0,5,4,0]
-- ---------------------------------------------------------------------
```

```haskell
densaAdispersa :: Num a => [(Int,a)] -> [a]
densaAdispersa []     = []
densaAdispersa [(n,a)] = a : replicate n 0
densaAdispersa ((n,a):(m,b):ps) =
  a : replicate (n-m-1) 0 ++ densaAdispersa ((m,b):ps)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    dispersa :: (Num a, Eq a) => Polinomio a -> [a]
-- tal que (dispersa p) es la representación dispersa del polinomio
-- p. Por ejemplo,
--    pol1           ==  x^5 + 5*x^2 + 4*x
--    dispersa pol1  ==  [1,0,0,5,4,0]
-- ---------------------------------------------------------------------

dispersa :: (Num a, Eq a) => Polinomio a -> [a]
dispersa = densaAdispersa . densa

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
-- tal que (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--    pol1                ==  x^5 + 5*x^2 + 4*x
--    coeficiente 2 pol1  ==  5
--    coeficiente 3 pol1  ==  0
-- ---------------------------------------------------------------------

coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
coeficiente k p | k == n                = coefLider p
                | k > grado (restoPol p) = 0
                | otherwise              = coeficiente k (restoPol p)
  where n = grado p

-- Otra definición equivalente es
coeficiente' :: (Num a, Eq a) => Int -> Polinomio a -> a
coeficiente' k p = busca k (densa p)
  where busca k1 ps = head ([a | (n,a) <- ps, n == k1] ++ [0])
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de los coeficientes del
-- polinomio p. Por ejemplo,
--    pol1               ==  x^5 + 5*x^2 + 4*x
--    coeficientes pol1  ==  [1,0,0,5,4,0]
-- ---------------------------------------------------------------------

coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <- [n,n-1..0]]
  where n = grado p

-- 2ª definición
coeficientes2 :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes2 = dispersa

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potencia p n) es la potencia n-ésima del polinomio p. Por
-- ejemplo,
--    pol2             ==  2*x + 3
--    potencia pol2 2  ==  4*x^2 + 12*x + 9
--    potencia pol2 3  ==  8*x^3 + 36*x^2 + 54*x + 27
-- ---------------------------------------------------------------------

potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potencia _ 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))

-- ---------------------------------------------------------------------
-- Ejercicio 9. Mejorar la definición de potencia definiendo la función
--    potenciaM :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potenciaM p n) es la potencia n-ésima del polinomio p,
-- utilizando las siguientes propiedades:
--    * Si n es par,   entonces x^n = (x^2)^(n/2)
--    * Si n es impar, entonces x^n = x * (x^2)^((n-1)/2)
-- Por ejemplo,
--    pol2              ==  2*x + 3
```

```
--    potenciaM pol2 2  ==  4*x^2 + 12*x + 9
--    potenciaM pol2 3  ==  8*x^3 + 36*x^2 + 54*x + 27
-- ---------------------------------------------------------------------

potenciaM :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potenciaM _ 0 = polUnidad
potenciaM p n
  | even n    = potenciaM (multPol p p) (n `div` 2)
  | otherwise = multPol p (potenciaM (multPol p p) ((n-1) `div` 2))

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
-- tal que (integral p) es la integral del polinomio p cuyos coefientes
-- son números racionales. Por ejemplo,
--    ghci> pol3
--    2*x^7 + 5*x^4 + 5*x^2
--    ghci> integral pol3
--    0.25*x^8 + x^5 + 1.6666666666666667*x^3
--    ghci> integral pol3 :: Polinomio Rational
--    1 % 4*x^8 + x^5 + 5 % 3*x^3
-- ---------------------------------------------------------------------

integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
integral p
  | esPolCero p = polCero
  | otherwise   = consPol (n+1) (b / fromIntegral (n+1)) (integral r)
  where n = grado p
        b = coefLider p
        r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
-- tal que (integralDef p a b) es la integral definida del polinomio p
-- cuyos coefientes son números racionales. Por ejemplo,
--    ghci> integralDef pol3 0 1
--    2.916666666666667
--    ghci> integralDef pol3 0 1 :: Rational
--    35 % 12
```

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

integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
integralDef p a b = valor q b - valor q a
  where q = integral p

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
-- tal que (multEscalar c p) es el polinomio obtenido multiplicando el
-- número c por el polinomio p. Por ejemplo,
--    pol2                    ==  2*x + 3
--    multEscalar 4 pol2      ==  8*x + 12
--    multEscalar (1%4) pol2  ==  1 % 2*x + 3 % 4
-- ---------------------------------------------------------------------

multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
multEscalar c p
  | esPolCero p = polCero
  | otherwise   = consPol n (c*b) (multEscalar c r)
  where n = grado p
        b = coefLider p
        r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    cociente:: (Fractional a, Eq a) =>
--               Polinomio a -> Polinomio a -> Polinomio a
-- tal que (cociente p q) es el cociente de la división de p entre
-- q. Por ejemplo,
--    pol4  ==  3 % 1*x^4 + 5 % 1*x^2 + 3 % 1
--    pol5  ==  6 % 1*x^2 + 2 % 1*x
--    cociente pol4 pol5  ==  1 % 2*x^2 + (-1) % 6*x + 8 % 9
-- ---------------------------------------------------------------------

cociente :: (Fractional a, Eq a) =>
            Polinomio a -> Polinomio a -> Polinomio a
cociente p q
  | n2 == 0   = multEscalar (1/a2) p
  | n1 < n2   = polCero
```

```
  | otherwise = consPol n3 a3 (cociente p3 q)
  where n1 = grado p
        a1 = coefLider p
        n2 = grado q
        a2 = coefLider q
        n3 = n1-n2
        a3 = a1/a2
        p3 = restaPol p (multPorTerm (creaTermino n3 a3) q)

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    resto:: (Fractional a, Eq a) =>
--            Polinomio a -> Polinomio a -> Polinomio a
-- tal que (resto p q) es el resto de la división de p entre q. Por
-- ejemplo,
--    pol4  ==  3 % 1*x^4 + 5 % 1*x^2 + 3 % 1
--    pol5  ==  6 % 1*x^2 + 2 % 1*x
--    resto pol4 pol5  ==  (-16) % 9*x + 3 % 1
-- ---------------------------------------------------------------------

resto :: (Fractional a, Eq a) =>
         Polinomio a -> Polinomio a -> Polinomio a
resto p q = restaPol p (multPol (cociente p q) q)

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    divisiblePol :: (Fractional a, Eq a) =>
--                    Polinomio a -> Polinomio a -> Bool
-- tal que (divisiblePol p q) se verifica si el polinomio p es divisible
-- por el polinomio q. Por ejemplo,
--    pol6  ==  8 % 1*x^2 + 14 % 1*x + 3 % 1
--    pol2  ==  2*x + 3
--    pol5  ==  6 % 1*x^2 + 2 % 1*x
--    divisiblePol pol6 pol2  ==  True
--    divisiblePol pol6 pol5  ==  False
-- ---------------------------------------------------------------------

divisiblePol :: (Fractional a, Eq a) =>
                Polinomio a -> Polinomio a -> Bool
divisiblePol p q = esPolCero (resto p q)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 16. El método de Horner para calcular el valor de un
-- polinomio se basa en representarlo de una forma forma alernativa. Por
-- ejemplo, para calcular el valor de
--    a*x^5 + b*x^4 + c*x^3 + d*x^2 + e*x + f
-- se representa como
--   (((((0 * x + a) * x + b) * x + c) * x + d) * x + e) * x + f
-- y se evalúa de dentro hacia afuera; es decir,
--   v(0) = 0
--   v(1) = v(0)*x+a = 0*x+a = a
--   v(2) = v(1)*x+b = a*x+b
--   v(3) = v(2)*x+c = (a*x+b)*x+c = a*x^2+b*x+c
--   v(4) = v(3)*x+d = (a*x^2+b*x+c)*x+d = a*x^3+b*x^2+c*x+d
--   v(5) = v(4)*x+e = (a*x^3+b*x^2+c*x+d)*x+e = a*x^4+b*x^3+c*x^2+d*x+e
--   v(6) = v(5)*x+f = (a*x^4+b*x^3+c*x^2+d*x+e)*x+f = a*x^5+b*x^4+c*x^3+d*x^2+e*
--
-- Definir la función
--    horner :: (Num a, Eq a) => Polinomio a -> a -> a
-- tal que (horner p x) es el valor del polinomio p al sustituir su
-- variable por el número x. Por ejemplo,
--    horner pol1 0     ==  0
--    horner pol1 1     ==  10
--    horner pol1 1.5   ==  24.84375
--    horner pol1 (3%2) ==  795 % 32
-- ---------------------------------------------------------------------

horner :: (Num a, Eq a) => Polinomio a -> a -> a
horner p x = hornerAux (coeficientes p) 0
  where hornerAux [] v     = v
        hornerAux (a:as) v = hornerAux as (v*x+a)

-- El cálculo de (horner pol1 2) es el siguiente
--    horner pol1 2
--    = hornerAux [1,0,0,5,4,0] 0
--    = hornerAux   [0,0,5,4,0] ( 0*2+1) = hornerAux   [0,0,5,4,0] 1
--    = hornerAux     [0,5,4,0] ( 1*2+0) = hornerAux     [0,5,4,0] 2
--    = hornerAux       [5,4,0] ( 2*2+0) = hornerAux       [5,4,0] 4
--    = hornerAux         [4,0] ( 4*2+5) = hornerAux         [4,0] 13
--    = hornerAux           [0] (13*2+4) = hornerAux           [0] 30
```

```haskell
--     = hornerAux            [] (30*2+0) = hornerAux            [] 60

-- Una defininición equivalente por plegado es
horner2 :: (Num a, Eq a) => Polinomio a -> a -> a
horner2 p x = foldl (\a b -> a*x + b) 0 (coeficientes p)
```

# Relación 32

# División y factorización de polinomios mediante la regla de Ruffini

```haskell
module Rel_32_sol where

-- ----------------------------------------------------------------------
-- Introducción                                                         --
-- ----------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es implementar la regla de
-- Ruffini y sus aplicaciones utilizando las implementaciones del TAD de
-- polinomio estudiadas en el tema 21 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-21.html
--
-- Para realizar los ejercicios hay que tener instalada la librería I1M
-- que se encuentra en http://hackage.haskell.org/package/I1M
--
-- Para instalar la librería de I1M, basta ejecutar en una consola
--    cabal update
--    cabal install I1M
--
-- Otra forma es descargar, en el directorio de ejercicios, la
-- implementación del TAD de polinomios:
-- + PolRepTDA      que está en http://bit.ly/1WJnS93
-- + PolRepDispersa que está en http://bit.ly/1WJnUO8
-- + PolRepDensa    que está en http://bit.ly/1WJnV4E
-- + PolOperaciones que está en http://bit.ly/1WJnTd7
```

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

import Test.QuickCheck

-- Hay que elegir una librería
import I1M.PolOperaciones
-- import PolOperaciones

-- ---------------------------------------------------------------------
-- Ejemplos                                                          --
-- ---------------------------------------------------------------------

ejPol1, ejPol2, ejPol3, ejPol4 :: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)
ejPol4 = consPol 3 1
                 (consPol 2 2
                          (consPol 1 (-1)
                                   (consPol 0 (-2) polCero)))

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    divisores :: Int -> [Int]
-- tal que (divisores n) es la lista de todos los divisores enteros de
-- n. Por ejemplo,
--    divisores 4     ==  [1,-1,2,-2,4,-4]
--    divisores (-6)  ==  [1,-1,2,-2,3,-3,6,-6]
-- ---------------------------------------------------------------------

divisores :: Int -> [Int]
divisores n = concat [[x,-x] | x <- [1..abs n], rem n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
-- tal que (coeficiente k p) es el coeficiente del término de grado k en
```

```haskell
-- p. Por ejemplo:
--     coeficiente 4 ejPol1 == 3
--     coeficiente 3 ejPol1 == 0
--     coeficiente 2 ejPol1 == -5
--     coeficiente 5 ejPol1 == 0
-- ---------------------------------------------------------------------

coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
coeficiente k p | k == gp      = coefLider p
                | k > grado rp = 0
                | otherwise    = coeficiente k rp
  where gp = grado p
        rp = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    terminoIndep :: (Num a, Eq a) => Polinomio  a -> a
-- tal que (terminoIndep p) es el término independiente del polinomio
-- p. Por ejemplo,
--    terminoIndep ejPol1 == 3
--    terminoIndep ejPol2 == 0
--    terminoIndep ejPol4 == -2
-- ---------------------------------------------------------------------

terminoIndep :: (Num a, Eq a) => Polinomio  a -> a
terminoIndep = coeficiente 0

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de coeficientes de p, ordenada
-- según el grado. Por ejemplo,
--     coeficientes ejPol1 == [3,0,-5,0,3]
--     coeficientes ejPol4 == [1,2,-1,-2]
--     coeficientes ejPol2 == [1,0,0,5,4,0]
-- ---------------------------------------------------------------------

coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <- [n,n-1..0]]
  where n = grado p
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    creaPol :: (Num a, Eq a) => [a] -> Polinomio a
-- tal que (creaPol cs) es el polinomio cuya lista de coeficientes es
-- cs. Por ejemplo,
--     creaPol [1,0,0,5,4,0] == x^5 + 5*x^2 + 4*x
--     creaPol [1,2,0,3,0]   == x^4 + 2*x^3 + 3*x
-- ---------------------------------------------------------------------

creaPol :: (Num a, Eq a) => [a] -> Polinomio a
creaPol []     = polCero
creaPol (a:as) = consPol n a (creaPol as)
  where n = length as

-- ---------------------------------------------------------------------
-- Ejercicio 6. Comprobar con QuickCheck que, dado un polinomio p, el
-- polinomio obtenido mediante creaPol a partir de la lista de
-- coeficientes de p coincide con p.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_coef :: Polinomio Int -> Bool
prop_coef p =
  creaPol (coeficientes p) == p

-- La comprobación es
--    ghci> quickCheck prop_coef
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir una función
--    pRuffini:: Int -> [Int] -> [Int]
-- tal que (pRuffini r cs) es la lista que resulta de aplicar un paso
-- del regla de Ruffini al número entero r y a la lista de coeficientes
-- cs. Por ejemplo,
--    pRuffini 2 [1,2,-1,-2] == [1,4,7,12]
--    pRuffini 1 [1,2,-1,-2] == [1,3,2,0]
-- ya que
--      | 1  2  -1  -2          | 1  2  -1  -2
```

```
--    2 |    2   8  14         1 |    1   3   2
--    --+--------------        --+-------------
--      | 1  4   7  12           | 1  3   2   0
-- ---------------------------------------------------------------------

-- 1ª definición
pRuffini :: Int -> [Int] -> [Int]
pRuffini r p@(c:cs) = c : [x+r*y | (x,y) <- zip cs (pRuffini r p)]

-- 2ª definición
pRuffini2 :: Int -> [Int] -> [Int]
pRuffini2 r = scanl1 (\s x -> s * r + x)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    cocienteRuffini:: Int -> Polinomio Int -> Polinomio Int
-- tal que (cocienteRuffini r p) es el cociente de dividir el polinomio
-- p por el polinomio x-r. Por ejemplo:
--    cocienteRuffini 2 ejPol4    == x^2 + 4*x + 7
--    cocienteRuffini (-2) ejPol4 == x^2 + -1
--    cocienteRuffini 3 ejPol4    == x^2 + 5*x + 14
-- ---------------------------------------------------------------------

-- 1ª definición
cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = creaPol (init (pRuffini r (coeficientes p)))

-- 2ª definición
cocienteRuffini2 :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini2 r = creaPol . pRuffini r . init . coeficientes

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    restoRuffini:: Int -> Polinomio Int -> Int
-- tal que (restoRuffini r p) es el resto de dividir el polinomio p por
-- el polinomio x-r. Por ejemplo,
--    restoRuffini 2 ejPol4    == 12
--    restoRuffini (-2) ejPol4 == 0
--    restoRuffini 3 ejPol4    == 40
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (pRuffini r (coeficientes p))

-- 2ª definición
restoRuffini2 :: Int -> Polinomio Int -> Int
restoRuffini2 r = last . pRuffini r . coeficientes

-- ---------------------------------------------------------------------
-- Ejercicio 10. Comprobar con QuickCheck que, dado un polinomio p y un
-- número entero r, las funciones anteriores verifican la propiedad de
-- la división euclídea.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_diviEuclidea :: Int -> Polinomio Int -> Bool
prop_diviEuclidea r p =
  p == sumaPol (multPol coci divi) rest
  where coci = cocienteRuffini r p
        divi = creaPol [1,-r]
        rest = creaTermino 0 (restoRuffini r p)

-- La comprobación es
--    ghci> quickCheck prop_diviEuclidea
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--     esRaizRuffini:: Int -> Polinomio Int -> Bool
-- tal que (esRaizRuffini r p) se verifica si r es una raiz de p, usando
-- para ello el regla de Ruffini. Por ejemplo,
--     esRaizRuffini 0 ejPol3 == True
--     esRaizRuffini 1 ejPol3 == False
-- ---------------------------------------------------------------------

esRaizRuffini :: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0

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

```haskell
-- Ejercicio 12. Definir la función
--     raicesRuffini :: Polinomio Int -> [Int]
-- tal que (raicesRuffini p) es la lista de las raices enteras de p,
-- calculadas usando el regla de Ruffini. Por ejemplo,
--     raicesRuffini ejPol1              == []
--     raicesRuffini ejPol2              == [0,-1]
--     raicesRuffini ejPol3              == [0]
--     raicesRuffini ejPol4              == [1,-1,-2]
--     raicesRuffini (creaPol [1,-2,1])  == [1,1]
-- ---------------------------------------------------------------------

raicesRuffini :: Polinomio Int -> [Int]
raicesRuffini p
  | esPolCero p = []
  | otherwise   = aux (0 : divisores (terminoIndep p))
  where aux [] = []
        aux (r:rs)
          | esRaizRuffini r p = r : raicesRuffini (cocienteRuffini r p)
          | otherwise         = aux rs

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    factorizacion :: Polinomio Int -> [Polinomio Int]
-- tal que (factorizacion p) es la lista de la descomposición del
-- polinomio p en factores obtenida mediante el regla de Ruffini. Por
-- ejemplo,
--  ejPol2                              ==  x^5 + 5*x^2 + 4*x
--  factorizacion ejPol2                == [1*x,1*x+1,x^3+-1*x^2+1*x+4]
--  ejPol4                              == x^3 + 2*x^2 + -1*x + -2
--  factorizacion ejPol4                == [1*x + -1,1*x + 1,1*x + 2,1]
--  factorizacion (creaPol [1,0,0,0,-1]) == [1*x + -1,1*x + 1,x^2 + 1]
-- ---------------------------------------------------------------------

factorizacion :: Polinomio Int -> [Polinomio Int]
factorizacion p
  | esPolCero p = [p]
  | otherwise   = aux (0 : divisores (terminoIndep p))
  where
    aux [] = [p]
    aux (r:rs)
```

```
        | esRaizRuffini r p =
            creaPol [1,-r] : factorizacion (cocienteRuffini r p)
        | otherwise = aux rs

-- ---------------------------------------------------------------------
-- Generador de polinomios                                             --
-- ---------------------------------------------------------------------

-- (genPol n) es un generador de polinomios. Por ejemplo,
--    ghci> sample (genPol 1)
--    7*x^9 + 9*x^8 + 10*x^7 + -14*x^5 + -15*x^2 + -10
--    -4*x^8 + 2*x
--    -8*x^9 + 4*x^8 + 2*x^6 + 4*x^5 + -6*x^4 + 5*x^2 + -8*x
--    -9*x^9 + x^5 + -7
--    8*x^10 + -9*x^7 + 7*x^6 + 9*x^5 + 10*x^3 + -1*x^2
--    7*x^10 + 5*x^9 + -5
--    -8*x^10 + -7
--    -5*x
--    5*x^10 + 4*x^4 + -3
--    3*x^3 + -4
--    10*x
genPol :: (Arbitrary a, Num a, Eq a) => Int -> Gen (Polinomio a)
genPol 0 = return polCero
genPol _ = do
  n <- choose (0,10)
  b <- arbitrary
  p <- genPol (div n 2)
  return (consPol n b p)

instance (Arbitrary a, Num a, Eq a) => Arbitrary (Polinomio a) where
  arbitrary = sized genPol
```

# Relación 33

# Programación dinámica: Caminos en una retícula

```
-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Data.List (genericLength)
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Se considera una retícula con sus posiciones numeradas,
-- desde el vértice superior izquierdo, hacia la derecha y hacia
-- abajo. Por ejemplo, la retícula de dimensión 3x4 se numera como sigue:
--    |-------+-------+-------+-------|
--    | (1,1) | (1,2) | (1,3) | (1,4) |
--    | (2,1) | (2,2) | (2,3) | (2,4) |
--    | (3,1) | (3,2) | (3,3) | (3,4) |
--    |-------+-------+-------+-------|
--
-- Definir, por recursión, la función
--    caminosR :: (Int,Int) -> [[(Int,Int)]]
-- tal que (caminosR (m,n)) es la lista de los caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
--    λ> caminosR (2,3)
--    [[(1,1),(1,2),(1,3),(2,3)],
--     [(1,1),(1,2),(2,2),(2,3)],
--     [(1,1),(2,1),(2,2),(2,3)]]
--    λ> mapM_ print (caminosR (3,4))
```

```
--    [(1,1),(1,2),(1,3),(1,4),(2,4),(3,4)]
--    [(1,1),(1,2),(1,3),(2,3),(2,4),(3,4)]
--    [(1,1),(1,2),(2,2),(2,3),(2,4),(3,4)]
--    [(1,1),(2,1),(2,2),(2,3),(2,4),(3,4)]
--    [(1,1),(1,2),(1,3),(2,3),(3,3),(3,4)]
--    [(1,1),(1,2),(2,2),(2,3),(3,3),(3,4)]
--    [(1,1),(2,1),(2,2),(2,3),(3,3),(3,4)]
--    [(1,1),(1,2),(2,2),(3,2),(3,3),(3,4)]
--    [(1,1),(2,1),(2,2),(3,2),(3,3),(3,4)]
--    [(1,1),(2,1),(3,1),(3,2),(3,3),(3,4)]
-- ---------------------------------------------------------------------

caminosR :: (Int,Int) -> [[(Int,Int)]]
caminosR p = map reverse (caminosRAux p)
  where
    caminosRAux (1,y) = [[(1,z) | z <- [y,y-1..1]]]
    caminosRAux (x,1) = [[(z,1) | z <- [x,x-1..1]]]
    caminosRAux (x,y) = [(x,y) : cs | cs <- caminosRAux (x-1,y) ++
                                            caminosRAux (x,y-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por programación dinámica, la función
--    caminosPD :: (Int,Int) -> [[(Int,Int)]]
-- tal que (caminosPD (m,n)) es la lista de los caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
--    λ> caminosPD (2,3)
--    [[(1,1),(1,2),(1,3),(2,3)],
--     [(1,1),(1,2),(2,2),(2,3)],
--     [(1,1),(2,1),(2,2),(2,3)]]
--    λ> mapM_ print (caminosPD (3,4))
--    [(1,1),(1,2),(1,3),(1,4),(2,4),(3,4)]
--    [(1,1),(1,2),(1,3),(2,3),(2,4),(3,4)]
--    [(1,1),(1,2),(2,2),(2,3),(2,4),(3,4)]
--    [(1,1),(2,1),(2,2),(2,3),(2,4),(3,4)]
--    [(1,1),(1,2),(1,3),(2,3),(3,3),(3,4)]
--    [(1,1),(1,2),(2,2),(2,3),(3,3),(3,4)]
--    [(1,1),(2,1),(2,2),(2,3),(3,3),(3,4)]
--    [(1,1),(1,2),(2,2),(3,2),(3,3),(3,4)]
--    [(1,1),(2,1),(2,2),(3,2),(3,3),(3,4)]
--    [(1,1),(2,1),(3,1),(3,2),(3,3),(3,4)]
```

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

caminosPD :: (Int,Int) -> [[(Int,Int)]]
caminosPD p = map reverse (matrizCaminos p ! p)

matrizCaminos :: (Int,Int) -> Array (Int,Int) [[(Int,Int)]]
matrizCaminos (m,n) = q
  where
    q = array ((1,1),(m,n)) [((i,j),f i j) | i <- [1..m], j <- [1..n]]
    f 1 y = [[(1,z) | z <- [y,y-1..1]]]
    f x 1 = [[(z,1) | z <- [x,x-1..1]]]
    f x y = [(x,y) : cs | cs <- q!(x-1,y) ++ q!(x,y-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comparar la eficiencia calculando el tiempo necesario
-- para evaluar las siguientes expresiones
--    length (head (caminosR (8,8)))
--    length (head (caminosR (8,8)))
--    maximum (head (caminosR (2000,2000)))
--    maximum (head (caminosPD (2000,2000)))
-- ---------------------------------------------------------------------

-- La comparación es
--    λ> length (head (caminosR (8,8)))
--    15
--    (0.02 secs, 504,056 bytes)
--    λ> length (head (caminosPD (8,8)))
--    15
--    (0.01 secs, 503,024 bytes)
--
--    λ> maximum (head (caminosR (2000,2000)))
--    (2000,2000)
--    (0.02 secs, 0 bytes)
--    λ> maximum (head (caminosPD (2000,2000)))
--    (2000,2000)
--    (1.30 secs, 199,077,664 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, usando caminosR, la función
--    nCaminosCR :: (Int,Int) -> Integer
```

```haskell
-- tal que (nCaminosCR (m,n)) es el número de caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
--      nCaminosR (2,3)                         ==  3
--      nCaminosR (3,4)                         ==  10
-- ---------------------------------------------------------------------

nCaminosCR :: (Int,Int) -> Integer
nCaminosCR = genericLength . caminosR

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, usando caminosPD, la función
--    nCaminosCPD :: (Int,Int) -> Integer
-- tal que (nCaminosCPD (m,n)) es el número de caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
-- ---------------------------------------------------------------------

nCaminosCPD :: (Int,Int) -> Integer
nCaminosCPD = genericLength . caminosPD

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir, por recursión, la función
--    nCaminosR :: (Int,Int) -> Integer
-- tal que (nCaminosR (m,n)) es el número de caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
-- ---------------------------------------------------------------------

nCaminosR :: (Int,Int) -> Integer
nCaminosR (1,_) = 1
nCaminosR (_,1) = 1
nCaminosR (x,y) = nCaminosR (x-1,y) + nCaminosR (x,y-1)

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Definir, por programación dinámica, la función
--    nCaminosPD :: (Int,Int) -> Integer
-- tal que (nCaminosPD (m,n)) es el número de caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
--    nCaminosPD (3,4)  ==  10
-- ---------------------------------------------------------------------

nCaminosPD :: (Int,Int) -> Integer
```

```haskell
nCaminosPD p = matrizNCaminos p ! p

matrizNCaminos :: (Int,Int) -> Array (Int,Int) Integer
matrizNCaminos (m,n) = q
  where
    q = array ((1,1),(m,n)) [((i,j),f i j) | i <- [1..m], j <- [1..n]]
    f 1 _ = 1
    f _ 1 = 1
    f x y = q!(x-1,y) + q!(x,y-1)

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Los caminos desde (1,1) a (m,n) son las permutaciones
-- con repetición de m-1 veces la A (abajo) y n-1 veces la D
-- (derecha). Por tanto, su  número es
--    ((m-1)+(n-1))! / (m-1)!*(n-1)!
--
-- Definir, con la fórmula anterior, la función
--    nCaminosF :: (Int,Int) -> Integer
-- tal que (nCaminosF (m,n)) es el número de caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
--    nCaminosF (8,8)  ==  3432
-- ---------------------------------------------------------------------

nCaminosF :: (Int,Int) -> Integer
nCaminosF (m,n) =
  fact ((m-1)+(n-1)) `div` (fact (m-1) * fact (n-1))

fact :: Int -> Integer
fact n = product [1..fromIntegral n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.6. La fórmula anterior para el cálculo del número de
-- caminos se puede simplificar.
--
-- Definir, con la fórmula simplificada, la función
--    nCaminosFS :: (Int,Int) -> Integer
-- tal que (nCaminosFS (m,n)) es el número de caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
-- ---------------------------------------------------------------------
```

```haskell
nCaminosFS :: (Int,Int) -> Integer
nCaminosFS (m,n) =
  product [a+1..a+b] `div` product [2..b]
  where m' = fromIntegral (m-1)
        n' = fromIntegral (n-1)
        a  = max m' n'
        b  = min m' n'

-- ---------------------------------------------------------------------
-- Ejercicio 2.7. Comparar la eficiencia calculando el tiempo necesario
-- para evaluar las siguientes expresiones
--    nCaminosCR  (8,8)
--    nCaminosCPD (8,8)
--    nCaminosCR  (12,12)
--    nCaminosCPD (12,12)
--    nCaminosR   (12,12)
--    nCaminosPD  (12,12)
--    length (show (nCaminosPD (1000,1000)))
--    length (show (nCaminosF  (1000,1000)))
--    length (show (nCaminosFS (1000,1000)))
--    length (show (nCaminosF  (2*10^4,2*10^4)))
--    length (show (nCaminosFS (2*10^4,2*10^4)))
-- ---------------------------------------------------------------------

-- La comparación es
--    λ> nCaminosCR (8,8)
--    3432
--    (2.11 secs, 2,132,573,904 bytes)
--    λ> nCaminosCPD (8,8)
--    3432
--    (0.02 secs, 0 bytes)
--
--    λ> nCaminosCR (12,12)
--    705432
--    (18.24 secs, 3,778,889,608 bytes)
--    λ> nCaminosCPD (12,12)
--    705432
--    (3.56 secs, 548,213,968 bytes)
--    λ> nCaminosR (12,12)
--    705432
```

```
--    (2.12 secs, 278,911,248 bytes)
--    λ> nCaminosPD (12,12)
--    705432
--    (0.01 secs, 0 bytes)
--
--    λ> length (show (nCaminosPD (1000,1000)))
--    600
--    (4.88 secs, 693,774,912 bytes)
--    λ> length (show (nCaminosF (1000,1000)))
--    600
--    (0.01 secs, 0 bytes)
--    λ> length (show (nCaminosFS (1000,1000)))
--    600
--    (0.01 secs, 0 bytes)
--
--    λ> length (show (nCaminosF (2*10^4,2*10^4)))
--    12039
--    (8.01 secs, 2,376,767,288 bytes)
--    λ> length (show (nCaminosFS (2*10^4,2*10^4)))
--    12039
--    (2.84 secs, 836,245,992 bytes)
```

# Relación 34

# Programación dinámica: Turista en Manhattan

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

-- En el siguiente gráfico se representa en una cuadrícula el plano de
-- Manhattan. Cada línea es una opción a seguir; el número representa
-- las atracciones que se pueden visitar si se elige esa opción.
--
--          3         2         4         0
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |1        |0        |2        |4        |3
--     |    3    |    2    |    4    |    2    |
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |4        |6        |5        |2        |1
--     |    0    |    7    |    3    |    4    |
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |4        |4        |5        |2        |1
--     |    3    |    3    |    0    |    2    |
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |5        |6        |8        |5        |3
--     |    1    |    3    |    2    |    2    |
--     * ------- * ------- * ------- * ------- *
--
-- El turista entra por el extremo superior izquierda y sale por el
```

```
-- extremo inferior derecha. Sólo puede moverse en las direcciones Sur y
-- Este (es decir, hacia abajo o hacia la derecha).
--
-- Representamos el mapa mediante una matriz p tal que p(i,j) = (a,b),
-- donde a = nº de atracciones si se va hacia el sur y b = nº de
-- atracciones si se va al este. Además, ponemos un 0 en el valor del
-- número de atracciones por un camino que no se puede elegir. De esta
-- forma, el mapa anterior se representa por la matriz siguiente:
--
--    ( (1,3)   (0,2)   (2,4)   (4,0)  (3,0) )
--    ( (4,3)   (6,2)   (5,4)   (2,2)  (1,0) )
--    ( (4,0)   (4,7)   (5,3)   (2,4)  (1,0) )
--    ( (5,3)   (6,3)   (8,0)   (5,2)  (3,0) )
--    ( (0,1)   (0,3)   (0,2)   (0,2)  (0,0) )
--
-- En este caso, si se hace el recorrido
--    [S, E, S, E, S, S, E, E],
-- el número de atracciones es
--     1  3  6  7  5  8  2  2
-- cuya suma es 34.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Data.Matrix

-- ---------------------------------------------------------------------
-- § Ejercicios                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por recursión, la función
--    mayorNumeroVR :: Matrix (Int,Int) -> Int
-- tal que (mayorNumeroVR p) es el máximo número de atracciones que se
-- pueden visitar en el plano representado por la matriz p. Por ejemplo,
-- si se define la matriz anterior por
--    ej1, ej2, ej3 :: Matrix (Int,Int)
--    ej1 = fromLists [[(1,3),(0,2),(2,4),(4,0),(3,0)],
--                     [(4,3),(6,2),(5,4),(2,2),(1,0)],
```

```
--                      [(4,0),(4,7),(5,3),(2,4),(1,0)],
--                      [(5,3),(6,3),(8,0),(5,2),(3,0)],
--                      [(0,1),(0,3),(0,2),(0,2),(0,0)]]
--    ej2 = fromLists [[(1,3),(0,0)],
--                      [(0,3),(0,0)]]
--    ej3 = fromLists [[(1,3),(0,2),(2,0)],
--                      [(4,3),(6,2),(5,0)],
--                      [(0,0),(0,7),(0,0)]]
-- entonces
--    mayorNumeroVR ej1 == 34
--    mayorNumeroVR ej2 == 4
--    mayorNumeroVR ej3 == 17
-- ---------------------------------------------------------------------

ej1, ej2, ej3 :: Matrix (Int,Int)
ej1 = fromLists [[(1,3),(0,2),(2,4),(4,0),(3,0)],
                 [(4,3),(6,2),(5,4),(2,2),(1,0)],
                 [(4,0),(4,7),(5,3),(2,4),(1,0)],
                 [(5,3),(6,3),(8,0),(5,2),(3,0)],
                 [(0,1),(0,3),(0,2),(0,2),(0,0)]]
ej2 = fromLists [[(1,3),(0,0)],
                 [(0,3),(0,0)]]
ej3 = fromLists [[(1,3),(0,2),(2,0)],
                 [(4,3),(6,2),(5,0)],
                 [(0,0),(0,7),(0,0)]]

mayorNumeroVR :: Matrix (Int,Int) -> Int
mayorNumeroVR p = aux m n
  where m = nrows p
        n = ncols p
        aux 1 1 = 0
        aux 1 j = sum [snd (p !(1,k)) | k <-[1..j-1]]
        aux i 1 = sum [fst (p !(k,1)) | k <-[1..i-1]]
        aux i j = max (aux (i-1) j + fst (p !(i-1,j)))
                      (aux i (j-1) + snd (p !(i,j-1)))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por programación dinámica, la función
--    mayorNumeroVPD :: Matrix (Int,Int) -> Int
-- tal que (mayorNumeroVPD p) es el máximo número de atracciones que se
```

```haskell
-- pueden visitar en el plano representado por la matriz p. Por ejemplo,
--    mayorNumeroVPD ej1 == 34
--    mayorNumeroVPD ej2 == 4
--    mayorNumeroVPD ej3 == 17
-- ---------------------------------------------------------------------

mayorNumeroVPD :: Matrix (Int,Int) -> Int
mayorNumeroVPD p = matrizNumeroV p ! (m,n)
  where m = nrows p
        n = ncols p

matrizNumeroV :: Matrix (Int,Int) -> Matrix Int
matrizNumeroV p = q
  where m = nrows p
        n = ncols p
        q = matrix m n f
          where f (1,1) = 0
                f (1,j) = sum [snd (p !(1,k)) | k <-[1..j-1]]
                f (i,1) = sum [fst (p !(k,1)) | k <-[1..i-1]]
                f (i,j) = max (q !(i-1,j) + fst (p !(i-1,j)))
                              (q !(i,j-1) + snd (p !(i,j-1)))

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comparar la eficiencia observando las estadísticas
-- correspondientes a los siguientes cálculos
--    mayorNumeroVR  (fromList 13 13 [(n,n+1) | n <- [1..]])
--    mayorNumeroVPD (fromList 13 13 [(n,n+1) | n <- [1..]])
-- ---------------------------------------------------------------------

-- La comparación es
--    λ> mayorNumeroVR (fromList 13 13 [(n,n+1) | n <- [1..]])
--    2832
--    (6.54 secs, 5,179,120,504 bytes)
--    λ> mayorNumeroVPD (fromList 13 13 [(n,n+1) | n <- [1..]])
--    2832
--    (0.01 secs, 670,128 bytes)
```

# Relación 35

# Programación dinámica: Apilamiento de barriles

```
-- ---------------------------------------------------------------------
-- § Introducción                                                      --
-- ---------------------------------------------------------------------

-- Un montón de barriles se construye apilando unos encima de otros por
-- capas, de forma que en cada capa todos los barriles están apoyados
-- sobre dos de la capa inferior y todos los barriles de una misma capa
-- están pegados unos a otros. Por ejemplo, los siguientes montones son
-- válidos:
--
--       _          _   _                  _
--      / \        / \ / \                / \
--     _\_/_      _\_/_\_/_   _       _   _\_/_   _
--    / \ / \    / \ / \ / \ / \     / \ / \ / \ / \
--    \_/ \_/    \_/ \_/ \_/ \_/     \_/ \_/ \_/ \_/
--
-- y los siguientes no son válidos:
--
--     _   _          _       _              _   _
--    / \ / \        / \     / \            / \ / \
--    \_/_\_/_      _\_/_   _\_/_      _   _\_/_\_/
--      / \ / \    / \ / \ / \ / \     / \ / \ / \
--      \_/ \_/    \_/ \_/ \_/ \_/     \_/ \_/ \_/
--
-- Se puede comprobar que el número M(n) de formas distintas de
-- construir montones con n barriles en la base viene dado por la
-- siguiente fórmula:
--               n-1
```

```
--               -------
--                \
--                 \
--    M(n) = 1 +    )    (n-j) * M(j)
--                 /
--                /
--               -------
--                j = 1
--
-- El objetivo de esta relación es estudiar la transformación de
-- definiciones recursivas en otras con programación dinámica y comparar
-- su eficiencia.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import Data.Array

-- ---------------------------------------------------------------------
-- § Ejercicios                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por recursión, la función
--    montonesR :: Integer -> Integer
-- tal que (montonesR n) es el número de formas distintas de construir
-- montones con n barriles en la base. Por ejemplo,
--    montonesR 1   ==  1
--    montonesR 5   ==  34
--    montonesR 10  ==  4181
--    montonesR 15  ==  514229
--    montonesR 20  ==  63245986
-- ---------------------------------------------------------------------

montonesR :: Integer -> Integer
montonesR 1 = 1
montonesR n = 1 + sum [(n-j) * montonesR j | j <- [1..n-1]]

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

```haskell
-- Ejercicio 2. Definir, por programación dinámica, la función
--    montonesPD :: Integer -> Integer
-- tal que (montonesPD n) es el número de formas distintas de construir
-- montones con n barriles en la base. Por ejemplo,
--    montonesR 1   ==  1
--    montonesR 5   ==  34
--    montonesR 10  ==  4181
--    montonesR 15  ==  514229
--    montonesR 20  ==  63245986
--    length (show (montonesPD 1000))  ==  418
-- ---------------------------------------------------------------------

montonesPD :: Integer -> Integer
montonesPD n = (vectorMontones n) ! n

vectorMontones :: Integer -> Array Integer Integer
vectorMontones n = v where
  v = array (1,n) [(i,f i) | i <- [1..n]]
  f 1 = 1
  f k = 1 + sum [(k-j)*v!j | j <- [1..k-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comparar la eficiencia calculando el tiempo necesario
-- para evaluar las siguientes expresiones
--    montonesR  23
--    montonesPD 23
-- ---------------------------------------------------------------------

-- La comparación es
--    λ> montonesR 23
--    1134903170
--    (16.76 secs, 2,617,836,192 bytes)
--    λ> montonesPD 23
--    1134903170
--    (0.01 secs, 724,248 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Operando con las ecuaciones de M(n) se observa que
--    M(1) = 1                          = 1
--    M(2) = 1 + M(1)                   = M(1) + M(1)
```

```
--    M(3) = 1 + 2*M(1) + M(2)          = M(2) + (M(1) + M(2))
--    M(4) = 1 + 3*M(1) + 2*M(2) + M(3) = M(3) + (M(1) + M(2) + M(3))
-- En general,
--    M(n) = M(n-1) + (M(1) + ... + M(n-1))
--
-- Unsando la ecuación anterior, definir por recursión la función
--    montonesR2 :: Integer -> Integer
-- tal que (montonesR2 n) es el número de formas distintas de construir
-- montones con n barriles en la base. Por ejemplo,
--    montonesR2 1   ==  1
--    montonesR2 5   ==  34
--    montonesR2 10  ==  4181
--    montonesR2 15  ==  514229
--    montonesR2 20  ==  63245986
-- ---------------------------------------------------------------------

montonesR2 :: Integer -> Integer
montonesR2 = fst . montonesR2Aux

-- (montonesR2Aux n) es el par formado por M(n) y la suma
-- M(1)+...+M(n). Por ejemplo,
--    montonesR2Aux 10                  ==  (4181,6765)
--    montonesR 10                      ==  4181
--    sum [montonesR k | k <- [1..10]]  ==  6765
montonesR2Aux :: Integer -> (Integer,Integer)
montonesR2Aux 1 = (1,1)
montonesR2Aux n = (x+y,y+x+y)
  where (x,y) = montonesR2Aux (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Comparar la eficiencia calculando el tiempo necesario
-- para evaluar las siguientes expresiones
--    montonesR  23
--    montonesR2 23
--    length (show (montonesPD 1000))
--    length (show (montonesR2 1000))
-- ---------------------------------------------------------------------

-- La comparación es
--    λ> montonesR 23
```

```
--    1134903170
--    (16.76 secs, 2,617,836,192 bytes)
--    λ> montonesR2 23
--    1134903170
--    (0.01 secs, 602,104 bytes)
--    λ> length (show (montonesPD 1000))
--    418
--    (2.29 secs, 349,208,304 bytes)
--    λ> length (show (montonesR2 1000))
--    418
--    (0.01 secs, 1,600,192 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Usando la ecuación anterior y programación dinámica,
-- definir la función
--    montonesPD2 :: Integer -> Integer
-- tal que (montonesPD2 n) es el número de formas distintas de construir
-- montones con n barriles en la base. Por ejemplo,
--    montonesPD2 1   ==  1
--    montonesPD2 5   ==  34
--    montonesPD2 10  ==  4181
--    montonesPD2 15  ==  514229
--    montonesPD2 20  ==  63245986
-- ---------------------------------------------------------------------

montonesPD2 :: Integer -> Integer
montonesPD2 n = fst ((vectorMontones2 n) ! n)

vectorMontones2 :: Integer -> Array Integer (Integer,Integer)
vectorMontones2 n = v where
  v = array (1,n) [(i,f i) | i <- [1..n]]
  f 1 = (1,1)
  f k = (x+y,y+x+y)
    where (x,y) = v!(k-1)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Comparar la eficiencia calculando el tiempo necesario
-- para evaluar las siguientes expresiones
--    length (show (montonesR2  40000))
--    length (show (montonesPD2 40000))
```

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

-- La comparación es
--    λ> length (show (montonesR2 40000))
--    16719
--    (2.04 secs, 452,447,664 bytes)
--    λ> length (show (montonesPD2 40000))
--    16719
--    (2.12 secs, 466,528,472 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir, usando scanl1, la lista
--    sucMontones :: [Integer]
-- cuyos elementos son los números de formas distintas de construir
-- montones con n barriles en la base, para n = 1, 2, .... Por ejemplo,
--    take 10 sucMontones  ==  [1,2,5,13,34,89,233,610,1597,4181]
-- ---------------------------------------------------------------------

sucMontones :: [Integer]
sucMontones = 1 : zipWith (+) sucMontones (scanl1 (+) sucMontones)

-- El cálculo es
--    | sucMontones        | scanl1 (+) sucMontones |
--    | 1:...              | 1:...                  |
--    | 1:2:...            | 1:3:...                |
--    | 1:2:5:...          | 1:3:8:...              |
--    | 1:2:5:13:...       | 1:3:8:21:...           |
--    | 1:2:5:13:34:...    | 1:3:8:21:55:...        |
--    | 1:2:5:13:34:89:... | 1:3:8:21:55:144:...    |

-- ---------------------------------------------------------------------
-- Ejercicio 8. Usando la sucesión anterior, definir la función
--    montonesS :: Integer -> Integer
-- tal que (montonesS n) es el número de formas distintas de construir
-- montones con n barriles en la base. Por ejemplo,
--    montonesS 1   ==  1
--    montonesS 5   ==  34
--    montonesS 10  ==  4181
--    montonesS 15  ==  514229
--    montonesS 20  ==  63245986
```

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

montonesS :: Int -> Integer
montonesS n = sucMontones !! (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 9. Comparar la eficiencia calculando el tiempo necesario
-- para evaluar las siguientes expresiones
--    length (show (montonesR2  40000))
--    length (show (montonesPD2 40000))
--    length (show (montonesS   40000))
-- ---------------------------------------------------------------------

-- La comparación es
--    λ> length (show (montonesR2 40000))
--    16719
--    (2.04 secs, 452,447,664 bytes)
--    λ> length (show (montonesPD2 40000))
--    16719
--    (2.12 secs, 466,528,472 bytes)
--    λ> length (show (montonesS 40000))
--    16719
--    (0.72 secs, 298,062,216 bytes)
```

# Relación 36

# El problema del granjero mediante búsqueda en espacio de estado

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

-- Un granjero está parado en un lado del río y con él tiene un lobo,
-- una cabra y una repollo. En el río hay un barco pequeño. El granjero
-- desea cruzar el río con sus tres posesiones. No hay puentes y en el
-- barco hay solamente sitio para el granjero y un artículo. Si deja
-- la cabra con la repollo sola en un lado del río la cabra comerá la
-- repollo. Si deja el lobo y la cabra en un lado, el lobo se comerá a
-- la cabra. ¿Cómo puede cruzar el granjero el río con los tres
-- artículos, sin que ninguno se coma al otro?
--
-- El objetivo de esta relación de ejercicios es resolver el problema
-- del granjero mediante búsqueda en espacio de estados, utilizando las
-- implementaciones estudiadas en el tema 23
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-23.html
--
-- Para realizar los ejercicios hay que tener instalada la librería I1M
-- que se encuentra en
-- http://hackage.haskell.org/package/I1M
--
-- Para instalar la librería de I1M, basta ejecutar en una consola
--    cabal update
```

```
--    cabal install I1M
--
-- Otra forma es descargar, en el directorio de ejercicios, la
-- implementación de la búsqueda en espacio de estado
-- + BusquedaEnEspaciosDeEstados.hs que está en https://bit.ly/3dE8pDp

-- ---------------------------------------------------------------------
-- Importaciones                                                       --
-- ---------------------------------------------------------------------

-- Hay que elegir una librería
import I1M.BusquedaEnEspaciosDeEstados
-- import BusquedaEnEspaciosDeEstados

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir el tipo Orilla con dos constructores I y D que
-- representan las orillas izquierda y derecha, respectivamente.
-- ---------------------------------------------------------------------

data Orilla = I | D
  deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir el tipo Estado como abreviatura de una tupla que
-- representan en qué orilla se encuentra cada uno de los elementos
-- (granjero, lobo, cabra, repollo). Por ejemplo, (I,D,D,I) representa
-- que el granjero está en la izquierda, que el lobo está en la derecha,
-- que la cabra está en la derecha y el repollo está en la izquierda.
-- ---------------------------------------------------------------------

type Estado = (Orilla,Orilla,Orilla,Orilla)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir
--    inicial :: Estado
-- tal que inicial representa el estado en el que todos están en la
-- orilla izquierda.
-- ---------------------------------------------------------------------

inicial :: Estado
```

```haskell
inicial = (I,I,I,I)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir
--    final:: Estado
-- tal que final representa el estado en el que todos están en la
-- orilla derecha.
-- ---------------------------------------------------------------------

final :: Estado
final = (D,D,D,D)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--   seguro :: Estado -> Bool
-- tal que (seguro e) se verifica si el estado e es seguro; es decir,
-- que no puede estar en una orilla el lobo con la cabra sin el granjero
-- ni la cabra con el repollo sin el granjero. Por ejemplo,
--    seguro (I,D,D,I)  ==  False
--    seguro (D,D,D,I)  ==  True
--    seguro (D,D,I,I)  ==  False
--    seguro (I,D,I,I)  ==  True
-- ---------------------------------------------------------------------

-- 1ª definición
seguro :: Estado -> Bool
seguro (g,l,c,r)
  | l == c    = g == l
  | c == r    = g == c
  | otherwise = True

-- 2ª definición
seguro2 :: Estado -> Bool
seguro2 (g,l,c,r) = not (g /= c && (c == l || c == r))

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    opuesta :: Orilla -> Orilla
-- tal que (opuesta x) es la opuesta de la orilla x. Por ejemplo
--    opuesta I = D
```

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

opuesta :: Orilla -> Orilla
opuesta I = D
opuesta D = I

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    sucesoresE :: Estado -> [Estado]
-- tal que (sucesoresE e) es la lista de los sucesores seguros del
-- estado e. Por ejemplo,
--    sucesoresE (I,I,I,I)  ==  [(D,I,D,I)]
--    sucesoresE (D,I,D,I)  ==  [(I,I,D,I),(I,I,I,I)]
-- ---------------------------------------------------------------------

sucesoresE :: Estado -> [Estado]
sucesoresE e = [mov e | mov <- [m1,m2,m3,m4], seguro (mov e)]
  where m1 (g,l,c,r) = (opuesta g, l, c, r)
        m2 (g,l,c,r) = (opuesta g, opuesta l, c, r)
        m3 (g,l,c,r) = (opuesta g, l, opuesta c, r)
        m4 (g,l,c,r) = (opuesta g, l, c, opuesta r)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Los nodos del espacio de búsqueda son lista de estados
--    [e_n, ..., e_2, e_1]
-- donde e_1 es el estado inicial y para cada i (2 <= i <= n), e_i es un
-- sucesor de e_(i-1).
--
-- Definir el tipo de datos NodoRio para representar los nodos del
-- espacio de búsqueda. Por ejemplo,
--    ghci> :type (Nodo [(I,I,D,I),(I,I,I,I)])
--    (Nodo [(I,I,D,I),(I,I,I,I)]) :: NodoRio
-- ---------------------------------------------------------------------

data NodoRio = Nodo [Estado]
  deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    sucesoresN :: NodoRio -> [NodoRio]
```

```haskell
-- tal que (sucesoresN n) es la lista de los sucesores del nodo n. Por
-- ejemplo,
--    ghci> sucesoresN (Nodo [(I,I,D,I),(D,I,D,I),(I,I,I,I)])
--    [Nodo [(D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)],
--     Nodo [(D,I,D,D),(I,I,D,I),(D,I,D,I),(I,I,I,I)]]
-- ---------------------------------------------------------------------

sucesoresN :: NodoRio -> [NodoRio]
sucesoresN (Nodo (n@(e:es))) =
  [Nodo (e':n) | e' <- sucesoresE e, notElem e' es]
sucesoresN _ =
  error "Imposible"

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    esFinal:: NodoRio -> Bool
-- tal que (esFinal n) se verifica si n es un nodo final; es decir, su
-- primer elemento es el estado final. Por ejemplo,
--    esFinal (Nodo [(D,D,D,D),(I,I,I,I)])  ==  True
--    esFinal (Nodo [(I,I,D,I),(I,I,I,I)])  ==  False
-- ---------------------------------------------------------------------

esFinal :: NodoRio -> Bool
esFinal (Nodo (n:_)) = n == final
esFinal _            = error "Imposible"

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    granjeroEE :: [NodoRio]
-- tal que granjeroEE son las soluciones del problema del granjero
-- mediante el patrón de búsqueda en espacio de estados. Por ejemplo,
--    ghci> head granjeroEE
--    Nodo [(D,D,D,D),(I,D,I,D),(D,D,I,D),(I,D,I,I),
--          (D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)]
--    ghci> length granjeroEE
--    2
-- ---------------------------------------------------------------------

granjeroEE :: [NodoRio]
granjeroEE = buscaEE sucesoresN
```

```
esFinal
(Nodo [inicial])
```

# Relación 37

# Resolución de problemas mediante búsqueda en espacios de estados

```
#-}

module Rel_37_sol where

-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es resolver problemas
-- mediante búsqueda en espacio de estados, utilizando las
-- implementaciones estudiadas en el tema 23 que se pueden descargar
-- desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/codigosDeTemas.php
--
-- Las transparencias del tema 23 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-23.pdf

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
-- ---------------------------------------------------------------------

import I1M.BusquedaEnEspaciosDeEstados
import Data.List
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Las fichas del dominó se pueden representar por pares de
-- números enteros. El problema del dominó consiste en colocar todas las
-- fichas de una lista dada de forma que el segundo número de cada ficha
-- coincida con el primero de la siguiente.
--
-- Definir, mediante búsqueda en espacio de estados, la función
--    domino :: [(Int,Int)] -> [[(Int,Int)]]
-- tal que (domino fs) es la lista de las soluciones del problema del
-- dominó correspondiente a las fichas fs. Por ejemplo,
--    ghci> domino [(1,2),(2,3),(1,4)]
--    [[(4,1),(1,2),(2,3)],[(3,2),(2,1),(1,4)]]
--    ghci> domino [(1,2),(1,1),(1,4)]
--    [[(4,1),(1,1),(1,2)],[(2,1),(1,1),(1,4)]]
--    ghci> domino [(1,2),(3,4),(2,3)]
--    [[(1,2),(2,3),(3,4)],[(4,3),(3,2),(2,1)]]
--    ghci> domino [(1,2),(2,3),(5,4)]
--    []
-- ---------------------------------------------------------------------

-- Las fichas son pares de números enteros.
type Ficha  = (Int,Int)

-- Un problema está definido por la lista de fichas que hay que colocar
type Problema = [Ficha]

-- Los estados son los pares formados por la listas sin colocar y las
-- colocadas.
type Estado = ([Ficha],[Ficha])

-- (inicial p) es el estado inicial del problema p.
inicial :: Problema -> Estado
inicial p = (p,[])

-- (es final e) se verifica si e es un estado final.
esFinal :: Estado -> Bool
esFinal (fs,_) = null fs

sucesores :: Estado -> [Estado]
sucesores (fs,[]) =
```

```
  [(delete (a,b) fs, [(a,b)]) | (a,b) <- fs, a /= b] ++
  [(delete (a,b) fs, [(b,a)]) | (a,b) <- fs]
sucesores (fs,n@((x,y):qs)) =
  [(delete (u,v) fs,(u,v):n) | (u,v) <- fs, u /= v, v == x] ++
  [(delete (u,v) fs,(v,u):n) | (u,v) <- fs, u /= v, u == x] ++
  [(delete (u,v) fs,(u,v):n) | (u,v) <- fs, u == v, u == x]

soluciones :: Problema -> [Estado]
soluciones ps = buscaEE sucesores
                        esFinal
                        (inicial ps)

domino :: Problema -> [[Ficha]]
domino ps = map snd (soluciones ps)

-- ---------------------------------------------------------------------
-- Ejercicio 2. El problema de suma cero consiste en, dado el conjunto
-- de números enteros, encontrar sus subconjuntos no vacío cuyos
-- elementos sumen cero.
--
-- Definir, mediante búsqueda en espacio de estados, la función
--    suma0 :: [Int] -> [[Int]]
-- tal que (suma0 ns) es la lista de las soluciones del problema de suma
-- cero para ns. Por ejemplo,
--    ghci> suma0 [-7,-3,-2,5,8]
--    [[-3,-2,5]]
--    ghci> suma0 [-7,-3,-2,5,8,-1]
--    [[-7,-3,-2,-1,5,8],[-7,-1,8],[-3,-2,5]]
--    ghci> suma0 [-7,-3,1,5,8]
--    []
-- ---------------------------------------------------------------------


-- Los estados son ternas formadas por los números seleccionados, su
-- suma y los restantes números.
type EstadoSuma0 = ([Int], Int, [Int])

inicialSuma0 :: [Int] -> EstadoSuma0
inicialSuma0 ns = ([],0,ns)
```

```haskell
esFinalSuma0 :: EstadoSuma0 -> Bool
esFinalSuma0 (xs,s,_) = not (null xs) && s == 0

sucesoresSuma0 :: EstadoSuma0 -> [EstadoSuma0]
sucesoresSuma0 (xs,s,ns) = [(n:xs, n+s, delete n ns) | n <- ns]

solucionesSuma0 :: [Int] -> [EstadoSuma0]
solucionesSuma0 ns = buscaEE sucesoresSuma0
                             esFinalSuma0
                             (inicialSuma0 ns)

suma0 :: [Int] -> [[Int]]
suma0 ns = nub [sort xs | (xs,_,_) <- solucionesSuma0 ns]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se tienen dos jarras, una de 4 litros de capacidad y
-- otra de 3. Ninguna de ellas tiene marcas de medición. Se tiene una
-- bomba que permite llenar las jarras de agua. El problema de las
-- jarras consiste en determinar cómo se puede lograr tener exactamente
-- 2 litros de agua en la jarra de 4 litros de capacidad.
--
-- Definir, mediante búsqueda en espacio de estados, la función
--    jarras :: [[(Int,Int)]]
-- tal que su valor es la lista de las soluciones del problema de las
-- jarras, Por ejemplo,
--    ghci> jarras !! 4
--    [(0,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)]
-- La interpretación de la solución es:
--    (0,0) se inicia con las dos jarras vacías,
--    (4,0) se llena la jarra de 4 con el grifo,
--    (1,3) se llena la de 3 con la de 4,
--    (1,0) se vacía la de 3,
--    (0,1) se pasa el contenido de la primera a la segunda,
--    (4,1) se llena la primera con el grifo,
--    (2,3) se llena la segunda con la primera.
--
-- Nota. No importa el orden en el que se generan las soluciones.
-- ---------------------------------------------------------------------

-- Un estado es una lista de dos números. El primero es el contenido de
```

```haskell
-- la jarra de 4 litros y el segundo el de la de 3 litros.
type EstadoJarras = (Int,Int)

inicialJarras :: EstadoJarras
inicialJarras = (0,0)

esFinalJarras :: EstadoJarras -> Bool
esFinalJarras (x,_) = x == 2

sucesoresEjarras :: EstadoJarras -> [EstadoJarras]
sucesoresEjarras (x,y) =
  [(4,y) | x < 4] ++
  [(x,3) | y < 3] ++
  [(0,y) | x > 0] ++
  [(x,0) | y > 0] ++
  [(4,y-(4-x)) | x < 4, y > 0, x + y > 4] ++
  [(x-(3-y),3) | x > 0, y < 3, x + y > 3] ++
  [(x+y,0) | y > 0, x + y <= 4] ++
  [(0,x+y) | x > 0, x + y <= 3]

-- Los nodos son las soluciones parciales
type NodoJarras = [EstadoJarras]

inicialNjarras :: NodoJarras
inicialNjarras = [inicialJarras]

esFinalNjarras :: NodoJarras -> Bool
esFinalNjarras (e:_) = esFinalJarras e

sucesoresNjarras :: NodoJarras -> [NodoJarras]
sucesoresNjarras n@(e:es) =
  [e':n | e' <- sucesoresEjarras e,
          e' `notElem` n]

solucionesJarras :: [NodoJarras]
solucionesJarras = buscaEE sucesoresNjarras
                           esFinalNjarras
                           inicialNjarras

jarras :: [[(Int,Int)]]
```

```
jarras = map reverse solucionesJarras
```

# Relación 38

# Implementación del TAD de los grafos mediante listas

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

-- El objetivo de esta relación es implementar el TAD de los grafos
-- mediante listas, de manera análoga a las implementaciones estudiadas
-- en el tema 22 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-22.html
-- y usando la mismas signatura.

-- ---------------------------------------------------------------------
-- Signatura                                                           --
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module Rel_38_sol
    (Orientacion (..),
     Grafo,
     creaGrafo,  -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                 --                    Grafo v p
     dirigido,   -- (Ix v,Num p) => (Grafo v p) -> Bool
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristas,    -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where
```

```haskell
-- ---------------------------------------------------------------------
-- Librerías auxiliares                                                --
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

-- ---------------------------------------------------------------------
-- Representación de los grafos mediante listas                        --
-- ---------------------------------------------------------------------

-- Orientacion es D (dirigida) ó ND (no dirigida).
data Orientacion = D | ND
                 deriving (Eq, Show)

-- (Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p.
data Grafo v p = G Orientacion ([v],[((v,v),p)])
               deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicios                                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
-- tal que (creaGrafo o cs as) es un grafo (dirigido o no, según el
-- valor de o), con el par de cotas cs y listas de aristas as (cada
-- arista es un trío formado por los dos vértices y su peso). Por
-- ejemplo,
--    ghci> creaGrafo ND (1,3) [(1,2,12),(1,3,34)]
--    G ND ([1,2,3],[((1,2),12),((1,3),34),((2,1),12),((3,1),34)])
--    ghci> creaGrafo D (1,3) [(1,2,12),(1,3,34)]
--    G D ([1,2,3],[((1,2),12),((1,3),34)])
--    ghci> creaGrafo D (1,4) [(1,2,12),(1,3,34)]
--    G D ([1,2,3,4],[((1,2),12),((1,3),34)])
-- ---------------------------------------------------------------------

creaGrafo :: (Ix v, Num p) =>
```

```
             Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs as =
    G o (range cs, [((x1,x2),w) | (x1,x2,w) <- as] ++
                    if o == D then []
                    else [((x2,x1),w) | (x1,x2,w) <- as, x1 /= x2])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, con creaGrafo, la constante
--    ejGrafoND :: Grafo Int Int
-- para representar el siguiente grafo no dirigido
--              12
--        1 -------- 2
--        | \78     /|
--        |  \   32/ |
--        |   \   /  |
--      34|     5    |55
--        |   /   \  |
--        |  /44   \ |
--        | /     93\|
--        3 -------- 4
--              61
--    ghci> ejGrafoND
--    G ND ([1,2,3,4,5],
--          [((1,2),12),((1,3),34),((1,5),78),((2,4),55),((2,5),32),
--           ((3,4),61),((3,5),44),((4,5),93),((2,1),12),((3,1),34),
--           ((5,1),78),((4,2),55),((5,2),32),((4,3),61),((5,3),44),
--           ((5,4),93)])
-- ---------------------------------------------------------------------

ejGrafoND :: Grafo Int Int
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                (2,4,55),(2,5,32),
                                (3,4,61),(3,5,44),
                                (4,5,93)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, con creaGrafo, la constante
--    ejGrafoD :: Grafo Int Int
-- para representar el grafo anterior donde se considera que las aristas
-- son los pares (x,y) con x < y. Por ejemplo,
```

```
--    ghci> ejGrafoD
--    G D ([1,2,3,4,5],
--         [((1,2),12),((1,3),34),((1,5),78),((2,4),55),((2,5),32),
--          ((3,4),61),((3,5),44),((4,5),93)])
-- ---------------------------------------------------------------------

ejGrafoD :: Grafo Int Int
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                              (2,4,55),(2,5,32),
                              (3,4,61),(3,5,44),
                              (4,5,93)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
-- tal que (dirigido g) se verifica si g es dirigido. Por ejemplo,
--    dirigido ejGrafoD   ==  True
--    dirigido ejGrafoND  ==  False
-- ---------------------------------------------------------------------

dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
-- tal que (nodos g) es la lista de todos los nodos del grafo g. Por
-- ejemplo,
--    nodos ejGrafoND  ==  [1,2,3,4,5]
--    nodos ejGrafoD   ==  [1,2,3,4,5]
-- ---------------------------------------------------------------------

nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos (G _ (ns,_)) = ns

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
-- tal que (adyacentes g v) es la lista de los vértices adyacentes al
-- nodo v en el grafo g. Por ejemplo,
```

```
--    adyacentes ejGrafoND 4  ==  [5,2,3]
--    adyacentes ejGrafoD  4  ==  [5]
-- ---------------------------------------------------------------------

adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adyacentes (G _ (_,e)) v = nub [u | ((w,u),_) <- e, w == v]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
-- (aristaEn g a) se verifica si a es una arista del grafo g. Por
-- ejemplo,
--    aristaEn ejGrafoND (5,1)  ==  True
--    aristaEn ejGrafoND (4,1)  ==  False
--    aristaEn ejGrafoD  (5,1)  ==  False
--    aristaEn ejGrafoD  (1,5)  ==  True
-- ---------------------------------------------------------------------

aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
-- tal que (peso v1 v2 g) es el peso de la arista que une los vértices
-- v1 y v2 en el grafo g. Por ejemplo,
--    peso 1 5 ejGrafoND  ==  78
--    peso 1 5 ejGrafoD   ==  78
-- ---------------------------------------------------------------------

peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
peso x y (G _ (_,gs)) = head [c | ((x',y'),c) <- gs, x==x', y==y']

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
-- (aristasD g) es la lista de las aristas del grafo g. Por ejemplo,
--    ghci> aristas ejGrafoD
--    [(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
--     (3,5,44),(4,5,93)]
```

```
--    ghci> aristas ejGrafoND
--    [(1,2,12),(1,3,34),(1,5,78),(2,1,12),(2,4,55),(2,5,32),
--     (3,1,34),(3,4,61),(3,5,44),(4,2,55),(4,3,61),(4,5,93),
--     (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
-- ------------------------------------------------------------------

aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
aristas (G _ (_,g)) = [(v1,v2,p) | ((v1,v2),p) <- g]
```

# Relación 39

# Implementación del TAD de los grafos mediante diccionarios

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

-- El objetivo de esta relación es implementar el TAD de los grafos
-- mediante diccionarios, de manera análoga a las implementaciones
-- estudiadas en el tema 22 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-22.pdf
-- y usando la mismas signatura.

-- ---------------------------------------------------------------------
-- Signatura                                                           --
-- ---------------------------------------------------------------------

module Rel_39_sol
    (Orientacion (..),
     Grafo,
     creaGrafo,  -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                 --                 Grafo v p
     dirigido,   -- (Ix v,Num p) => (Grafo v p) -> Bool
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristas,    -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where

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

```haskell
-- Librerías auxiliares                                                --
-- ---------------------------------------------------------------------

import Data.List
import Data.Ix
import qualified Data.Map as M

-- ---------------------------------------------------------------------
-- Representación de los grafos mediante diccionarios                  --
-- ---------------------------------------------------------------------

-- Orientacion es D (dirigida) ó ND (no dirigida).
data Orientacion = D | ND
  deriving (Eq, Show)

-- (Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p.
data Grafo v p = G Orientacion (M.Map v [(v,p)])
  deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicios                                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
-- tal que (creaGrafo o cs as) es un grafo (dirigido o no, según el
-- valor de o), con el par de cotas cs y listas de aristas as (cada
-- arista es un trío formado por los dos vértices y su peso). Por
-- ejemplo,
--    ghci> creaGrafo ND (1,3) [(1,2,12),(1,3,34)]
--    G ND (fromList [(1,[(2,12),(3,34)]),(2,[(1,12)]),(3,[(1,34)])])
--    ghci> creaGrafo D (1,3) [(1,2,12),(1,3,34)]
--    G D (fromList [(1,[(2,12),(3,34)])])
--    ghci> creaGrafo D (1,4) [(1,2,12),(1,3,34)]
--    G D (fromList [(1,[(2,12),(3,34)])])
-- ---------------------------------------------------------------------

creaGrafo :: (Ix v, Num p) =>
             Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
```

```
creaGrafo o cs vs = G o (foldr f dInicial zs)
    where f (v1,(v2,p)) = M.insertWith (++) v1 [(v2,p)]
          zs = (if o == D then []
                else [(x2,(x1,p))|(x1,x2,p) <- vs, x1 /= x2]) ++
               [(x1,(x2,p)) | (x1,x2,p) <- vs]
          xs = [x1 | (x1,x2,p) <- vs] `union` [x2 | (x1,x2,p) <- vs]
          dInicial = foldr (\y d -> M.insert y [] d) M.empty xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, con creaGrafo, la constante
--    ejGrafoND :: Grafo Int Int
-- para representar el siguiente grafo no dirigido
--              12
--        1 -------- 2
--        | \78     /|
--        |  \   32/ |
--        |   \   /  |
--      34|     5    |55
--        |   /   \  |
--        |  /44   \ |
--        | /     93\|
--        3 -------- 4
--             61
--    ghci> ejGrafoND
--    G ND (fromList [(1,[(2,12),(3,34),(5,78)]),
--                    (2,[(1,12),(4,55),(5,32)]),
--                    (3,[(1,34),(4,61),(5,44)]),
--                    (4,[(2,55),(3,61),(5,93)]),
--                    (5,[(1,78),(2,32),(3,44),(4,93)])])
-- ---------------------------------------------------------------------

ejGrafoND :: Grafo Int Int
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                (2,4,55),(2,5,32),
                                (3,4,61),(3,5,44),
                                (4,5,93)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, con creaGrafo, la constante
--    ejGrafoD :: Grafo Int Int
```

```
-- para representar el grafo anterior donde se considera que las aristas
-- son los pares (x,y) con x < y. Por ejemplo,
--    ghci> ejGrafoD
--    G D (fromList [(1,[(2,12),(3,34),(5,78)]),
--                   (2,[(4,55),(5,32)]),
--                   (3,[(4,61),(5,44)]),
--                   (4,[(5,93)])])

-- ---------------------------------------------------------------------

ejGrafoD :: Grafo Int Int
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                              (2,4,55),(2,5,32),
                              (3,4,61),(3,5,44),
                              (4,5,93)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
-- tal que (dirigido g) se verifica si g es dirigido. Por ejemplo,
--    dirigido ejGrafoD   ==  True
--    dirigido ejGrafoND  ==  False
-- ---------------------------------------------------------------------

dirigido :: (Ix v,Num p) => Grafo v p -> Bool
dirigido (G o _) = o == D

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
-- tal que (nodos g) es la lista de todos los nodos del grafo g. Por
-- ejemplo,
--    nodos ejGrafoND  ==  [1,2,3,4,5]
--    nodos ejGrafoD   ==  [1,2,3,4,5]
-- ---------------------------------------------------------------------

nodos :: (Ix v,Num p) => Grafo v p -> [v]
nodos (G _ d) = M.keys d

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

```haskell
-- Ejercicio 6. Definir la función
--    adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
-- tal que (adyacentes g v) es la lista de los vértices adyacentes al
-- nodo v en el grafo g. Por ejemplo,
--    adyacentes ejGrafoND 4  ==  [5,2,3]
--    adyacentes ejGrafoD  4  ==  [5]
-- ---------------------------------------------------------------------

adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adyacentes (G _ d) v = map fst (d M.! v)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
-- (aristaEn g a) se verifica si a es una arista del grafo g. Por
-- ejemplo,
--    aristaEn ejGrafoND (5,1)  ==  True
--    aristaEn ejGrafoND (4,1)  ==  False
--    aristaEn ejGrafoD  (5,1)  ==  False
--    aristaEn ejGrafoD  (1,5)  ==  True
-- ---------------------------------------------------------------------

aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
-- tal que (peso v1 v2 g) es el peso de la arista que une los vértices
-- v1 y v2 en el grafo g. Por ejemplo,
--    peso 1 5 ejGrafoND  ==  78
--    peso 1 5 ejGrafoD   ==  78
-- ---------------------------------------------------------------------

peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
peso x y (G _ g) = head [c | (a,c) <- g M.! x, a == y]

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
```

```haskell
-- (aristasD g) es la lista de las aristas del grafo g. Por ejemplo,
--    ghci> aristas ejGrafoD
--    [(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
--     (3,5,44),(4,5,93)]
--    ghci> aristas ejGrafoND
--    [(1,2,12),(1,3,34),(1,5,78),(2,1,12),(2,4,55),(2,5,32),
--     (3,1,34),(3,4,61),(3,5,44),(4,2,55),(4,3,61),(4,5,93),
--     (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
-- ----------------------------------------------------------------------

aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
aristas (G o g) = [(v1,v2,w) | v1 <- nodos (G o g) , (v2,w) <- g M.! v1]
```

# Relación 40

# Problemas básicos con el TAD de los grafos

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

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de los grafos estudiado en el tema 22
--    http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-22.html
--
-- Para realizar los ejercicios hay que tener instalada la librería I1M
-- que se encuentra en http://hackage.haskell.org/package/I1M
--
-- Para instalar la librería de I1M, basta ejecutar en una consola
--    cabal update
--    cabal install I1M
--
-- Otra forma es descargar, en el directorio de ejercicios, la
-- implementación del TAD de grafos
-- + GrafoConVectorDeAdyacencia que está en http://bit.ly/1SQnG4S
-- + GrafoConMatrizDeAdyacencia que está en http://bit.ly/1SQnGlB
--
-- Los módulos anteriores se encuentras en la página de códigos
-- http://bit.ly/1SQnAKO

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

{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}
```

```haskell
{-# OPTIONS_GHC -fno-warn-orphans #-}

module Rel_38_sol where

import Data.Array
import Data.List (nub)
import Test.QuickCheck

-- Hay que elegir una librería
import I1M.Grafo
-- import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia

-- ---------------------------------------------------------------------
-- Ejemplos                                                            --
-- ---------------------------------------------------------------------

-- Para los ejemplos se usarán los siguientes grafos.
g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 :: Grafo Int Int
g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                         (2,4,55),(2,5,32),
                         (3,4,61),(3,5,44),
                         (4,5,93)]
g2 = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                        (2,4,55),(2,5,32),
                        (4,3,61),(4,5,93)]
g3 = creaGrafo D (1,3) [(1,2,0),(2,2,0),(3,1,0),(3,2,0)]
g4 = creaGrafo D (1,4) [(1,2,3),(2,1,5)]
g5 = creaGrafo D (1,1) [(1,1,0)]
g6 = creaGrafo D (1,4) [(1,3,0),(3,1,0),(3,3,0),(4,2,0)]
g7 = creaGrafo ND (1,4) [(1,3,0)]
g8 = creaGrafo D (1,5) [(1,1,0),(1,2,0),(1,3,0),(2,4,0),(3,1,0),
                        (4,1,0),(4,2,0),(4,4,0),(4,5,0)]
g9 = creaGrafo D (1,5) [(4,1,1),(4,3,2),(5,1,0)]
g10 = creaGrafo ND (1,3) [(1,2,1),(1,3,1),(2,3,1),(3,3,1)]
g11 = creaGrafo D (1,3) [(1,2,1),(1,3,1),(2,3,1),(3,3,1)]
g12 = creaGrafo ND (1,4) [(1,1,0),(1,2,0),(3,3,0)]

-- ---------------------------------------------------------------------
-- Ejercicio 1. El grafo completo de orden n, K(n), es un grafo no
```

```
-- dirigido cuyos conjunto de vértices es {1,..n} y tiene una arista
-- entre cada par de vértices distintos. Definir la función,
--    completo :: Int -> Grafo Int Int
-- tal que (completo n) es el grafo completo de orden n. Por ejemplo,
--    ghci> completo 4
--    G ND (array (1,4) [(1,[(2,0),(3,0),(4,0)]),
--                       (2,[(1,0),(3,0),(4,0)]),
--                       (3,[(1,0),(2,0),(4,0)]),
--                       (4,[(1,0),(2,0),(3,0)])])
-- ------------------------------------------------------------------

completo :: Int -> Grafo Int Int
completo n =
  creaGrafo ND (1,n) [(x,y,0) | x <- [1..n], y <- [x+1..n]]

-- ------------------------------------------------------------------
-- Ejercicio 2. El ciclo de orden n, C(n), es un grafo no dirigido
-- cuyo conjunto de vértices es {1,...,n} y las aristas son
--    (1,2), (2,3), ..., (n-1,n), (n,1)
-- Definir la función
--    grafoCiclo :: Int -> Grafo Int Int
-- tal que (grafoCiclo n) es el grafo ciclo de orden n. Por ejemplo,
--    ghci> grafoCiclo 3
--    G ND (array (1,3) [(1,[(3,0),(2,0)]),(2,[(1,0),(3,0)]),(3,[(2,0),(1,0)])])
-- ------------------------------------------------------------------

grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n =
  creaGrafo ND (1,n) ((n,1,0):[(x,x+1,0) | x <- [1..n-1]])

-- ------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    nVertices :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nVertices g) es el número de vértices del grafo g. Por
-- ejemplo,
--    nVertices (completo 4)  ==  4
--    nVertices (completo 5)  ==  5
-- ------------------------------------------------------------------

nVertices :: (Ix v,Num p) => Grafo v p ->  Int
```

```haskell
nVertices = length . nodos

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    noDirigido :: (Ix v,Num p) => Grafo v p ->  Bool
-- tal que (noDirigido g) se verifica si el grafo g es no dirigido. Por
-- ejemplo,
--    noDirigido g1            ==  True
--    noDirigido g2            ==  False
--    noDirigido (completo 4)  ==  True
-- ---------------------------------------------------------------------

noDirigido :: (Ix v,Num p) => Grafo v p ->  Bool
noDirigido = not . dirigido

-- ---------------------------------------------------------------------
-- Ejercicio 5. En un un grafo g, los incidentes de un vértice v es el
-- conjuntos de vértices x de g para los que hay un arco (o una arista)
-- de x a v; es decir, que v es adyacente a x. Definir la función
--    incidentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
-- tal que (incidentes g v) es la lista de los vértices incidentes en el
-- vértice v. Por ejemplo,
--    incidentes g2 5  ==  [1,2,4]
--    adyacentes g2 5  ==  []
--    incidentes g1 5  ==  [1,2,3,4]
--    adyacentes g1 5  ==  [1,2,3,4]
-- ---------------------------------------------------------------------

incidentes :: (Ix v,Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

-- ---------------------------------------------------------------------
-- Ejercicio 6. En un un grafo g, los contiguos de un vértice v es el
-- conjuntos de vértices x de g tales que x es adyacente o incidente con
-- v. Definir la función
--    contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
-- tal que (contiguos g v) es el conjunto de los vértices de g contiguos
-- con el vértice v. Por ejemplo,
--    contiguos g2 5  ==  [1,2,4]
--    contiguos g1 5  ==  [1,2,3,4]
```

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

contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
contiguos g v = nub (adyacentes g v ++ incidentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
-- tal que (lazos g) es el conjunto de los lazos (es decir, aristas
-- cuyos extremos son iguales) del grafo g. Por ejemplo,
--    ghci> lazos g3
--    [(2,2)]
--    ghci> lazos g2
--    []
-- ---------------------------------------------------------------------

lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
lazos g = [(x,x) | x <- nodos g, aristaEn g (x,x)]

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    nLazos :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nLazos g) es el número de lazos del grafo g. Por
-- ejemplo,
--    nLazos g3  ==  1
--    nLazos g2  ==  0
-- ---------------------------------------------------------------------

nLazos :: (Ix v,Num p) => Grafo v p ->  Int
nLazos = length . lazos

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    nAristas :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nAristas g) es el número de aristas del grafo g. Si g es no
-- dirigido, las aristas de v1 a v2 y de v2 a v1 sólo se cuentan una
-- vez. Por ejemplo,
--    nAristas g1            ==  8
--    nAristas g2            ==  7
--    nAristas g10           ==  4
```

```
--    nAristas g12           ==  3
--    nAristas (completo 4)  ==  6
--    nAristas (completo 5)  ==  10
-- ---------------------------------------------------------------------

nAristas :: (Ix v,Num p) => Grafo v p ->  Int
nAristas g | dirigido g = length (aristas g)
           | otherwise  = (length (aristas g) + nLazos g) `div` 2

-- 2ª definición
nAristas2 :: (Ix v,Num p) => Grafo v p ->  Int
nAristas2 g | dirigido g = length (aristas g)
            | otherwise  = length [(x,y) | (x,y,_) <- aristas g, x <= y]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    prop_nAristasCompleto :: Int -> Bool
-- tal que (prop_nAristasCompleto n) se verifica si el número de aristas
-- del grafo completo de orden n es n*(n-1)/2 y, usando la función,
-- comprobar que la propiedad se cumple para n de 1 a 20.
-- ---------------------------------------------------------------------

prop_nAristasCompleto :: Int -> Bool
prop_nAristasCompleto n =
  nAristas (completo n) == n*(n-1) `div` 2

-- La comprobación es
--    ghci> and [prop_nAristasCompleto n | n <- [1..20]]
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 11. El grado positivo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g adyacentes con v. Definir la función
--    gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (gradoPos g v) es el grado positivo del vértice v en el grafo
-- g. Por ejemplo,
--    gradoPos g1 5  ==  4
--    gradoPos g2 5  ==  0
--    gradoPos g2 1  ==  3
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)

-- 2ª definición
gradoPos2 :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos2 g = length . adyacentes g

-- ---------------------------------------------------------------------
-- Ejercicio 12. El grado negativo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g incidentes con v. Definir la función
--    gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (gradoNeg g v) es el grado negativo del vértice v en el grafo
-- g. Por ejemplo,
--    gradoNeg g1 5  ==  4
--    gradoNeg g2 5  ==  3
--    gradoNeg g2 1  ==  0
-- ---------------------------------------------------------------------

gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 13. El grado de un vértice v de un grafo dirigido g, es el
-- número de aristas de g que contiene a v. Si g es no dirigido, el
-- grado de un vértice v es el número de aristas incidentes en v, teniendo
-- en cuenta que los lazos se cuentan dos veces. Definir la función
--    grado :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (grado g v) es el grado del vértice v en el grafo g. Por
-- ejemplo,
--    grado g1 5  ==  4
--    grado g2 5  ==  3
--    grado g2 1  ==  3
--    grado g3 2  ==  4
--    grado g3 1  ==  2
--    grado g3 3  ==  2
--    grado g5 1  ==  2
--    grado g10 3 ==  4
--    grado g11 3 ==  4
```

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

grado :: (Ix v,Num p) => Grafo v p -> v -> Int
grado g v | dirigido g           = gradoNeg g v + gradoPos g v
          | (v,v) `elem` lazos g = length (incidentes g v) + 1
          | otherwise            = length (incidentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 14. Comprobar con QuickCheck que para cualquier grafo g, la
-- suma de los grados positivos de los vértices de g es igual que la
-- suma de los grados negativos de los vértices de g.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaGrados :: Grafo Int Int -> Bool
prop_sumaGrados g =
  sum [gradoPos g v | v <- vs] == sum [gradoNeg g v | v <- vs]
  where vs = nodos g

-- La comprobación es
--    ghci> quickCheck prop_sumaGrados
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 15. En la teoría de grafos, se conoce como "Lema del
-- apretón de manos" la siguiente propiedad: la suma de los grados de
-- los vértices de g es el doble del número de aristas de g.
-- Comprobar con QuickCheck que para cualquier grafo g, se verifica
-- dicha propiedad.
-- ---------------------------------------------------------------------

prop_apretonManos :: Grafo Int Int -> Bool
prop_apretonManos g =
  sum [grado g v | v <- nodos g] == 2 * nAristas g

-- La comprobación es
--    ghci> quickCheck prop_apretonManos
--    +++ OK, passed 100 tests.

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

```haskell
-- Ejercicio 16. Comprobar con QuickCheck que en todo grafo, el número
-- de nodos de grado impar es par.
-- ------------------------------------------------------------------

prop_numNodosGradoImpar :: Grafo Int Int -> Bool
prop_numNodosGradoImpar g = even m
  where vs = nodos g
        m = length [v | v <- vs, odd (grado g v)]

-- La comprobación es
--    ghci> quickCheck prop_numNodosGradoImpar
--    +++ OK, passed 100 tests.

-- ------------------------------------------------------------------
-- Ejercicio 17. Definir la propiedad
--   prop_GradoCompleto :: Int -> Bool
-- tal que (prop_GradoCompleto n) se verifica si todos los vértices del
-- grafo completo K(n) tienen grado n-1. Usarla para comprobar que dicha
-- propiedad se verifica para los grafos completos de grados 1 hasta 30.
-- ------------------------------------------------------------------

prop_GradoCompleto :: Int -> Bool
prop_GradoCompleto n =
  and [grado g v == (n-1) | v <- nodos g]
  where g = completo n

-- La comprobación es
--    ghci> and [prop_GradoCompleto n | n <- [1..30]]
--    True

-- ------------------------------------------------------------------
-- Ejercicio 18. Un grafo es regular si todos sus vértices tienen el
-- mismo grado. Definir la función
--    regular :: (Ix v,Num p) => Grafo v p -> Bool
-- tal que (regular g) se verifica si todos los nodos de g tienen el
-- mismo grado.
--    regular g1            ==  False
--    regular g2            ==  False
--    regular (completo 4)  ==  True
-- ------------------------------------------------------------------
```

```haskell
regular :: (Ix v,Num p) => Grafo v p -> Bool
regular g = and [grado g v == k | v <- vs]
  where vs = nodos g
        k  = grado g (head vs)

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la propiedad
--    prop_CompletoRegular :: Int -> Int -> Bool
-- tal que (prop_CompletoRegular m n) se verifica si todos los grafos
-- completos desde el de orden m hasta el de orden n son regulares y
-- usarla para comprobar que todos los grafos completo desde el de orden
-- 1 hasta el de orden 30 son regulares.
-- ---------------------------------------------------------------------

prop_CompletoRegular :: Int -> Int -> Bool
prop_CompletoRegular m n =
  and [regular (completo x) | x <- [m..n]]

-- La comprobación es
--    ghci> prop_CompletoRegular 1 30
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 20. Un grafo es k-regular si todos sus vértices son de
-- grado k. Definir la función
--    regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
-- tal que (regularidad g) es la regularidad de g. Por ejemplo,
--    regularidad g1              ==  Nothing
--    regularidad (completo 4)    ==  Just 3
--    regularidad (completo 5)    ==  Just 4
--    regularidad (grafoCiclo 4)  ==  Just 2
--    regularidad (grafoCiclo 5)  ==  Just 2
-- ---------------------------------------------------------------------

regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
regularidad g
  | regular g = Just (grado g (head (nodos g)))
  | otherwise = Nothing
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la propiedad
--    prop_completoRegular :: Int -> Bool
-- tal que (prop_completoRegular n) se verifica si el grafo completo de
-- orden n es (n-1)-regular. Por ejemplo,
--    prop_completoRegular 5  ==  True
-- y usarla para comprobar que la cumplen todos los grafos completos
-- desde orden 1 hasta 20.
-- ---------------------------------------------------------------------

prop_completoRegular :: Int -> Bool
prop_completoRegular n =
  regularidad (completo n) == Just (n-1)

-- La comprobación es
--    ghci> and [prop_completoRegular n | n <- [1..20]]
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir la propiedad
--    prop_cicloRegular :: Int -> Bool
-- tal que (prop_cicloRegular n) se verifica si el grafo ciclo de orden
-- n es 2-regular. Por ejemplo,
--    prop_cicloRegular 2  ==  True
-- y usarla para comprobar que la cumplen todos los grafos ciclos
-- desde orden 3 hasta 20.
-- ---------------------------------------------------------------------

prop_cicloRegular :: Int -> Bool
prop_cicloRegular n =
  regularidad (grafoCiclo n) == Just 2

-- La comprobación es
--    ghci> and [prop_cicloRegular n | n <- [3..20]]
--    True

-- ---------------------------------------------------------------------
-- § Generador de grafos                                             --
-- ---------------------------------------------------------------------
```

```haskell
-- (generaGND n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--    ghci> generaGND 3 [4,2,5]
--    (ND,array (1,3) [(1,[(2,4),(3,2)]),
--                     (2,[(1,4),(3,5)]),
--                      3,[(1,2),(2,5)])])
--    ghci> generaGND 3 [4,-2,5]
--    (ND,array (1,3) [(1,[(2,4)]),(2,[(1,4),(3,5)]),(3,[(2,5)])])
generaGND :: Int -> [Int] -> Grafo Int Int
generaGND n ps  = creaGrafo ND (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n], x < y]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- (generaGD n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--    ghci> generaGD 3 [4,2,5]
--    (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                    (2,[]),
--                    (3,[])])
--    ghci> generaGD 3 [4,2,5,3,7,9,8,6]
--    (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                    (2,[(1,3),(2,7),(3,9)]),
--                    (3,[(1,8),(2,6)])])
generaGD :: Int -> [Int] -> Grafo Int Int
generaGD n ps = creaGrafo D (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n]]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- genGD es un generador de grafos dirigidos. Por ejemplo,
--    ghci> sample genGD
--    (D,array (1,4) [(1,[(1,1)]),(2,[(3,1)]),(3,[(2,1),(4,1)]),(4,[(4,1)])])
--    (D,array (1,2) [(1,[(1,6)]),(2,[])])
--    ...
genGD :: Gen (Grafo Int Int)
genGD = do n <- choose (1,10)
           xs <- vectorOf (n*n) arbitrary
           return (generaGD n xs)
```

```haskell
-- genGND es un generador de grafos dirigidos. Por ejemplo,
--    ghci> sample genGND
--    (ND,array (1,1) [(1,[])])
--    (ND,array (1,3) [(1,[(2,3),(3,13)]),(2,[(1,3)]),(3,[(1,13)])])
--    ...
genGND :: Gen (Grafo Int Int)
genGND = do n <- choose (1,10)
            xs <- vectorOf (n*n) arbitrary
            return (generaGND n xs)

-- genG es un generador de grafos. Por ejemplo,
--    ghci> sample genG
--    (D,array (1,3) [(1,[(2,1)]),(2,[(1,1),(2,1)]),(3,[(3,1)])])
--    (ND,array (1,3) [(1,[(2,2)]),(2,[(1,2)]),(3,[])])
--    ...
genG :: Gen (Grafo Int Int)
genG = do d <- choose (True,False)
          n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          if d then return (generaGD n xs)
               else return (generaGND n xs)

-- Los grafos está contenido en la clase de los objetos generables
-- aleatoriamente.
instance Arbitrary (Grafo Int Int) where
  arbitrary = genG
```