Ejercicios de “Informática de 1ºde Matemáticas” (2011–12)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 1 de Octubre de 2011 (Versión de 30 de mayo de 2012)

Índice general

1 Definiciones elementales de funciones (1) 7

2 Definiciones elementales de funciones (2) 13

3 Definiciones por comprensión (1) 23

4 Definiciones por comprensión (2) 29

5 Definiciones por comprensión (3): El cifrado César 39

6 Definiciones por recursión 45

7 Definiciones por recursión y por comprensión (1) 53

8 Definiciones por recursión y por comprensión (2) 69

9 Definiciones sobre cadenas, orden superior y plegado 81

10 Definiciones por plegado 99

11 Codificación y transmisión de mensajes 107

12 Resolución de problemas matemáticos 113

13 Demostración de propiedades por inducción 125

14 El 2011 y los números primos 133

15 Listas infinitas 141

16 Ejercicios de exámenes del curso 2010-11 149

17 Combinatoria 155

18 Tipos de datos algebraicos 173

19 Tipos de datos algebraicos: árboles binarios **179**

20 Tipos de datos algebraicos: fórmulas proposicionales **187**

21 Tipos de datos algebraicos: Modelización de juego de cartas **191**

22 Cálculo numérico **201**

23 Ecuación con factoriales **211**

24 Aplicaciones de la programación funcional con listas infinitas **215**

25 División y factorización de polinomios mediante la regla de Ruffini **221**

26 Operaciones con el TAD de polinomios **229**

27 Operaciones con vectores y matrices **237**

28 Ejercicios complementarios **255**

29 Relaciones binarias **271**

30 Operaciones con conjuntos **279**

31 Implementación del TAD de los grafos mediante listas **299**

32 Problemas básicos con el TAD de los grafos **305**

33 Enumeraciones de los números racionales **319**

A Exámenes **327**

A.1 Examen 1 (26 de Octubre de 2011) 327
A.2 Examen 2 (30 de Noviembre de 2011) 328
A.3 Examen 3 (25 de Enero de 2012) 330
A.4 Examen 4 (29 de Febrero de 2012) 334
A.5 Examen 5 (21 de Marzo de 2012) 337
A.6 Examen 6 (2 de Mayo de 2012) 340

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 2011–12.

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-11/temas/2011-12-IM-temas-PF.pdf

[2] http://www.glc.us.es/~jalonso/ejerciciosI1M2011G1

Relación 1

Definiciones elementales de funciones (1)

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

-- En esta relación se plantean ejercicios con definiciones
-- elementales (no recursivas) de funciones para ejercitar la
-- introducción a Haskell presentada en el tema 2 y cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-2t.pdf
-- Para solucionar los ejercicios puede ser útil el "Resumen de
-- funciones de Haskell" que se encuentra en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/doc/resumen_Haskell.pdf
-- En concreto, se estudian funciones para calcular
-- * la media de 3 números,
-- * la suma de euros de una colección de monedas,
-- * el volumen de la esfera,
-- * el área de una corona circular,
-- * la intercalación de pares,
-- * la última cifra de un número,
-- * la rotación de listas,
-- * el rango de una lista,
-- * el reconocimiento de palíndromos,
-- * la igualdad y diferencia de 3 elementos,
-- * la igualdad de 4 elementos,
-- * el máximo de 3 elementos,
```

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función ultimaCifra tal que (ultimaCifra x)
-- es la última cifra del nímero x. Por ejemplo,
--    ultimaCifra 325  ==  5
-- ---------------------------------------------------------------------

ultimaCifra x = rem x 10

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

rota1 xs = tail xs ++ [head xs]

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

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

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

rango xs = [minimum xs, maximum xs]

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

palindromo xs = xs == reverse xs

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función divisionSegura tal que
-- (divisionSegura x y) es x/y si y no es cero e y 9999 en caso
-- contrario. Por ejemplo,
--    divisionSegura 7 2  ==  3.5
--    divisionSegura 7 0  ==  9999.0
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 16. En geometría, la fórmula de Herón, descubierta por
-- Herón de Alejandría, dice que el área de un triángulo cuyo lados
-- miden a, b y c es la raíz cuadrada de s(s-a)(s-b)(s-c) donde s es el
-- semiperímetro
--    s = (a+b+c)/2
```

```
-- Definir la función area tal que (area a b c) es el área de un
-- triángulo de lados a, b y c. Por ejemplo,
--    area 3 4 5  ==  6.0
-- ---------------------------------------------------------------------

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

Relación 2

Definiciones elementales de funciones (2)

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

-- En esta relación se presentan ejercicios con definiciones elementales
-- (no recursivas) de funciones correspondientes al tema 4 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-4t.pdf
-- En concreto, se estudian funciones para calcular
-- * el módulo de un vector,
-- * el cuadrante de un punto,
-- * el intercambio de coordenadas,
-- * el punto simétrico,
-- * las raíces de las ecuaciones cuadráticas y
-- * la disyunción excluyente,
-- * los finales de una lista,
-- * los segmentos de una lista,
-- * el mediano de 3 números,
-- * la distancia entre dos puntos,
-- * los extremos de una lista,
-- * el punto medio entre otros dos,
-- * la permutación cíclica de una lista,
-- * el mayor número de 2 cifra con dos dígitos dados,
-- * la propiedad triangular,
-- * la forma reducida de un número racional,
```

```
-- * la suma de dos números racionales,
-- * el producto de dos números racionales,
-- * la propiedad de igualdad de números racionales,
-- * la suma de dos números complejos,
-- * el producto de dos números complejos y
-- * el conjugado de un número complejo.

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función modulo tal que (modulo v) es el
-- módulo del vector v. Por ejemplo,
--    modulo (3,4)  ==  5.0
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función cuadrante tal que (cuadrante p) es
-- es cuadrante del punto p (se supone que p no está sobre los
-- ejes). Por ejemplo,
--    cuadrante (3,5)    ==  1
--    cuadrante (-3,5)   ==  2
--    cuadrante (-3,-5)  ==  3
--    cuadrante (3,-5)   ==  4
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función intercambia tal que (intercambia p)
-- es el punto obtenido intercambiando las coordenadas del punto p. Por
-- ejemplo,
--    intercambia (2,5)  ==  (5,2)
--    intercambia (5,2)  ==  (2,5)
-- ---------------------------------------------------------------------

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función simetricoH tal que (simetricoH p) es
-- el punto simétrico de p respecto del eje horizontal. Por ejemplo,
--    simetricoH (2,5)   ==  (2,-5)
--    simetricoH (2,-5)  ==  (2,5)
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. (Raíces de una ecuación de segundo grado) Definir la
-- función raices de forma que (raices a b c) devuelve la lista de las
-- raices reales de la ecuación  ax^2 + bx + c = 0. Por ejemplo,
--    raices 1 (-2) 1 == [1.0,1.0]
--    raices 1 3 2    == [-1.0,-2.0]
-- ---------------------------------------------------------------------

-- 1ª solución
raices_1 a b c = [(-b+d)/t,(-b-d)/t]
    where d = sqrt (b^2 - 4*a*c)
          t = 2*a

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. La disyunción excluyente xor de dos fórmulas se verifica
-- si una es verdadera y la otra es falsa.
-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir la función xor_1 que calcule la disyunción
-- excluyente a partir de la tabla de verdad. Usar 4 ecuaciones, una por
-- cada línea de la tabla.
-- ---------------------------------------------------------------------

xor_1 :: Bool -> Bool -> Bool
```

```
xor_1 True  True  = False
xor_1 True  False = True
xor_1 False True  = True
xor_1 False False = False

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir la función xor_2 que calcule la disyunción
-- excluyente a partir de la tabla de verdad y patrones. Usar 2
-- ecuaciones, una por cada valor del primer argumento.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir la función xor_3 que calcule la disyunción
-- excluyente a partir de la disyunción (||), conjunción (&&) y negación
-- (not). Usar 1 ecuación.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.4. Definir la función xor_4 que calcule la disyunción
-- excluyente a partir de desigualdad (/=). Usar 1 ecuación.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. 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 8. 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 9. 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
-- ---------------------------------------------------------------------

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

-- Otra solución es
mediano' x y z
    | a <= x && x <= b = x
    | a <= y && y <= b = y
    | otherwise        = z
    where a = minimum [x,y,z]
          b = maximum [x,y,z]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función distancia tal que (distancia p1 p2)
-- es la distancia entre los puntos p1 y p2. Por ejemplo,
--    distancia (1,2) (4,6)  ==  5.0
-- ---------------------------------------------------------------------

distancia (x1,y1) (x2,y2) = sqrt((x1-x2)^2+(y1-y2)^2)

-- ---------------------------------------------------------------------
-- Ejercicio 11. 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 12. Definir la función puntoMedio tal que (puntoMedio p1 p2)
-- es el punto medio entre los puntos p1 y p2. Por ejemplo,
--    puntoMedio (0,2) (0,6)  ==  (0.0,4.0)
--    puntoMedio (-1,2) (7,6) ==  (3.0,4.0)
-- ---------------------------------------------------------------------

puntoMedio (x1,y1) (x2,y2) = ((x1+x2)/2, (y1+y2)/2)

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir una función ciclo que permute cíclicamente los
-- elementos de una lista, pasando el último elemento al principio de la
-- lista. Por ejemplo,
--    ciclo [2, 5, 7, 9]  == [9,2,5,7]
--    ciclo []            == [9,2,5,7]
--    ciclo [2]           == [2]
-- ---------------------------------------------------------------------

ciclo [] = []
ciclo xs = last xs : init xs

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la funcion numeroMayor 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
-- ---------------------------------------------------------------------

numeroMayor x y = a*10 + b
    where a = max x y
          b = min x y
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 15. Las longitudes de los lados de un triángulo no pueden
-- ser cualesquiera. Para que pueda construirse el triángulo, tiene que
-- cumplirse la propiedad triangular; es decir, longitud de cada lado
-- tiene que ser menor que la suma de los otros dos lados.
--
-- Definir la función triangular tal que (triangular a b c) se verifica
-- si a, b y c complen la propiedad triangular. Por ejemplo,
--    triangular 3 4 5   ==  True
--    triangular 30 4 5  ==  False
--    triangular 3 40 5  ==  False
--    triangular 3 4 50  ==  False
-- ---------------------------------------------------------------------

triangular a b c = a < b+c && b < a+c && c < a+b

-- ---------------------------------------------------------------------
-- Ejercicio 16. Los números racionales pueden representarse mediante
-- pares de números enteros. Por ejemplo, el número 2/5 puede
-- representarse mediante el par (2,5).
-- ---------------------------------------------------------------------
-- Ejercicio 16.1. Definir la función formaReducida tal que
-- (formaReducida x) es la forma reducida del número racional x. Por
-- ejemplo,
--    formaReducida (4,10)  ==  (2,5)
-- ---------------------------------------------------------------------

formaReducida (a,b) = (a `div` c, b  `div` c)
    where c = gcd a b

-- ---------------------------------------------------------------------
-- Ejercicio 16.2. Definir la función sumaRacional tal que
-- (sumaRacional x y) es la suma de los números racionales x e y. Por ejemplo,
--    sumaRacional (2,3) (5,6)  ==  (3,2)
-- ---------------------------------------------------------------------

sumaRacional (a,b) (c,d) = formaReducida (a*d+b*c, b*d)

-- ---------------------------------------------------------------------
-- Ejercicio 16.3. Definir la función productoRacional tal que
```

```
-- (productoRacional x y) es el producto de los números racionales x e
-- y. Por ejemplo,
--    productoRacional (2,3) (5,6)  ==  (5,9)
-- ---------------------------------------------------------------------

productoRacional (a,b) (c,d) = formaReducida (a*c, b*d)

-- ---------------------------------------------------------------------
-- Ejercicio 16.4. Definir la función igualdadRacional 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 (a,b) (c,d) =
    formaReducida (a,b) == formaReducida (c,d)

-- ---------------------------------------------------------------------
-- Ejercicio 17. Los números complejos pueden representarse mediante
-- pares de números complejos. Por ejemplo, el número 2+5i puede
-- representarse mediante el par (2,5).
-- ---------------------------------------------------------------------
-- Ejercicio 17.1. Definir la función sumaComplejos tal que
-- (sumaComplejos x y) es la suma de los números complejos x e y. Por
-- ejemplo,
--    sumaComplejos (2,3) (5,6)  ==  (7,9)
-- ---------------------------------------------------------------------

sumaComplejos (a,b) (c,d) = (a+c, b+d)

-- ---------------------------------------------------------------------
-- Ejercicio 17.2. Definir la función productoComplejos tal que
-- (productoComplejos x y) es el producto de los números complejos x e
-- y. Por ejemplo,
--    productoComplejos (2,3) (5,6)  ==  (-8,27)
-- ---------------------------------------------------------------------

productoComplejos (a,b) (c,d) = (a*c-b*d, a*d+b*c)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 17.3. Definir la función conjugado tal que (conjugado x) es
-- el conjugado del número complejo z. Por ejemplo,
--    conjugado (2,3) ==  (2,-3)
-- ---------------------------------------------------------------------

conjugado (a,b) = (a,-b)
```

Relación 3

Definiciones por comprensión (1)

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

-- En esta relación se presentan ejercicios con definiciones por
-- comprensión correspondientes al tema 5 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf
-- En concreto, se estudian funciones para calcular
-- * la suma de los cuadrados de los n primeros números,
-- * listas con un elemento replicado,
-- * ternas pitagóricas,
-- * números perfectos,
-- * producto cartesiano,
-- * posiciones de un elemento en una lista,
-- * producto escalar y
-- * la solución del problema 1 del proyecto Euler.

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un entero positivo es perfecto si es igual a la suma de
-- sus factores, excluyendo el propio número.
--
```

```
-- Definir por comprensión la función
--    perfectos :: Int -> [Int]
-- tal que (perfectos n) es la lista de todos los números perfectos
-- menores que n. Por ejemplo,
--    perfectos 500  ==  [6,28,496]
-- Indicación: Usar la función factores del tema 5.
-- ---------------------------------------------------------------------

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. En el tema se ha definido la función
--    posiciones :: Eq a => a -> [a] -> [Int]
-- tal que (posiciones x xs) es la lista de las posiciones ocupadas por
-- el elemento x en la lista xs. Por ejemplo,
--    posiciones 5 [1,5,3,5,5,7]  ==  [1,3,4]
--
-- Definir, usando la función busca (definida en el tema 5), la función
--    posiciones' :: Eq a => a -> [a] -> [Int]
-- tl que posiciones' sea equivalente a posiciones.
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. El producto escalar de dos listas de enteros xs y ys de
-- longitud n viene dado por la suma de los productos de los elementos
-- correspondientes.
--
-- Definir por comprensión la función
--    productoEscalar :: [Int] -> [Int] -> Int
-- tal que (productoEscalar xs ys) es el producto escalar de las listas
-- xs e ys. Por ejemplo,
--    productoEscalar [1,2,3] [4,5,6]  ==  32
```

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

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

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

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

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

Relación 4

Definiciones por comprensión (2)

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

-- En esta relación se presentan más ejercicios con definiciones por
-- comprensión correspondientes al tema 5 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función errorE tal que (errorE x) es el
```

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

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

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

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

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

-- El límite es 1.

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función errorLimSeno tal que
-- (errorLimSeno x) es el menor número de términos de la sucesión
-- sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
-- que x. Por ejemplo,
--    errorLimSeno 0.1     ==   2.0
--    errorLimSeno 0.01    ==   5.0
--    errorLimSeno 0.001   ==  13.0
--    errorLimSeno 0.0001  ==  41.0
-- ---------------------------------------------------------------------
```

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

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

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

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

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

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

suma n = sum [1..n]

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Los triángulo aritmético se forman como sigue
--     1
```

```
--      2  3
--      4  5  6
--      7  8  9 10
--     11 12 13 14 15
--     16 16 18 19 20 21
-- Definir la función linea tal que (linea n) es la línea n-ésima de los
-- triángulos aritméticos. Por ejemplo,
--    linea 4  ==  [7,8,9,10]
--    linea 5  ==  [11,12,13,14,15]
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. 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 5.1. Definir la función nombres 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 5.2. Definir la función musicos 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 = [x | (x,m,_,_) <- bd, m == "Musica"]

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir la función seleccion 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"]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Definir, usando el apartado anterior, la función
```

```
-- musicos’ 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 5.5. Definir la función vivas 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]

-- ---------------------------------------------------------------------
-- Ejercicio 6. 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]        ==  []
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.  La distancia de Hamming entre dos listas es el número
-- de posiciones en que los correspondientes elementos son
-- distintos. Por ejemplo, la distancia de Hamming entre "roma" y "loba"
-- es 2 (porque hay 2 posiciones en las que los elementos
-- correspondientes son distintos: la 1ª y la 3ª).
--
-- Definir la función
```

```
--    distancia :: Eq a => [a] -> [a] -> Int
-- tal que (distancia xs ys) es la distancia de Hamming entre xs e
-- ys. Por ejemplo,
--    distancia "romano" "comino"  ==  2
--    distancia "romano" "camino"  ==  3
--    distancia "roma"   "comino"  ==  2
--    distancia "roma"   "camino"  ==  3
--    distancia "romano" "ron"     ==  1
--    distancia "romano" "cama"    ==  2
--    distancia "romano" "rama"    ==  1
-- ---------------------------------------------------------------------

distancia :: Eq a => [a] -> [a] -> Int
distancia xs ys = sum [1 | (x,y) <- zip xs ys, x /= y]

-- ---------------------------------------------------------------------
-- Ejercicio 8. La suma de la serie
--    1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + ...
-- es pi^2/6. Por tanto, pi se puede aproximar mediante la raíz cuadrada
-- de 6 por la suma de la serie.
--
-- Definir la función aproximaPi tal que (aproximaPi n) es la aproximación
-- de pi obtenida mediante n términos de la serie. Por ejemplo,
--    aproximaPi 4    == sqrt(6*(1/1^2 + 1/2^2 + 1/3^2 + 1/4^2))
--                    == 2.9226129861250305
--    aproximaPi 1000 == 3.1406380562059946
-- ---------------------------------------------------------------------

aproximaPi n = sqrt(6*sum [1/x^2 | x <- [1..n]])

-- ---------------------------------------------------------------------
-- Ejercicio 9. 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.
-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir la función numeroAbundante 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
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Definir la función numerosAbundantesMenores 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 :: Int -> [Int]
numerosAbundantesMenores n = [x | x <- [1..n], numeroAbundante x]

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Definir la función todosPares 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 9.4. Definir la constante primerAbundanteImpar que calcule
-- el primer número natural abundante impar. Determinar el valor de
-- dicho número.
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. (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],
                                 a^2 + b^2 == c^2]

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Definir la constante euler9 tal que euler9 es producto
-- abc donde (a,b,c) es la única terna pitagórica tal que a+b+c=1000.
-- Calcular el valor de euler9.
-- ---------------------------------------------------------------------

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

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

Relación 5

Definiciones por comprensión (3): El cifrado César

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

-- En el tema 5, cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf
-- se estudió, como aplicación de las definiciones por comprennsión, el
-- cifrado César. El objetivo de esta relación es modificar el programa
-- de cifrado César para que pueda utilizar también letras
-- mayúsculas. Por ejemplo,
--    *Main> descifra "Ytit Ufwf Sfif"
--    "Todo Para Nada"
-- Para ello, se propone la modificación de las funciones
-- correspondientes del tema 5.

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

import Data.Char

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relación 6

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-11/temas/tema-6.pdf

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir por recursión la función
--    potencia :: Integer -> Integer -> Integer
-- tal que (potencia x n) es x elevado al número natural n. Por ejemplo,
--    potencia 2 3  ==  8
-- ---------------------------------------------------------------------

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir por recursión la función
--    replicate' :: Int -> a -> [a]
-- tal que (replicate' n x) es la lista formado por n copias del
-- elemento x. Por ejemplo,
--    replicate' 3 2  ==  [2,2,2]
-- ---------------------------------------------------------------------

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

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir por recursión la función
--    elem' :: Eq a => a -> [a] -> Bool
-- tal que (elem' x xs) se verifica si x pertenece a la lista xs. Por
-- ejemplo,
--    elem' 3 [2,3,5]  ==  True
--    elem' 4 [2,3,5]  ==  False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir por recursión la función
--    mezcla :: Ord a => [a] -> [a] -> [a]
-- tal que (mezcla xs ys) es la lista obtenida mezclando las listas
-- ordenadas xs e ys. Por ejemplo,
--    mezcla [2,5,6] [1,3,4]  ==  [1,2,3,4,5,6]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir por recursión la función
--    ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada. Por
-- ejemplo,
--    ordenada [2,3,5]  ==  True
--    ordenada [2,5,3]  ==  False
-- ---------------------------------------------------------------------

ordenada :: Ord a => [a] -> Bool
ordenada []       = True
```

```
ordenada [_]      = True
ordenada (x:y:xs) = x <= y && ordenada (y:xs)

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir por recursión la función
--    borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
-- x en la lista xs. Por ejemplo,
--    borra 1 [1,2,1]  ==  [2,1]
--    borra 3 [1,2,1]  ==  [1,2,1]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir por recursión la función
--    esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutación de
-- ys. Por ejemplo,
--    esPermutacion [1,2,1] [2,1,1]  ==  True
--    esPermutacion [1,2,1] [1,2,2]  ==  False
-- ---------------------------------------------------------------------

esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     []     = True
esPermutacion []     (y:ys) = False
esPermutacion (x:xs) ys     = elem x ys && esPermutacion xs (borra x ys)

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    mitades :: [a] -> ([a],[a])
-- tal que (mitades xs) es el par formado por las dos mitades en que se
-- divide xs tales que sus longitudes difieren como máximo en uno. Por
-- ejemplo,
--    mitades [2,3,5,7,9]  ==  ([2,3],[5,7,9])
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir por recursión la función
--    ordMezcla :: Ord a => [a] -> [a]
-- tal que (ordMezcla xs) es la lista obtenida ordenado xs por mezcla
-- (es decir, considerando que la lista vacía y las listas unitarias
-- están ordenadas y cualquier otra lista se ordena mezclando las dos
-- listas que resultan de ordenar sus dos mitades por separado). Por
-- ejemplo,
--    ordMezcla [5,2,3,1,7,2,5]  =>  [1,2,2,3,5,5,7]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir por recursión la función
--    take' :: Int -> [a] -> [a]
-- tal que (take' n xs) es la lista de los n primeros elementos de
-- xs. Por ejemplo,
--    take' 3 [4..12]  =>  [4,5,6]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir por recursión la función
--    last' :: [a] -> a
-- tal que (last xs) es el último elemento de xs. Por ejemplo,
--    last' [2,3,5]  =>  5
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 15. 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 16. (Problema 5 del proyecto Euler) El problema se encuentra
-- en http://goo.gl/L5bb y consiste en calcular el menor número
-- divisible por los números del 1 al 20. Lo resolveremos mediante los
-- distintos apartados de este ejercicio.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 16.1. Definir por recursión la función
--    menorDivisible :: Integer -> Integer -> Integer
-- tal que (menorDivisible a b) es el menor número divisible por los
-- números del a al b. Por ejemplo,
--    menorDivisible 2 5  ==  60
-- Indicación: Usar la función lcm tal que (lcm x y) es el mínimo común
-- múltiplo de x e y.
-- ---------------------------------------------------------------------

menorDivisible :: Integer -> Integer -> Integer
```

```
menorDivisible  a b
    | a == b    = a
    | otherwise = lcm a (menorDivisible (a+1) b)

-- ---------------------------------------------------------------------
-- Ejercicio 16.2. Definir la constante
--    euler5 :: Integer
-- tal que euler5 es el menor número divisible por los números del 1 al
-- 20 y calcular su valor.
-- ---------------------------------------------------------------------

euler5 :: Integer
euler5 = menorDivisible 1 20

-- El cálculo es
--    ghci> euler5
--    232792560
```

Relación 7

Definiciones por recursión y por comprensión (1)

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

-- En esta relación se presentan ejercicios con dos definiciones (una
-- por recursión y otra por comprensión) y la comprobación de la
-- equivalencia de las dos definiciones con QuickCheck. Los ejercicios
-- corresponden a los temas 5 y 6 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-6.pdf
-- En concreto, se estudian funciones para calcular
-- * la suma de los cuadrados de los n primeros números,
-- * el número de bloques de escaleras triangulares,
-- * la suma de los cuadrados de los impares entre los n primeros números,
-- * la lista de las cifras de un número,
-- * la suma de las cifras de un número,
-- * la pertenencia a las cifras de un número,
-- * el número de cifras de un número,
-- * el número correspondiente a las cifras de un número,
-- * la concatenación de dos números,
-- * la primera cifra de un número,
-- * la última cifra de un número,
-- * el número con las cifras invertidas,
-- * si un número es capicúa,
-- * el exponente de la mayor potencia de un número que divide a otro,
```

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

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

import Test.QuickCheck
import Data.List

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 13.1. Definir, por recursión, la función
--    mayorExponenteR :: Integer -> Integer -> Integer
-- tal que (mayorExponenteR a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
--    mayorExponenteR 2 8    ==  3
--    mayorExponenteR 2 9    ==  0
--    mayorExponenteR 5 100  ==  2
--    mayorExponenteR 2 60   ==  2
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 13.2. Definir, por recursión, la función
--    mayorExponenteC :: Integer -> Integer -> Integer
-- tal que (mayorExponenteC a b) es el exponente de la mayor potencia de
-- a que divide a b. Por ejemplo,
--    mayorExponenteC 2 8    ==  3
--    mayorExponenteC 5 100  ==  2
```

```
--    mayorExponenteC 5 101  ==  0
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 14.4. Definir la función
```

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 16. (Problema 16 del proyecto Euler) El problema se
-- encuentra en http://goo.gl/4uWh y consiste en calcular la suma de las
```

```
-- cifras de 2^1000. Lo resolveremos mediante los distintos apartados de
-- este ejercicio.
-- ---------------------------------------------------------------------

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

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

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

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

Relación 8

Definiciones por recursión y por comprensión (2)

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

-- En esta relación se presentan ejercicios con dos definiciones (una
-- por recursión y otra por comprensión) y la comprobación de la
-- equivalencia de las dos definiciones con QuickCheck. Los ejercicios
-- corresponden a los temas 5 y 6 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-6.pdf

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    cuadradosC :: [Integer] -> [Integer]
-- tal que (cuadradosC xs) es la lista de los cuadrados de xs. Por
-- ejemplo,
--    cuadradosC [1,2,3]  ==  [1,4,9]
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    cuadradosR :: [Integer] -> [Integer]
-- tal que (cuadradosR xs) es la lista de los cuadrados de xs. Por
-- ejemplo,
--    cuadradosR [1,2,3]  ==  [1,4,9]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por comprensión, la función
--    imparesC :: [Integer] -> [Integer]
-- tal que (imparesC xs) es la lista de los números impares de xs. Por
-- ejemplo,
--    imparesC [1,2,3]  ==  [1,3]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    imparesR :: [Integer] -> [Integer]
-- tal que (imparesR xs) es la lista de los números impares de xs. Por
-- ejemplo,
--    imparesR [1,2,3]  ==  [1,3]
-- ---------------------------------------------------------------------

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

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

```
-- Ejercicio 3.1. Definir, por comprensión, la función
--    imparesCuadradosC :: [Integer] -> [Integer]
-- tal que (imparesCuadradosC xs) es la lista de los cuadrados de los
-- números impares de xs. Por ejemplo,
--    imparesCuadradosC [1,2,3]  ==  [1,9]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por recursión, la función
--    imparesCuadradosR :: [Integer] -> [Integer]
-- tal que (imparesCuadradosR xs) es la lista de los cuadrados de los
-- números impares de xs. Por ejemplo,
--    imparesCuadradosR [1,2,3]  ==  [1,9]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir, por comprensión, la función
--    sumaCuadradosImparesC :: [Integer] -> Integer
-- tal que (sumaCuadradosImparesC xs) es la suma de los cuadrados de los
-- números impares de la lista xs. Por ejemplo,
--    sumaCuadradosImparesC [1,2,3]  ==  10
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por recursión, la función
--    sumaCuadradosImparesR :: [Integer] -> Integer
-- tal que (sumaCuadradosImparesR xs) es la suma de los cuadrados de los
-- números impares de la lista xs. Por ejemplo,
--    sumaCuadradosImparesR [1,2,3]  ==  10
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, usando funciones predefinidas, la función
--    entreL :: Integer -> Integer -> [Integer]
-- tal que (entreL m n) es la lista de los números entre m y n. Por
-- ejemplo,
--    entreL 2 5  ==  [2,3,4,5]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir, por recursión, la función
--    entreR :: Integer -> Integer -> [Integer]
-- tal que (entreR m n) es la lista de los números entre m y n. Por
-- ejemplo,
--    entreR 2 5  ==  [2,3,4,5]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir, por comprensión, la función
--    mitadPares :: [Int] -> [Int]
-- tal que (mitadPares xs) es la lista de las mitades de los elementos
-- de xs que son pares. Por ejemplo,
--    mitadPares [0,2,1,7,8,56,17,18]  ==  [0,1,4,28,9]
-- ---------------------------------------------------------------------

mitadPares :: [Int] -> [Int]
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir, por recursión, la función
--    mitadParesRec :: [Int] -> [Int]
-- tal que (mitadParesRec []) es la lista de las mitades de los elementos
-- de xs que son pares. Por ejemplo,
--    mitadParesRec [0,2,1,7,8,56,17,18]  ==  [0,1,4,28,9]
-- ---------------------------------------------------------------------

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

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

-- La propiedad es
prop_mitadPares :: [Int] -> Bool
prop_mitadPares xs =
    mitadPares xs == mitadParesRec xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir, por comprensión, la función
--    enRango :: Int -> Int -> [Int] -> [Int]
-- tal que (enRango a b xs) es la lista de los elementos de xs mayores o
-- iguales que a y menores o iguales que b. Por ejemplo,
--    enRango  5 10 [1..15]   ==  [5,6,7,8,9,10]
--    enRango 10  5 [1..15]   ==  []
--    enRango  5  5 [1..15]   ==  [5]
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir, por recursión, la función
--    enRangoRec :: Int -> Int -> [Int] -> [Int]
-- tal que (enRangoRec a b []) es la lista de los elementos de xs
-- mayores o iguales que a y menores o iguales que b. Por ejemplo,
--    enRangoRec  5 10 [1..15]  ==  [5,6,7,8,9,10]
--    enRangoRec 10 5 [1..15]   ==  []
--    enRangoRec  5 5 [1..15]   ==  [5]
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir, por comprensión, la función
--    sumaPositivos :: [Int] -> Int
-- tal que (sumaPositivos xs) es la suma de los números positivos de
-- xs. Por ejemplo,
--    sumaPositivos [0,1,-3,-2,8,-1,6]  ==  15
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Definir, por recursión, la función
--    sumaPositivosRec :: [Int] -> Int
-- tal que (sumaPositivosRec xs) es la suma de los números positivos de
-- xs. Por ejemplo,
--    sumaPositivosRec [0,1,-3,-2,8,-1,6]  ==  15
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaPositivos :: [Int] -> Bool
prop_sumaPositivos xs =
    sumaPositivos xs == sumaPositivosRec xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 9. El doble factorial de un número n se define por
--    n!! = n*(n-2)* ... * 3 * 1, si n es impar
--    n!! = n*(n-2)* ... * 4 * 2, si n es par
--    1!! = 1
--    0!! = 1
-- Por ejemplo,
--    8!! = 8*6*4*2   = 384
--    9!! = 9*7*5*3*1 = 945
```

```
-- Definir, por recursión, la función
--    dobleFactorial :: Integer -> Integer
-- tal que (dobleFactorial n) es el doble factorial de n. Por ejemplo,
--    dobleFactorial 8  ==  384
--    dobleFactorial 9  ==  945
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10. 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]        ==  []
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 11. La distancia de Hamming entre dos listas es el
-- número de posiciones en que los correspondientes elementos son
-- distintos. Por ejemplo, la distancia de Hamming entre "roma" y "loba"
-- es 2 (porque hay 2 posiciones en las que los elementos
-- correspondientes son distintos: la 1ª y la 3ª).
--
-- Definir la función
--    distancia :: Eq a => [a] -> [a] -> Int
-- tal que (distancia xs ys) es la distancia de Hamming entre xs e
-- ys. Por ejemplo,
--    distancia "romano" "comino"  ==  2
--    distancia "romano" "camino"  ==  3
--    distancia "roma"   "comino"  ==  2
--    distancia "roma"   "camino"  ==  3
--    distancia "romano" "ron"     ==  1
```

```
--    distancia "romano" "cama"    ==  2
--    distancia "romano" "rama"    ==  1
-- ---------------------------------------------------------------------

-- Por comprensión:
distancia :: Eq a => [a] -> [a] -> Int
distancia xs ys = length [(x,y) | (x,y) <- zip xs ys, x /= y]

-- Por recursión:
distancia' :: Eq a => [a] -> [a] -> Int
distancia' [] ys = 0
distancia' xs [] = 0
distancia' (x:xs) (y:ys) | x /= y    = 1 + distancia' xs ys
                         | otherwise = distancia' xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 12. La suma de la serie
--    1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + ...
-- es pi^2/6. Por tanto, pi se puede aproximar mediante la raíz cuadrada
-- de 6 por la suma de la serie.
--
-- Definir la función aproximaPi tal que (aproximaPi n) es la aproximación
-- de pi obtenida mediante n términos de la serie. Por ejemplo,
--    aproximaPi 4    == sqrt(6*(1/1^2 + 1/2^2 + 1/3^2 + 1/4^2))
--                    == 2.9226129861250305
--    aproximaPi 1000 == 3.1406380562059946
-- ---------------------------------------------------------------------

-- Por comprensión:
aproximaPi n = sqrt(6*sum [1/x^2 | x <- [1..n]])

-- Por recursión:
aproximaPi' n = sqrt(6*aproximaPi'' n)

aproximaPi'' 1 = 1
aproximaPi'' n = 1/n^2 + aproximaPi'' (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 13.1. Definir por recursión la función
--    sustituyeImpar :: [Int] -> [Int]
```

```
-- tal que (sustituyeImpar xs) es la lista obtenida sustituyendo cada
-- número impar de xs por el siguiente número par. Por ejemplo,
--    sustituyeImpar [2,5,7,4]  ==  [2,6,8,4]
-- ---------------------------------------------------------------------

sustituyeImpar :: [Int] -> [Int]
sustituyeImpar []     = []
sustituyeImpar (x:xs) | odd x     = (x+1): sustituyeImpar xs
                      | otherwise = x:sustituyeImpar xs

-- ---------------------------------------------------------------------
-- Ejercicio 13.2. Comprobar con QuickChek la siguiente propiedad: para
-- cualquier lista de números enteros xs, todos los elementos de la
-- lista (sustituyeImpar xs) son números pares.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sustituyeImpar :: [Int] -> Bool
prop_sustituyeImpar xs = and [even x | x <- sustituyeImpar xs]

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

-- ---------------------------------------------------------------------
-- Ejercicio 14.1. El número e se puede definir como la suma de la
-- serie:
--    1/0! + 1/1! + 1/2! + 1/3! +...
-- Definir la función aproxE tal que (aproxE n) es la aproximación de e
-- que se obtiene sumando los términos de la serie hasta 1/n!. Por
-- ejemplo,
--    aproxE 10  ==  2.718281801146385
--    aproxE 100  ==  2.7182818284590455
-- ---------------------------------------------------------------------

aproxE n = 1 + sum [ 1 / factorial k | k <- [1..n]]

factorial n = product [1..n]

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

```
-- Ejercicio 14.2. Definir la constante e como 2.71828459.
-- ---------------------------------------------------------------------

e = 2.71828459

-- ---------------------------------------------------------------------
-- Ejercicio 14.3. Definir la función errorE tal que (errorE x) es el
-- menor número de términos de la serie anterior necesarios para obtener
-- e con un error menor que x.
--    errorE 0.1     ==  3.0
--    errorE 0.01    ==  4.0
--    errorE 0.001   ==  6.0
--    errorE 0.0001  ==  7.0
-- ---------------------------------------------------------------------

errorE x = head [n | n <- [0..], abs(aproxE n - e) < x]
```

Relación 9

Definiciones sobre cadenas, orden superior y plegado

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

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

```
--   propiedad.
--
-- La 4ª parte contiene ejercicios sobre definiciones mediante
-- map, filter y plegado. En concreto, en la 4ª parte, se estudian
-- funciones para calcular
-- * la lista de los valores de los elementos que cumplen una propiedad,
-- * la concatenación de una lista de listas,
-- * la redefinición de la función map y
-- * la redefinición de la función filter.
--
-- Estos ejercicios corresponden a los temas 5, 6 y 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-6.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-7.pdf

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

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

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

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

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

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

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

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

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

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

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

```
--    divideMedia [1,2,3]         ==  ([1.0],[3.0])
-- Definir la función divideMedia por filtrado, comprensión y
-- recursión.
-- ---------------------------------------------------------------------

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

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

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

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

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

```
-- La propiedad es
prop_divideMedia :: [Double] -> Bool
prop_divideMedia xs =
    divideMediaC xs == d &&
    divideMediaR xs == d
    where d = divideMediaF xs

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

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

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

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

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

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

-- La comprobación es
```

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

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

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

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

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

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

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

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

```
--    sumaDigitosRec :: String -> Int
-- tal que (sumaDigitosRec xs) es la suma de los dígitos de la cadena
-- xs. Por ejemplo,
--    sumaDigitosRec "SE 2431 X"  ==  10
-- Nota: Usar las funciones isDigit y digitToInt.
-- ---------------------------------------------------------------------

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

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

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

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

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por recursión, la función
--    mayusculaInicialRec :: String -> String
-- tal que (mayusculaInicialRec xs) es la palabra xs con la letra
-- inicial en mayúscula y las restantes en minúsculas. Por ejemplo,
--    mayusculaInicialRec "sEviLLa"  ==  "Sevilla"
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
-- si ys es un prefijo de xs.
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

```
--    takeWhile (<7) [2,3,9,4,5]  ==  [2,3]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relación 10

Definiciones por plegado

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

-- Esta relación contiene ejercicios con definiciones mediante
-- plegado. En concreto, se estudian definiciones por plegado para
-- calcular
-- * el máximo elemento de una lista,
-- * el mínimo elemento de una lista,
-- * la inversa de una lista,
-- * el número correspondiente a la lista de sus cifras,
-- * la suma de las sumas de las listas de una lista de listas,
-- * la lista obtenida borrando las ocurrencias de un elemento y
-- * la diferencia de dos listas.
--
-- Los ejercicios de esta relación corresponden al tema 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-7.pdf

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

import Data.Char
import Test.QuickCheck

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

```haskell
-- Ejercicio 1.1. 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
-- 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))

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. 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
-- Nota: La función maximumP es equivalente a la predefinida maximum.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, mediante plegado con foldr1, la función
--    minimunP :: Ord a => [a] -> a
-- tal que (minimunR xs) es el máximo de la lista xs. Por ejemplo,
--    minimunP [3,7,2,5]  ==  2
-- Nota: La función minimunP es equivalente a la predefinida minimun.
-- ---------------------------------------------------------------------

minimumP :: Ord a => [a] -> a
minimumP = foldr1 min

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir, mediante recursión, la función
```

```
--    inversaR :: [a] -> [a]
-- tal que (inversaR xs) es la inversa de la lista xs. Por ejemplo,
--    inversaR [3,5,2,4,7]  ==  [7,4,2,5,3]
-- ---------------------------------------------------------------------

inversaR :: [a] -> [a]
inversaR []     = []
inversaR (x:xs) = (inversaR xs) ++ [x]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, mediante plegado, la función
--    inversaP :: [a] -> [a]
-- tal que (inversaP xs) es la inversa de la lista xs. Por ejemplo,
--    inversaP [3,5,2,4,7]  ==  [7,4,2,5,3]
-- ---------------------------------------------------------------------

inversaP :: [a] -> [a]
inversaP = foldr f []
    where f x y = y ++ [x]

-- La definición anterior puede simplificarse a
inversaP_2 :: [a] -> [a]
inversaP_2 = foldr f []
    where f x = (++ [x])

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, por recursión con acumulador, la función
--    inversaR' :: [a] -> [a]
-- tal que (inversaR' xs) es la inversa de la lista xs. Por ejemplo,
--    inversaR' [3,5,2,4,7]  ==  [7,4,2,5,3]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. La función de plegado foldl está definida por
--    foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
--    foldl f ys xs = aux ys xs
--        where aux ys []     = ys
--              aux ys (x:xs) = aux (f ys x) xs
-- Definir, mediante plegado con foldl, la función
--    inversaP' :: [a] -> [a]
-- tal que (inversaP' xs) es la inversa de la lista xs. Por ejemplo,
--    inversaP' [3,5,2,4,7]  ==  [7,4,2,5,3]
-- ------------------------------------------------------------------------

inversaP' :: [a] -> [a]
inversaP' = foldl f []
    where f ys x = x:ys

-- La definición anterior puede simplificarse lambda:
inversaP'_2 :: [a] -> [a]
inversaP'_2= foldl (\ys x -> x:ys) []

-- La definición puede simplificarse usando flip:
inversaP'_3 :: [a] -> [a]
inversaP'_3 = foldl (flip(:)) []

-- ------------------------------------------------------------------------
-- Ejercicio 3.5. Comprobar con QuickCheck que las funciones reverse,
-- inversaP e inversaP' son equivalentes.
-- ------------------------------------------------------------------------

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

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

-- ------------------------------------------------------------------------
-- Ejercicio 3.6. Comparar la eficiencia de inversaP e inversaP'
-- calculando el tiempo y el espacio que usado en evaluar las siguientes
```

```
-- expresiones:
--    head (inversaP [1..100000])
--    head (inversaP' [1..100000])
-- ---------------------------------------------------------------------

-- La sesión es
--    ghci> :set +s
--    ghci> head (inversaP [1..100000])
--    100000
--    (0.41 secs, 20882460 bytes)
--    ghci> head (inversaP' [1..100000])
--    1
--    (0.00 secs, 525148 bytes)
--    ghci> :unset +s

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir, por recursión con acumulador, la función
--    dec2entR :: [Int] -> Int
-- tal que (dec2entR xs) es el entero correspondiente a la expresión
-- decimal xs. Por ejemplo,
--    dec2entR [2,3,4,5]  ==  2345
-- ---------------------------------------------------------------------

dec2entR :: [Int] -> Int
dec2entR xs = dec2entR' 0 xs
    where dec2entR' a []     = a
          dec2entR' a (x:xs) = dec2entR' (10*a+x) xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por plegado con foldl, la función
--    dec2entP :: [Int] -> Int
-- tal que (dec2entP xs) es el entero correspondiente a la expresión
-- decimal xs. Por ejemplo,
--    dec2entP [2,3,4,5]  ==  2345
-- ---------------------------------------------------------------------

dec2entP :: [Int] -> Int
dec2entP = foldl f 0
    where f a x = 10*a+x
```

```
-- La definición puede simplificarse usando lambda:
dec2entP' :: [Int] -> Int
dec2entP' = foldl (\a x -> 10*a+x) 0

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, mediante recursión, la función
--    sumllR :: Num a => [[a]] -> a
-- tal que (sumllR xss) es la suma de las sumas de las listas de xss.
-- Por ejemplo,
--    sumllR [[1,3],[2,5]]  ==  11
-- ---------------------------------------------------------------------

sumllR :: Num a => [[a]] -> a
sumllR [] = 0
sumllR (xs:xss) = sum xs + sumllR xss

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir, mediante plegado, la función
--    sumllP :: Num a => [[a]] -> a
-- tal que (sumllP xss) es la suma de las sumas de las listas de xss. Por
-- ejemplo,
--    sumllP [[1,3],[2,5]]  ==  11
-- ---------------------------------------------------------------------

sumllP :: Num a => [[a]] -> a
sumllP = foldr f 0
    where f xs n = sum xs + n

-- La definición anterior puede simplificarse usando lambda
sumllP' :: Num a => [[a]] -> a
sumllP' = foldr (\xs n -> sum xs + n) 0

sumllT :: Num a => [[a]] -> a
sumllT xs = aux 0 xs
       where aux a [] = a
             aux a (xs:xss) = aux (a + sum xs) xss

sumllTP :: Num a => [[a]] -> a
sumllTP = foldl (\a xs -> a + sum xs) 0
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir, mediante recursión, la función
--    borraR :: Eq a => a -> a -> [a]
-- tal que (borraR y xs) es la lista obtenida borrando las ocurrencias de
-- y en xs. Por ejemplo,
--    borraR 5 [2,3,5,6]    ==  [2,3,6]
--    borraR 5 [2,3,5,6,5]  ==  [2,3,6]
--    borraR 7 [2,3,5,6,5]  ==  [2,3,5,6,5]
-- ---------------------------------------------------------------------

borraR :: Eq a => a -> [a] -> [a]
borraR z [] = []
borraR z (x:xs) | z == x    = borraR z xs
                | otherwise = x : borraR z xs

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir, mediante plegado, la función
--    borraP :: Eq a => a -> a -> [a]
-- tal que (borraP y xs) es la lista obtenida borrando las ocurrencias de
-- y en xs. Por ejemplo,
--    borraP 5 [2,3,5,6]    ==  [2,3,6]
--    borraP 5 [2,3,5,6,5]  ==  [2,3,6]
--    borraP 7 [2,3,5,6,5]  ==  [2,3,5,6,5]
-- ---------------------------------------------------------------------

borraP :: Eq a => a -> [a] -> [a]
borraP z = foldr f []
    where f x y | z == x    = y
                | otherwise = x:y

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir, mediante recursión, la función
--    diferenciaR :: Eq a => [a] -> [a] -> [a]
-- tal que (diferenciaR xs ys) es la diferencia del conjunto xs e ys; es
-- decir el conjunto de los elementos de xs que no pertenecen a ys. Por
-- ejemplo,
```

```
--    diferenciaR [2,3,5,6] [5,2,7]  ==  [3,6]
-- ---------------------------------------------------------------------

diferenciaR :: Eq a => [a] -> [a] -> [a]
diferenciaR xs ys = aux xs xs ys
    where aux a xs []     = a
          aux a xs (y:ys) = aux (borraR y a) xs ys

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

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir, mediante plegado con foldl, la función
--    diferenciaP :: Eq a => [a] -> [a] -> [a]
-- tal que (diferenciaP xs ys) es la diferencia del conjunto xs e ys; es
-- decir el conjunto de los elementos de xs que no pertenecen a ys. Por
-- ejemplo,
--    diferenciaP [2,3,5,6] [5,2,7]  ==  [3,6]
-- ---------------------------------------------------------------------

diferenciaP :: Eq a => [a] -> [a] -> [a]
diferenciaP xs ys = foldl (flip borraR) xs ys

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

Relación 11

Codificación y transmisión de mensajes

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

-- En esta relación se va a modificar el programa de transmisión de
-- cadenas para detectar errores de transmisión sencillos usando bits de
-- paridad. Es decir, cada octeto de ceros y unos generado durante la
-- codificación se extiende con un bit de paridad que será un uno si el
-- número contiene un número impar de unos y cero en caso contrario. En
-- la decodificación, en cada número binario de 9 cifras debe
-- comprobarse que la paridad es correcta, en cuyo caso se descarta el
-- bit de paridad. En caso contrario, debe generarse un mensaje de error
-- en la paridad.
--
-- Los ejercicios de esta relación corresponden al tema 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-7.pdf

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

import Data.Char

-- ---------------------------------------------------------------------
-- Notas. Se usarán las siguientes definiciones del tema
-- ---------------------------------------------------------------------
```

```
type Bit = Int

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    paridad :: [Bit] -> Bit
-- tal que (paridad bs) es el bit de paridad de bs; es decir, 1 si bs
-- contiene un número impar de unos y 0 en caso contrario. Por ejemplo,
--    paridad [0,1,1]      =>  0
--    paridad [0,1,1,0,1]  =>  1
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    agregaParidad :: [Bit] -> [Bit]
-- tal que (agregaParidad bs) es la lista obtenida añadiendo al
-- principio de bs su paridad. Por ejemplo,
--    agregaParidad [0,1,1]      =>  [0,0,1,1]
--    agregaParidad [0,1,1,0,1]  =>  [1,0,1,1,0,1]
-- ---------------------------------------------------------------------

agregaParidad :: [Bit] -> [Bit]
```

```
agregaParidad bs = (paridad bs) : bs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    codifica :: String -> [Bit]
-- tal que (codifica c) es la codificación de la cadena c como una lista
-- de bits obtenida convirtiendo cada carácter en un número Unicode,
-- convirtiendo cada uno de dichos números en un octeto con su paridad y
-- concatenando los octetos con paridad para obtener una lista de
-- bits. Por ejemplo,
--    *Main> codifica "abc"
--    [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    separa9 :: [Bit] -> [[Bit]]
-- tal que (separa9 bs)} es la lista obtenida separando la lista de bits
-- bs en listas de 9 elementos. Por ejemplo,
--    *Main> separa9 [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
--    [[1,1,0,0,0,0,1,1,0],[1,0,1,0,0,0,1,1,0],[0,1,1,0,0,0,1,1,0]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    compruebaParidad :: [Bit] -> [Bit ]
-- tal que (compruebaParidad bs) es el resto de bs si el primer elemento
-- de bs es el bit de paridad del resto de bs y devuelve error de
-- paridad en caso contrario. Por ejemplo,
--    *Main> compruebaParidad [1,1,0,0,0,0,1,1,0]
--    [1,0,0,0,0,1,1,0]
--    *Main> compruebaParidad [0,1,0,0,0,0,1,1,0]
--    *** Exception: paridad erronea
```

```
-- Usar la función del preludio
--    error :: String -> a
-- tal que (error c) devuelve la cadena c.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    descodifica :: [Bit] -> String
-- tal que (descodifica bs) es la cadena correspondiente a la lista de
-- bits con paridad. Para ello, en cada número binario de 9 cifras debe
-- comprobarse que la paridad es correcta, en cuyo caso se descarta el
-- bit de paridad. En caso contrario, debe generarse un mensaje de error
-- en la paridad. Por ejemplo,
--    descodifica [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
--    =>  "abc"
--    descodifica [1,0,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
--    =>  "*** Exception: paridad erronea
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. Se define la función
--    transmite :: ([Bit] -> [Bit]) -> String -> String
--    transmite canal =  descodifica . canal . codifica
-- tal que (transmite c t) es la cadena obtenida transmitiendo la cadena
-- t a través del canal c. Calcular el reultado de trasmitir la cadena
-- "Conocete a ti mismo" por el canal identidad (id) y del canal que
-- olvida el primer bit (tail).
-- ---------------------------------------------------------------------

transmite :: ([Bit] -> [Bit]) -> String -> String
transmite canal =  descodifica . canal . codifica
```

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

Relación 12

Resolución de problemas matemáticos

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

-- En esta relación se plantea la resolución de distintos problemas
-- matemáticos. En concreto,
-- * el problema de Ullman sobre la existencia de subconjunto del tamaño
--   dado y con su suma acotada,
-- * las descomposiciones de un número como suma de dos cuadrados,
-- * el problema 145 del proyecto Euler,
-- * el grafo de una función sobre los elementos que cumplen una
--   propiedad,
-- * los números semiperfectos,
-- * el producto, por plegado, de los números que verifican una propiedad,
-- * el carácter funcional de una relación,
-- * las cabezas y las colas de una lista y
-- * la identidad de Bezout.
--
-- Además, de los 2 primeros se presentan distintas definiciones y se
-- compara su eficiencia.
--
-- Estos ejercicios corresponden a los temas 5, 6 y 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-6.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-7.pdf
```

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

import Test.QuickCheck
import Data.List

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, usando funciones de orden superior, la función
--    grafoReducido :: Eq a => (a -> b) -> (a -> Bool) -> [a] -> [(a,b)]
-- tal que (grafoReducido f p xs) es la lista (sin repeticiones) de los
-- pares formados por los elementos de xs que verifican el predicado p
-- y sus imágenes. Por ejemplo,
--    grafoReducido (^2) even [1..9]  ==  [(2,4),(4,16),(6,36),(8,64)]
--    grafoReducido (+4) even (replicate 40 1) == []
--    grafoReducido (*5) even (replicate 40 2) == [(2,10)]
-- ---------------------------------------------------------------------

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

-- --------------------------------------------------------------------
-- Ejercicio 6.1. Un número natural n se denomina semiperfecto si es la
-- suma de algunos de sus divisores propios. Por ejemplo, 18 es
-- semiperfecto ya que sus divisores son 1, 2, 3, 6, 9 y se cumple que
-- 3+6+9=18.
--
-- Definir la función
--    esSemiPerfecto :: Int -> Bool
-- tal que (esSemiPerfecto n) se verifica si n es semiperfecto. Por
-- ejemplo,
```

```
--    esSemiPerfecto 18 == True
--    esSemiPerfecto 9  == False
--    esSemiPerfecto 24 == True
-- ---------------------------------------------------------------------

esSemiPerfecto :: Int -> Bool
esSemiPerfecto n =
    or [sum ys == n | ys <- subconjuntos (divisores n)]

-- (divisores n) es la lista de los divisores propios de n. Por ejemplo,
--    divisores 18 == [1,2,3,6,9]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n-1], mod n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir la constante primerSemiPerfecto tal que su
-- valor es el primer número semiperfecto.
-- ---------------------------------------------------------------------

primerSemiPerfecto :: Int
primerSemiPerfecto = head [n | n <- [1..], esSemiPerfecto n]

-- La evaluación es
--    *Main> primerSemiPerfecto
--    6

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir la función
--    semiPerfecto :: Int -> Int
-- tal que (semiPerfecto n) es el n-ésimo número semiperfecto. Por
-- ejemplo,
--    semiPerfecto 1   == 6
--    semiPerfecto 4   == 20
--    semiPerfecto 100 == 414
-- ---------------------------------------------------------------------

semiPerfecto :: Int -> Int
semiPerfecto n = semiPerfectos !! n

-- semiPerfectos es la lista de los números semiPerfectos. Por ejemplo,
```

```
--    take 4 semiPerfectos  ==  [6,12,18,20]
semiPerfectos = [n | n <- [1..], esSemiPerfecto n]

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir mediante plegado la función
--    producto :: Num a => [a] -> a
-- tal que (producto xs) es el producto de los elementos de la lista
-- xs. Por ejemplo,
--    producto [2,1,-3,4,5,-6] == 720
-- ---------------------------------------------------------------------

producto :: Num a => [a] -> a
producto = foldr (*) 1

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir mediante plegado la función
--    productoPred :: Num a => (a -> Bool) -> [a] -> a
-- tal que (productoPred p xs) es el producto de los elementos de la
-- lista xs que verifican el predicado p. Por ejemplo,
--    productoPred even [2,1,-3,4,-5,6] == 48
-- ---------------------------------------------------------------------

productoPred :: Num a => (a -> Bool) -> [a] -> a
productoPred p = foldr (\x y -> if p x then x*y else y) 1

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Definir la función
--    productoPos :: (Num a, Ord a) => [a] -> a
-- tal que (productoPos xs) esel producto de los elementos estríctamente
-- positivos de la lista xs. Por ejemplo,
--    productoPos [2,1,-3,4,-5,6] == 48
-- ---------------------------------------------------------------------

productoPos :: (Num a, Ord a) => [a] -> a
productoPos = productoPred (>0)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Las relaciones finitas se pueden representar mediante
-- listas de pares. Por ejemplo,
--    r1, r2, r3 :: [(Int, Int)]
```

```
--    r1 = [(1,3), (2,6), (8,9), (2,7)]
--    r2 = [(1,3), (2,6), (8,9), (3,7)]
--    r3 = [(1,3), (2,6), (8,9), (3,6)]
-- Definir la función
--    esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
-- tal que (esFuncion r) se verifica si la relación r es una función (es
-- decir, a cada elemento del dominio de la relación r le corresponde un
-- único elemento). Por ejemplo,
--    esFuncion r1 == False
--    esFuncion r2 == True
--    esFuncion r3 == True
-- ---------------------------------------------------------------------

r1, r2, r3 :: [(Int, Int)]
r1 = [(1,3), (2,6), (8,9), (2,7)]
r2 = [(1,3), (2,6), (8,9), (3,7)]
r3 = [(1,3), (2,6), (8,9), (3,6)]

esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
esFuncion [] = True
esFuncion ((x,y):r) =
    [y' | (x',y') <- r, x == x', y /= y'] == [] && esFuncion r

-- --------------------------------------------------------------------
-- Ejercicio 9.1. Se denomina cola de una lista l a una sublista no
-- vacía de l formada por un elemento y los siguientes hasta el
-- final. Por ejemplo, [3,4,5] es una cola de la lista [1,2,3,4,5].
--
-- Definir la función
--    colas :: [a] -> [[a]]
-- tal que (colas xs) es la lista de las colas
-- de la lista xs. Por ejemplo,
--    colas []        == [[]]
--    colas [1,2]     == [[1,2],[2],[]]
--    colas [4,1,2,5] == [[4,1,2,5],[1,2,5],[2,5],[5],[]]
-- ---------------------------------------------------------------------

colas :: [a] -> [[a]]
colas []     = [[]]
colas (x:xs) = (x:xs) : colas xs
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Comprobar con QuickCheck que las funciones colas y
-- tails son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_colas :: [Int] -> Bool
prop_colas xs = colas xs == tails xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Se denomina cabeza de una lista l a una sublista no
-- vacía de la formada por el primer elemento y los siguientes hasta uno
-- dado. Por ejemplo, [1,2,3] es una cabeza de [1,2,3,4,5].
--
-- Definir la función
--    cabezas :: [a] -> [[a]]
-- tal que (cabezas xs) es la lista de las cabezas de la lista xs. Por
-- ejemplo,
--    cabezas []          == [[]]
--    cabezas [1,4]       == [[],[1],[1,4]]
--    cabezas [1,4,5,2,3] == [[],[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-- ---------------------------------------------------------------------

-- 1. Por recursión
cabezas :: [a] -> [[a]]
cabezas []     = [[]]
cabezas (x:xs) = [] : [x:ys | ys <- cabezas xs]

-- 2. Usando patrones de plegado.
cabezasP :: [a] -> [[a]]
cabezasP = foldr (\x y -> [x]:[x:ys | ys <- y]) []

-- 3. Usando colas y funciones de orden superior.
cabezas3 :: [a] -> [[a]]
cabezas3 xs = reverse (map reverse (colas (reverse xs)))
```

```
-- La anterior definición puede escribirse sin argumentos como
cabezas3' :: [a] -> [[a]]
cabezas3' = reverse . map reverse . (colas . reverse)

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Comprobar con QuickCheck que las funciones cabezas y
-- inits son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_cabezas :: [Int] -> Bool
prop_cabezas xs = cabezas xs == inits xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. [La identidad de Bezout] Definir la función
--    bezout :: Integer -> Integer -> (Integer, Integer)
-- tal que (bezout a b) es un par de números x e y tal que a*x+b*y es el
-- máximo común divisor de a y b. Por ejemplo,
--    bezout 21 15  ==  (-2,3)
-- Indicación: Se puede usar la función quotRem tal que (quotRem x y) es
-- el par formado por el cociente y el resto de dividir x entre y.
-- ---------------------------------------------------------------------

-- Ejemplo de cálculo
--    a  b   q r
--    36 21  1 15   (1)
--    21 15  1  6   (2)
--    15  6  2  3   (3)
--     6  3  2  0
--     3  0
-- Por tanto,
--    3 = 15 - 6*2              [por (3)]
--      = 15 - (21-15*1)*2      [por (2)]
--      = 21*(-2) + 15*3
--      = 21*(-2)+ (36-21*1)*3  [por (1)]
```

```
--      = 36*3 + 21*(-5)

-- Sean q, r el cociente y el resto de a entre b, d el máximo común
-- denominador de a y b y (x,y) el valor de (bezout b r) . Entonces,
--    a = bp+r
--    d = bx+ry
-- Por tanto,
--    d = bx + (a-bp)y
--      = ay + b(x-qy)
-- Luego,
--    bezout a b = (y,x-qy)

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. Comprobar con QuickCheck que si a>0, b>0 y
-- (x,y) es el valor de (bezout a b), entonces a*x+b*y es igual al
-- máximo común divisor de a y b.
-- ---------------------------------------------------------------------

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

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

Relación 13

Demostración de propiedades por inducción

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

-- En esta relación se plantean ejercicios de demostración por inducción
-- de propiedades de programas. En concreto,
--   * la suma de los n primeros impares es n^2,
--   * 1 + 2^0 + 2^1 + 2^2 + ... + 2^n = 2^(n+1),
--   * todos los elementos de (copia n x) son iguales a x,
-- Además, se plantea la definición de la traspuesta de una matriz.
--
-- Estos ejercicios corresponden al tema 8 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-8.pdf

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir por recursión la función
--    sumaImpares :: Int -> Int
-- tal que (sumaImpares n) es la suma de los n primeros números
```

```
-- impares. Por ejemplo,
--    sumaImpares 5  ==  25
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, sin usar recursión, la función
--    sumaImpares' :: Int -> Int
-- tal que (sumaImpares' n) es la suma de los n primeros números
-- impares. Por ejemplo,
--    *Main> sumaImpares' 5  ==  25
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    sumaImparesIguales :: Int -> Int -> Bool
-- tal que (sumaImparesIguales m n) se verifica si para todo x entre m y
-- n se tiene que (sumaImpares x) y (sumaImpares' x) son iguales.
--
-- Comprobar que (sumaImpares x) y (sumaImpares' x) son iguales para
-- todos los números x entre 1 y 100.
-- ---------------------------------------------------------------------

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

-- La comprobación es
--    *Main>  sumaImparesIguales 1 100
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
```

```
--    grafoSumaImpares :: Int -> Int -> [(Int,Int)]
-- tal que (grafoSumaImpares m n) es la lista formadas por los números x
-- entre m y n y los valores de (sumaImpares x).
--
-- Calcular (grafoSumaImpares 1 9).
-- ---------------------------------------------------------------------

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

-- El cálculo es
--    *Main> grafoSumaImpares 1 9
--    [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64),(9,81)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Demostrar por inducción que para todo n,
-- (sumaImpares n) es igual a n^2.
-- ---------------------------------------------------------------------

{-
 Caso base: Hay que demostrar que
    sumaImpares 0 = 0^2
 En efecto,
    sumaImpares 0   [por hipótesis]
    = 0               [por sumaImpares.1]
    = 0^2             [por aritmética]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
     sumaImpares n = n^2
  Hay que demostrar que
     sumaImpares (n+1) = (n+1)^2
  En efecto,
     sumaImpares (n+1) =
     = (sumaImpares n) + (2*n+1)        [por sumaImpares.2]
     = n^2 + (2*n+1)                    [por H.I.]
     = (n+1)^2                          [por álgebra]
-}
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir por recursión la función
--    sumaPotenciasDeDosMasUno :: Int -> Int
-- tal que
--    sumaPotenciasDeDosMasUno n = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
-- Por ejemplo,
--    sumaPotenciasDeDosMasUno 3 == 16
-- ---------------------------------------------------------------------

sumaPotenciasDeDosMasUno :: Int -> Int
sumaPotenciasDeDosMasUno 0     = 2
sumaPotenciasDeDosMasUno (n+1) = sumaPotenciasDeDosMasUno n + 2^(n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir por comprensión la función
--    sumaPotenciasDeDosMasUno' :: Int -> Int
-- tal que
--    (sumaPotenciasDeDosMasUno' n) = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
-- Por ejemplo,
--    sumaPotenciasDeDosMasUno' 3  ==  16
-- ---------------------------------------------------------------------

sumaPotenciasDeDosMasUno' :: Int -> Int
sumaPotenciasDeDosMasUno' n = 1 + sum [2^x | x <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Demostrar por inducción que
--    sumaPotenciasDeDosMasUno n = 2^(n+1)
-- ---------------------------------------------------------------------

{-
  Caso base: Hay que demostrar que
     sumaPotenciasDeDosMasUno 0 = 2^(0+1)
  En efecto,
       sumaPotenciasDeDosMasUno 0
     = 2                             [por sumaPotenciasDeDosMasUno.1]
     = 2^(0+1)                       [por aritmética]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
     sumaPotenciasDeDosMasUno n = 2^(n+1)
```

```
  Hay que demostrar que
     sumaPotenciasDeDosMasUno (n+1) = 2^((n+1)+1)
  En efecto,
       sumaPotenciasDeDosMasUno (n+1)
     = (sumaPotenciasDeDosMasUno n) + 2^(n+1)  [por sumaPotenciasDeDosMasUno.2]
     = 2^(n+1) + 2^(n+1)                       [por H.I.]
     = 2^((n+1)+1)                             [por aritmética]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir por recursión la función
--    copia :: Int -> a -> [a]
-- tal que (copia n x) es la lista formado por n copias del elemento
-- x. Por ejemplo,
--    copia 3 2  ==  [2,2,2]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir por recursión la función
--    todos :: (a -> Bool) -> [a] -> Bool
-- tal que (todos p xs) se verifica si todos los elementos de xs cumplen
-- la propiedad p. Por ejemplo,
--    todos even [2,6,4]  ==  True
--    todos even [2,5,4]  ==  False
-- ---------------------------------------------------------------------

todos :: (a -> Bool) -> [a] -> Bool
todos p []       = True                 -- todos.1
todos p (x : xs) = p x && todos p xs    -- todos.2

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck que todos los elementos de
-- (copia n x) son iguales a x.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```
prop_copia :: Eq a => Int -> a -> Bool
prop_copia n x =
    todos (==x) (copia n' x)
    where n' = abs n

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Demostrar, por inducción en n, que todos los elementos
-- de (copia n x) son iguales a x.
-- ---------------------------------------------------------------------

{-
  Hay que demostrar que para todo n y todo x,
     todos (==x) (copia n x)

  Caso base: Hay que demostrar que
     todos (==x) (copia 0 x) = True
  En efecto,
       todos (== x) (copia 0 x)
     = todos (== x) []            [por copia.1]
     = True                       [por todos.1]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
     todos (==x) (copia n x) = True
  Hay que demostrar que
     todos (==x) (copia (n+1) x) = True
  En efecto,
       todos (==x) (copia (n+1) x)
     = todos (==x) (x : copia n x )         [por copia.2]
     = x == x && todos (==x) (copia n x )   [por todos.2]
     = True && todos (==x) (copia n x )     [por def. de ==]
     = todos (==x) (copia n x )             [por def. de &&]
     = True                                 [por H.I.]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir por plegado la función
```

```
--    todos' :: (a -> Bool) -> [a] -> Bool
-- tal que (todos' p xs) se verifica si todos los elementos de xs cumplen
-- la propiedad p. Por ejemplo,
--    todos' even [2,6,4]  ==>  True
--    todos' even [2,5,4]  ==>  False
-- ---------------------------------------------------------------------

-- 1ª definición:
todos'_1 :: (a -> Bool) -> [a] -> Bool
todos'_1 p = foldr f True
             where f x y = p x && y

-- 2ª definición:
todos'_2 :: (a -> Bool) -> [a] -> Bool
todos'_2 p = foldr f True
             where f x y = (((&&) . p) x) y

-- 3ª definición:
todos' :: (a -> Bool) -> [a] -> Bool
todos' p = foldr ((&&) . p) True

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    traspuesta :: [[a]] -> [[a]]
-- tal que (traspuesta m) es la traspuesta de la matriz m. Por ejemplo,
--    traspuesta [[1,2,3],[4,5,6]]    ==  [[1,4],[2,5],[3,6]]
--    traspuesta [[1,4],[2,5],[3,6]]  ==  [[1,2,3],[4,5,6]]
-- ---------------------------------------------------------------------

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

Relación 14

El 2011 y los números primos

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

-- Cada comienzo de año se suelen buscar propiedades numéricas del
-- número del año. En el 2011 se han buscado propiedades que relacionan
-- el 2011 y los números primos. En este ejercicio vamos a realizar la
-- búsqueda de dichas propiedades con Haskell.

import Data.List (sort)

-- La criba de Erastótenes es un método para calcular números primos. Se
-- comienza escribiendo todos los números desde 2 hasta (supongamos)
-- 100. El primer número (el 2) es primo. Ahora eliminamos todos los
-- múltiplos de 2. El primero de los números restantes (el 3) también es
-- primo. Ahora eliminamos todos los múltiplos de 3. El primero de los
-- números restantes (el 5) también es primo ... y así
-- sucesivamente. Cuando no quedan números, se han encontrado todos los
-- números primos en el rango fijado.

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    elimina :: Int -> [Int] -> [Int]
-- tal que (elimina n xs) es la lista obtenida eliminando en la lista xs
-- los múltiplos de n. Por ejemplo,
--    elimina 3 [2,3,8,9,5,6,7]  ==  [2,8,5,7]
-- ---------------------------------------------------------------------
```

```
-- Por comprensión:
elimina :: Int -> [Int] -> [Int]
elimina n xs = [ x | x <- xs, x `mod` n /= 0 ]

-- Por recursión:
eliminaR :: Int -> [Int] -> [Int]
eliminaR n [] = []
eliminaR n (x:xs) | mod x n == 0 = eliminaR n xs
                  | otherwise    = x : eliminaR n xs

-- Por plegado:
eliminaP :: Int -> [Int] -> [Int]
eliminaP n = foldr f []
             where f x y | mod x n == 0 = y
                         | otherwise    = x:y

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    criba :: [Int] -> [Int]
-- tal que (criba xs) es la lista obtenida cribando la lista xs con el
-- método descrito anteriormente. Por ejemplo,
--    criba [2..20]          ==  [2,3,5,7,11,13,17,19]
--    take 10 (criba [2..])  ==  [2,3,5,7,11,13,17,19,23,29]
-- ---------------------------------------------------------------------

criba :: [Int] -> [Int]
criba []     = []
criba (n:ns) = n : criba (elimina n ns)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    primos :: [Int]
-- cuyo valor es la lista de los números primos. Por ejemplo,
--    take 10 primos  ==  [2,3,5,7,11,13,17,19,23,29]
-- ---------------------------------------------------------------------

primos :: [Int]
primos = criba [2..]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    esPrimo :: Int -> Bool
-- tal que (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 9  ==  False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Comprobar que 2011 es primo.
-- ---------------------------------------------------------------------

-- La comprobación es
--     ghci> esPrimo 2011
--     True

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    prefijosConSuma :: [Int] -> Int -> [[Int]]
-- tal que (prefijosConSuma xs n) es la lista de los prefijos de xs cuya
-- suma es n. Por ejemplo,
--    prefijosConSuma [1..10] 3  == [[1,2]]
--    prefijosConSuma [1..10] 4  == []
-- ---------------------------------------------------------------------

prefijosConSuma :: [Int] -> Int -> [[Int]]
prefijosConSuma [] 0 = [[]]
prefijosConSuma [] n = []
prefijosConSuma (x:xs) n
    | x < n  = [x:ys | ys <- prefijosConSuma xs (n-x)]
    | x == n = [[x]]
    | x > n  = []

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    consecutivosConSuma :: [Int] -> Int -> [[Int]]
-- (consecutivosConSuma xs n) es la lista de los elementos consecutivos
```

```
-- de xs cuya suma es n. Por ejemplo,
--    consecutivosConSuma [1..10] 9  == [[2,3,4],[4,5],[9]]
-- ---------------------------------------------------------------------

consecutivosConSuma :: [Int] -> Int -> [[Int]]
consecutivosConSuma [] 0 = [[]]
consecutivosConSuma [] n = []
consecutivosConSuma (x:xs) n =
    (prefijosConSuma (x:xs) n) ++ (consecutivosConSuma xs n)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    primosConsecutivosConSuma :: Int -> [[Int]]
-- tal que (primosConsecutivosConSuma n) es la lista de los números
-- primos consecutivos cuya suma es n. Por ejemplo,
--    ghci> primosConsecutivosConSuma 41
--    [[2,3,5,7,11,13],[11,13,17],[41]]
-- ---------------------------------------------------------------------

primosConsecutivosConSuma :: Int -> [[Int]]
primosConsecutivosConSuma n =
    consecutivosConSuma (takeWhile (<=n) primos) n

-- ---------------------------------------------------------------------
-- Ejercicio 9. Calcular las descomposiciones de 2011 como sumas de
-- primos consecutivos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> primosConsecutivosConSuma 2011
--    [[157,163,167,173,179,181,191,193,197,199,211],[661,673,677],[2011]]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    propiedad1 :: Int -> Bool
-- tal que (propiedad1 n) se verifica si n sólo se puede expresar como
-- sumas de 1, 3 y 11 primos consecutivos. Por ejemplo,
--    propiedad1 2011  ==  True
--    propiedad1 2010  ==  False
-- ---------------------------------------------------------------------
```

```
propiedad1 :: Int -> Bool
propiedad1 n =
    sort (map length (primosConsecutivosConSuma n)) == [1,3,11]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Calcular los años hasta el 3000 que cumplen la
-- propiedad1.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> [n | n <- [1..3000], propiedad1 n]
--    [883,2011]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    sumaCifras :: Int -> Int
-- tal que (sumaCifras x) es la suma de las cifras del número x. Por
-- ejemplo,
--    sumaCifras 254  ==  11
-- ---------------------------------------------------------------------

sumaCifras :: Int -> Int
sumaCifras x = sum [read [y] | y <- show x]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    sumaCifrasLista :: [Int] -> Int
-- tal que (sumaCifrasLista xs) es la suma de las cifras de la lista de
-- números xs. Por ejemplo,
--    sumaCifrasLista [254, 61]  ==  18
-- ---------------------------------------------------------------------

-- Por comprensión:
sumaCifrasLista :: [Int] -> Int
sumaCifrasLista xs = sum [sumaCifras y | y <- xs]

-- Por recursión:
sumaCifrasListaR :: [Int] -> Int
sumaCifrasListaR [] = 0
```

```
sumaCifrasListaR (x:xs) = sumaCifras x + sumaCifrasListaR xs

-- Por plegado:
sumaCifrasListaP :: [Int] -> Int
sumaCifrasListaP = foldr f 0
                   where f x y = sumaCifras x + y

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    propiedad2 :: Int -> Bool
-- tal que (propiedad2 n) se verifica si n puede expresarse como suma de
-- 11 primos consecutivos y la suma de las cifras de los 11 sumandos es
-- un número primo. Por ejemplo,
--    propiedad2 2011  ==  True
--    propiedad2 2000  ==  False
-- ---------------------------------------------------------------------

propiedad2 :: Int -> Bool
propiedad2 n = [xs | xs <- primosConsecutivosConSuma n,
                     length xs == 11,
                     esPrimo (sumaCifrasLista xs)]
               /= []

-- ---------------------------------------------------------------------
-- Ejercicio 15. Calcular el primer año que cumple la propiedad1 y la
-- propiedad2.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [1..], propiedad1 n, propiedad2 n]
--    2011

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    propiedad3 :: Int -> Bool
-- tal que (propiedad3 n) se verifica si n puede expresarse como suma de
-- tantos números primos consecutivos como indican sus dos últimas
-- cifras. Por ejemplo,
--    propiedad3 2011  ==  True
--    propiedad3 2000  ==  False
```

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

propiedad3 :: Int -> Bool
propiedad3 n = [xs | xs <- primosConsecutivosConSuma n,
                     length xs == a]
               /= []
    where a = mod n 100

-- ---------------------------------------------------------------------
-- Ejercicio 17. Calcular el primer año que cumple la propiedad1 y la
-- propiedad3.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [1..], propiedad1 n, propiedad3 n]
--    2011

-- Hemos comprobado que 2011 es el menor número que cumple las
-- propiedades 1 y 2 y también es el menor número que cumple las
-- propiedades 1 y 3.
```

Relación 15

Listas infinitas

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

-- En esta relación se presentan ejercicios con listas infinitas y
-- evaluación perezosa. En concreto, se estudian funciones para calcular
-- * la lista de las potencias de un número menores que otro dado,
-- * la lista obtenida repitiendo un elemento infinitas veces,
-- * la lista obtenida repitiendo un elemento un número finito de veces,
-- * la cadena obtenida cada elemento tantas veces como indica su
--   posición,
-- * la aplicación iterada de una función a un elemento,
-- * la lista de las sublistas de longitud dada y
-- * la sucesión de Collatz.
--
-- Estos ejercicios corresponden al tema 10 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-10.pdf

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

import Test.QuickCheck

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

```
--    potenciasMenores :: Int -> Int -> [Int]
-- tal que (potenciasMenores x y) es la lista de las potencias de x
-- menores que y. Por ejemplo,
--    potenciasMenores 2 1000  ==  [2,4,8,16,32,64,128,256,512]
-- ---------------------------------------------------------------------

potenciasMenores :: Int -> Int -> [Int]
potenciasMenores x y = takeWhile (<y) (map (x^) [1..])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión y comprensión, la función
--    repite :: a -> [a]
-- tal que (repite x) es la lista infinita cuyos elementos son x. Por
-- ejemplo,
--    repite 5           == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
--    take 3 (repite 5)  ==  [5,5,5]
-- Nota: La función repite es equivalente a la función repeat definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------

-- Por recursión:
repite :: a -> [a]
repite x = x : repite x

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

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

-- Por recursión:
repiteFinita :: Int -> a -> [a]
repiteFinita 0 x = []
```

```
repiteFinita n x = x : repiteFinita (n-1) x

-- Por comprensión:
repiteFinita' :: Int -> a -> [a]
repiteFinita' n x = [x | _ <- [1..n]]

-- También se puede definir usando repite
repiteFinita2 :: Int -> a -> [a]
repiteFinita2 n x = take n (repite x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se considera la función
--    eco :: String -> String
-- tal que (eco xs) es la cadena obtenida a partir de la cadena xs
-- repitiendo cada elemento tantas veces como indica su posición: el
-- primer elemento se repite 1 vez, el segundo 2 veces y así
-- sucesivamente. Por ejemplo,
--    eco "abcd"  ==  "abbcccdddd"
-- 1. Escribir una definición 'no recursiva' de la función eco.
-- 2. Escribir una definición 'recursiva' de la función eco.
-- ---------------------------------------------------------------------

-- Una definición no recursiva es
ecoNR :: String -> String
ecoNR xs = concat [replicate i x | (i,x) <- zip [1..] xs]

-- Una definición recursiva es
ecoR :: String -> String
ecoR xs = aux 1 xs
    where aux n [] = []
          aux n (x:xs) = replicate n x ++ aux (n+1) xs

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por recursión, la función
--    itera :: (a -> a) -> a -> [a]
-- tal que (itera f x) es la lista cuyo primer elemento es x y los
-- siguientes elementos se calculan aplicando la función f al elemento
-- anterior. Por ejemplo,
--    Main> itera (+1) 3
--    [3,4,5,6,7,8,9,10,11,12,{Interrupted!}
```

```
--    Main> itera (*2) 1
--    [1,2,4,8,16,32,64,{Interrupted!}
--    Main> itera (`div` 10) 1972
--    [1972,197,19,1,0,0,0,0,0,0,{Interrupted!}
-- Nota: La función repite es equivalente a la función iterate definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------

itera :: (a -> a) -> a -> [a]
itera f x = x : itera f (f x)

-- ---------------------------------------------------------------------------
-- Ejercicio 6, Definir la función
--    agrupa :: Int -> [a] -> [[a]]
-- tal que (agrupa n xs) es la lista formada por listas de n elementos
-- conscutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--    Main> agrupa 2 [3,1,5,8,2,7]
--    [[3,1],[5,8],[2,7]]
--    Main> agrupa 2 [3,1,5,8,2,7,9]
--    [[3,1],[5,8],[2,7],[9]]
--    Main> agrupa 5 "todo necio confunde valor y precio"
--    ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-- ---------------------------------------------------------------------------

-- Una definición recursiva de agrupa es
agrupa :: Int -> [a] -> [[a]]
agrupa n [] = []
agrupa n xs = take n xs : agrupa n (drop n xs)

-- Una definición no recursiva es
agrupa' :: Int -> [a] -> [[a]]
agrupa' n = takeWhile (not . null)
          . map (take n)
          . iterate (drop n)

-- Puede verse su funcionamiento en el siguiente ejemplo,
--    iterate (drop 2) [5..10]
--    ==> [[5,6,7,8,9,10],[7,8,9,10],[9,10],[],[],...
--    map (take 2) (iterate (drop 2) [5..10])
```

```
--    ==> [[5,6],[7,8],[9,10],[],[],[],[],...
--    takeWhile (not . null) (map (take 2) (iterate (drop 2) [5..10]))
--    ==> [[5,6],[7,8],[9,10]]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir, y comprobar, con QuickCheck las dos propiedades
-- que caracterizan a la función agrupa:
-- * todos los grupos tienen que tener la longitud determinada (salvo el
--   último que puede tener una longitud menor) y
-- * combinando todos los grupos se obtiene la lista inicial.
-- ---------------------------------------------------------------------

-- La primera propiedad es
prop_AgrupaLongitud :: Int -> [Int] -> Property
prop_AgrupaLongitud n xs =
    n > 0 && not (null gs) ==>
      and [length g == n | g <- init gs] &&
      0 < length (last gs) && length (last gs) <= n
    where gs = agrupa n xs

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

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

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

-- ---------------------------------------------------------------
-- Sea la siguiente operación, aplicable a cualquier número entero
-- positivo:
--    * Si el número es par, se divide entre 2.
--    * Si el número es impar, se multiplica por 3 y se suma 1.
-- Dado un número cualquiera, podemos considerar su órbita, es decir,
-- las imágenes sucesivas al iterar la función. Por ejemplo, la órbita
```

```
-- de 13 es
--    13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1,...
-- Si observamos este ejemplo, la órbita de 13 es periódica, es decir,
-- se repite indefinidamente a partir de un momento dado). La conjetura
-- de Collatz dice que siempre alcanzaremos el 1 para cualquier número
-- con el que comencemos. Ejemplos:
--    * Empezando en n = 6 se obtiene 6, 3, 10, 5, 16, 8, 4, 2, 1.
--    * Empezando en n = 11 se obtiene: 11, 34, 17, 52, 26, 13, 40, 20,
--      10, 5, 16, 8, 4, 2, 1.
--    * Empezando en n = 27, la sucesión tiene 112 pasos, llegando hasta
--      9232 antes de descender a 1:  27, 82, 41, 124, 62, 31, 94, 47,
--      142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274,
--      137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263,
--      790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502,
--      251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
--      479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644,
--      1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308,
--      1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
--      61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,
--      16, 8, 4, 2, 1.

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    siguiente :: Integer -> Integer
-- tal que (siguiente n) es el siguiente de n en la sucesión de
-- Collatz. Por ejemplo,
--    siguiente 13  ==  40
--    siguiente 40  ==  20
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir, por recursión, la función
--    collatz :: Integer -> [Integer]
-- tal que (collatz n) es la órbita de Collatz d n hasta alcanzar el
-- 1. Por ejemplo,
--    collatz 13  ==  [13,40,20,10,5,16,8,4,2,1]
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir, sin recursión, la función
--    collatz' :: Integer -> [Integer]
-- tal que (collatz' n) es la órbita de Collatz d n hasta alcanzar el
-- 1. Por ejemplo,
--    collatz' 13  ==  [13,40,20,10,5,16,8,4,2,1]
-- Indicación: Usar takeWhile e iterate.
-- ---------------------------------------------------------------------

collatz' :: Integer -> [Integer]
collatz' n = (takeWhile (/=1) (iterate siguiente n)) ++ [1]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    menorCollatzMayor :: Int -> Integer
-- tal que (menorCollatzMayor x) es el menor número cuya órbita de
-- Collatz tiene más de x elementos. Por ejemplo,
--    menorCollatzMayor 100  ==  27
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    menorCollatzSupera :: Integer -> Integer
-- tal que (menorCollatzSupera x) es el menor número cuya órbita de
-- Collatz tiene algún elemento mayor que x. Por ejemplo,
--    menorCollatzSupera 100  ==  15
-- ---------------------------------------------------------------------

menorCollatzSupera :: Integer -> Integer
menorCollatzSupera x =
    head [y | y <- [1..], maximum (collatz y) > x]
```

```
-- Otra definición alternativa es
menorCollatzSupera' :: Integer -> Integer
menorCollatzSupera' x = head [n | n <- [1..], t <- collatz' n, t > x]
```

Relación 16

Ejercicios de exámenes del curso 2010-11

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

-- En esta relación se presenta los ejercicios de exámenes del curso
-- 2010-11.

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir por recursión la función
--    sumaR :: Num b => (a -> b) -> [a] -> b
-- tal que (suma f xs) es la suma de los valores obtenido aplicando la
-- función f a lo elementos de la lista xs. Por ejemplo,
--    sumaR (*2)  [3,5,10]  ==  36
--    sumaR (/10) [3,5,10]  ==  1.8
 -- ---------------------------------------------------------------------

sumaR :: Num b => (a -> b) -> [a] -> b
sumaR f []     = 0
sumaR f (x:xs) = f x + sumaR f xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir por plegado la función
--    sumaP :: Num b => (a -> b) -> [a] -> b
-- tal que (suma f xs) es la suma de los valores obtenido aplicando la
-- función f a lo elementos de la lista xs. Por ejemplo,
--    sumaP (*2)  [3,5,10]  ==  36
--    sumaP (/10) [3,5,10]  ==  1.8
-- ---------------------------------------------------------------------

sumaP :: Num b => (a -> b) -> [a] -> b
sumaP f = foldr (\x y -> (f x) + y) 0

-- ---------------------------------------------------------------------
-- Ejercicio 3. El enunciado del problema 1 de la Olimpiada
-- Iberoamericana de Matemática Universitaria del 2006 es el siguiente:
--    Sean m y n números enteros mayores que 1. Se definen los conjuntos
--    P(m) = {1/m, 2/m,..., (m-1)/m} y P(n) = {1/n, 2/n,..., (n-1)/n}.
--    Encontrar la distancia entre P(m) y P(n), que se define como
--    mín {|a - b| : a en P(m), b en P(n)}.
-- Definir la función
--    distancia :: Float -> Float -> Float
-- tal que (distancia m n) es la distancia entre P(m) y P(n). Por
-- ejemplo,
--    distancia 2 7 == 7.142857e-2
--    distancia 2 8 == 0.0
-- ---------------------------------------------------------------------

distancia :: Float -> Float -> Float
distancia m n =
    minimum [abs (i/m - j/n) | i <- [1..m-1], j <- [1..n-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 4. El enunciado del problema 580 de "Números y
-- algo más.." es el siguiente:
--    ¿Cuál es el menor número que puede expresarse como la suma de 9,
--    10 y 11 números consecutivos?
-- (El problema se encuentra en http://goo.gl/1K3t7 )
-- A lo largo de los distintos apartados de este ejercicio se resolverá
-- el problema.
-- ---------------------------------------------------------------------
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    consecutivosConSuma :: Int -> Int -> [[Int]]
-- tal que (consecutivosConSuma x n) es la lista de listas de n números
-- consecutivos cuya suma es x. Por ejemplo,
--    consecutivosConSuma 12 3  ==  [[3,4,5]]
--    consecutivosConSuma 10 3  ==  []
-- ---------------------------------------------------------------------

consecutivosConSuma :: Int -> Int -> [[Int]]
consecutivosConSuma x n =
    [[y..y+n-1] | y <- [1..x], sum [y..y+n-1] == x]

-- Se puede hacer una definición sin búsqueda, ya que por la fórmula de
-- la suma de progresiones aritméticas, la expresión
--    sum [y..y+n-1] == x
-- se reduce a
--    (y+(y+n-1))n/2 = x
-- De donde se puede despejar la y, ya que
--    2yn+n^2-n = 2x
--    y = (2x-n^2+n)/2n
-- De la anterior anterior se obtiene la siguiente definición de
-- consecutivosConSuma que no utiliza búsqueda.

consecutivosConSuma' :: Int -> Int -> [[Int]]
consecutivosConSuma' x n
    | z >= 0 && mod z (2*n) == 0 = [[y..y+n-1]]
    | otherwise                  = []
    where z = 2*x-n^2+n
          y = div z (2*n)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    esSuma :: Int -> Int -> Bool
-- tal que (esSuma x n) se verifica si x es la suma de n números
-- naturales consecutivos. Por ejemplo,
--    esSuma 12 3  ==  True
--    esSuma 10 3  ==  False
-- ---------------------------------------------------------------------
```

```
esSuma :: Int -> Int -> Bool
esSuma x n = consecutivosConSuma x n /= []

-- También puede definirse directamente sin necesidad de
-- consecutivosConSuma como se muestra a continuación.
esSuma' :: Int -> Int -> Bool
esSuma' x n = or [sum [y..y+n-1] == x | y <- [1..x]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    menorQueEsSuma :: [Int] -> Int
-- tal que (menorQueEsSuma ns) es el menor número que puede expresarse
-- como suma de tantos números consecutivos como indica ns. Por ejemplo,
--    menorQueEsSuma [3,4]  ==  18
-- Lo que indica que 18 es el menor número se puede escribir como suma
-- de 3 y de 4 números consecutivos. En este caso, las sumas son
-- 18 = 5+6+7 y 18 = 3+4+5+6.
-- ---------------------------------------------------------------------

menorQueEsSuma :: [Int] -> Int
menorQueEsSuma ns =
    head [x | x <- [1..], and [esSuma x n | n <- ns]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Usando la función menorQueEsSuma calcular el menor
-- número que puede expresarse como la suma de 9, 10 y 11 números
-- consecutivos.
-- ---------------------------------------------------------------------

-- La solución es
--    *Main> menorQueEsSuma [9,10,11]
--    495

-- ---------------------------------------------------------------------
-- Ejercicio 5. (Problema 303 del proyecto Euler) Definir la función
--    multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
-- tal que (multiplosRestringidos n x) es la lista de los múltiplos de n
-- tales que todas sus cifras verifican la propiedad p. Por ejemplo,
--    take 4 (multiplosRestringidos 5 (<=3))  ==  [10,20,30,100]
```

```haskell
--    take 5 (multiplosRestringidos 3 (<=4))  ==  [3,12,21,24,30]
--    take 5 (multiplosRestringidos 3 even)   ==  [6,24,42,48,60]
-- ---------------------------------------------------------------------

multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
multiplosRestringidos n p =
    [y | y <- [n,2*n..], all p (cifras y)]

-- (cifras n) es la lista de las cifras de n, Por ejemplo,
--    cifras 327  ==  [3,2,7]
cifras :: Int -> [Int]
cifras n = [read [x] | x <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 6. 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)]
-- 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, x < n-x, elem (n-x) primosN]
    where primosN = takeWhile (<=n) primos

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. [2 puntos] 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 [1,2,0,4,5,6,48,7,2]  ==  [[],[2,0,4],[6,48],[2]]
-- ---------------------------------------------------------------------

segmentos :: (a -> Bool) -> [a] -> [[a]]
segmentos _ [] = []
segmentos p xs =
    takeWhile p xs : (segmentos p (dropWhile (not.p) (dropWhile p xs)))
```

Relación 17

Combinatoria

```
-- ---------------------------------------------------------------------
-- 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.
-- Además, se estudia dos temas relacionados:
--    * Reconocimiento y generación de subconjuntos y
--    * El triángulo de Pascal

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

import Test.QuickCheck
import Data.List (genericLength)

-- ---------------------------------------------------------------------
-- § Subconjuntos
-- ---------------------------------------------------------------------
```

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

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

-- ---------------------------------------------------------------------
-- 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
-- y subconjunto' son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_equivalencia :: [Int] -> [Int] -> 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

-- 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
```

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

intercala :: a -> [a] -> [[a]]
intercala x [] = [[x]]
intercala x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala x ys]

-- ---------------------------------------------------------------------
-- 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 []     = [[]]
permutaciones (x:xs) =
    concat [intercala x ys | ys <- permutaciones xs]

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

permutacionesN :: Integer -> [[Integer]]
permutacionesN n = permutaciones [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir, usando permutacionesN, la función
--    numeroPermutacionesN :: Integer -> Integer
-- tal que (numeroPermutacionesN n) es el número de permutaciones de un
```

```haskell
-- conjunto con n elementos. Por ejemplo,
--    numeroPermutacionesN 3  ==  6
--    numeroPermutacionesN 4  ==  24
-- ---------------------------------------------------------------------

numeroPermutacionesN :: Integer -> Integer
numeroPermutacionesN = genericLength . permutacionesN

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

```haskell
prop_numeroPermutacionesN n =
    and [numeroPermutacionesN x == numeroPermutacionesN' x | x <- [1..n]]

-- ---------------------------------------------------------------------
-- § Combinaciones
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- 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
combinaciones_1 :: Integer -> [a] -> [[a]]
combinaciones_1 n xs =
    [ys | ys <- subconjuntos xs, genericLength ys == n]

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

-- La anterior definición se puede escribir usando map:
combinaciones_3 :: Integer -> [a] -> [[a]]
combinaciones_3 0 _ = [[]]
combinaciones_3 _ [] = []
combinaciones_3 (k+1) (x:xs) =
    map (x:) (combinaciones_3 k xs) ++ combinaciones_3 (k+1) xs

-- Nota. La segunda definición es más eficiente como se comprueba en la
```

```
-- siguiente sesión
--    ghci> :set +s
--    ghci> length (combinaciones_1 2 [1..15])
--    105
--    (0.19 secs, 6373848 bytes)
--    ghci> length (combinaciones_2 2 [1..15])
--    105
--    (0.01 secs, 525360 bytes)
--    ghci> length (combinaciones_3 2 [1..15])
--    105
--    (0.02 secs, 528808 bytes)

-- En lo que sigue, usaremos combinaciones como combinaciones_2
combinaciones :: Integer -> [a] -> [[a]]
combinaciones = combinaciones_2

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    combinacionesN :: Integer -> Integer -> [[Int]]
-- 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]]
-- ---------------------------------------------------------------------

combinacionesN :: Integer -> Integer -> [[Integer]]
combinacionesN n k = combinaciones k [1..n]

-- ---------------------------------------------------------------------
-- 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)

-- Puede definirse por composición
numeroCombinaciones_2 :: Integer -> Integer -> Integer
numeroCombinaciones_2 = (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
numeroCombinaciones_3 :: Integer -> Integer -> Integer
numeroCombinaciones_3 = 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
--    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]]
-- ---------------------------------------------------------------------

combinacionesRN :: Integer -> Integer -> [[Integer]]
combinacionesRN n k = combinacionesR k [1..n]

-- ---------------------------------------------------------------------
-- 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
-- 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 =
  concat (map 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
-- 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 = genericLength (variacionesN n k)

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

-- ---------------------------------------------------------------------
-- § 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 _ [] = [[]]
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]]

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

-- ---------------------------------------------------------------------
-- Ejercicio 34.1. 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 la función
--    pascal :: Integer -> [Integer]
-- tal que (pascal n) es la n-ésima fila del triángulo de Pascal. Por
-- ejemplo,
--    pascal 6  ==  [1,5,10,10,5,1]
-- ---------------------------------------------------------------------

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

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

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

-- las definiciones son equivalentes
prop_pares :: [Integer] -> Bool
prop_pares xs =
    pares xs == pares' xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 34.2. Comprobar con QuickCheck, que la fila n-ésima del
-- triángulo de Pascal tiene n elementos.
```

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 34.3. Comprobar con QuickCheck, que la suma de los
-- elementos de la fila n-ésima del triángulo de Pascal es igual a
-- 2^(n-1).
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 34.4. Comprobar con QuickCheck, que el m-ésimo elemento de
-- la fila (n+1)-ésima del triángulo de Pascal es el número combinatorio
-- (comb n m).
-- ---------------------------------------------------------------------

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

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

Relación 18

Tipos de datos algebraicos

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

-- En esta relación se presenta ejercicios sobre tipos de datos
-- algebraicos. Se consideran dos tipos de datos algebraicos: los
-- números naturales (para los que se define su producto) y los árboles
-- binarios, para los que se definen funciones para calcular:
--   * la ocurrencia de un elemento en el árbol,
--   * el número de hojas
--   * el carácter balanceado de un árbol,
--   * el árbol balanceado correspondiente a una lista,
--
-- Los ejercicios corresponden al tema 9 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-9.pdf

-- ---------------------------------------------------------------------
-- Ejercicio 1. Usando el tipo de dato Nat y la función suma definidas
-- en las transparencias del tema 9, definir la función
--    producto :: Nat -> Nat -> Nat
-- tal que (producto m n) es el producto de los números naturales m y
-- n. Por ejemplo,
--    ghci> producto (Suc (Suc Cero)) (Suc (Suc (Suc Cero)))
--    Suc (Suc (Suc (Suc (Suc (Suc Cero)))))
-- ---------------------------------------------------------------------
```

```
data Nat = Cero | Suc Nat
           deriving (Eq, Show)

suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)

producto :: Nat -> Nat -> Nat
producto Cero _    = Cero
producto (Suc m) n = suma n (producto m n)

-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se trabajará con árboles binarios
-- definidos como sigue
--    data Arbol = Hoja Int
--               | Nodo Arbol Int Arbol
--               deriving (Show, Eq)
-- Por ejemplo, el árbol
--         5
--        / \
--       /   \
--      3     7
--     / \   / \
--    1   4 6   9
-- se representa por
--    Nodo (Nodo (Hoja 1) 3 (Hoja 4))
--         5
--         (Nodo (Hoja 6) 7 (Hoja 9))
-- ---------------------------------------------------------------------

data Arbol = Hoja Int
           | Nodo Arbol Int Arbol
           deriving (Show, Eq)

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

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

```
-- Ejercicio 2. Definir la función
--    ocurre :: Int -> Arbol -> Bool
-- tal que (ocurre x a) se verifica si x ocurre en el árbol a como valor
-- de un nodo o de una hoja. Por ejemplo,
--    ocurre  4 ejArbol  ==  True
--    ocurre 10 ejArbol  ==  False
-- ---------------------------------------------------------------------

ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d

-- ---------------------------------------------------------------------
-- Ejercicio 3. En el preludio está definido el tipo de datos
--    data Ordering = LT | EQ | GT
-- junto con la función
--    compare :: Ord a => a -> a -> Ordering
-- que decide si un valor en un tipo ordenado es menor (LT), igual (EQ)
-- o mayor (GT) que otro.
--
-- Usando esta función, redefinir la función
--    ocurre :: Int -> Arbol -> Bool
-- del ejercicio anterior.
-- ---------------------------------------------------------------------

ocurre' :: Int -> Arbol -> Bool
ocurre' m (Hoja n)     = m == n
ocurre' m (Nodo i n d) = case compare m n of
                           LT -> ocurre' m i
                           EQ -> True
                           GT -> ocurre' m d

-- ---------------------------------------------------------------------
-- Ejercicio 4. ¿Porqué la segunda definición de ocurre es más eficiente
-- que la primera?
-- ---------------------------------------------------------------------

-- La nueva definición es más eficiente porque sólo necesita una
-- comparación por nodo, mientras que la definición de las
-- transparencias necesita dos comparaciones por nodo.
```

```
-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se trabajará con árboles binarios
-- definidos como sigue
--    type ArbolB = HojaB Int
--                | NodoB ArbolB ArbolB
--                deriving Show
-- Por ejemplo, el árbol
--         .
--        / \
--       /   \
--      .     .
--     / \   / \
--    1   4 6   9
-- se representa por
--    NodoB (NodoB (HojaB 1) (HojaB 4))
--          (NodoB (HojaB 6) (HojaB 9))
-- ---------------------------------------------------------------------

data ArbolB = HojaB Int
            | NodoB ArbolB ArbolB
            deriving Show

ejArbolB :: ArbolB
ejArbolB = NodoB (NodoB (HojaB 1) (HojaB 4))
                 (NodoB (HojaB 6) (HojaB 9))

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    nHojas :: ArbolB -> Int
-- tal que (nHojas a) es el número de hojas del árbol a. Por ejemplo,
--    nHojas (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7)))  ==  3
--    nHojas ejArbolB ==  4
-- ---------------------------------------------------------------------

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

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

```
-- Ejercicio 6. Se dice que un árbol de este tipo es balanceado si es
-- una hoja o bien si para cada nodo se tiene que el número de hojas en
-- cada uno de sus subárboles difiere como máximo en uno y sus
-- subárboles son balanceados. Definir la función
--    balanceado :: ArbolB -> BoolB
-- tal que (balanceado a) se verifica si a es un árbol balanceado. Por
-- ejemplo,
--    balanceado ejArbolB
--    ==> True
--    balanceado (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7)))
--    ==> True
--    balanceado (NodoB (HojaB 5) (NodoB (HojaB 3) (NodoB (HojaB 5) (HojaB 7))))
--    ==> False
-- ---------------------------------------------------------------------

balanceado :: ArbolB -> Bool
balanceado (HojaB _)     = True
balanceado (NodoB a1 a2) = abs (nHojas a1 - nHojas a2) <= 1 &&
                           balanceado a1 &&
                           balanceado a2

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    mitades :: [a] -> ([a],[a])
-- tal que (mitades xs) es un par de listas que se obtiene al dividir xs
-- en dos mitades cuya longitud difiere como máximo en uno. Por ejemplo,
--    mitades [2,3,5,1,4,7]    ==  ([2,3,5],[1,4,7])
--    mitades [2,3,5,1,4,7,9]  ==  ([2,3,5],[1,4,7,9])
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    arbolBalanceado :: [Int] -> ArbolB
-- tal que (arbolBalanceado xs) es el árbol balanceado correspondiente
-- a la lista xs. Por ejemplo,
--    ghci> arbolBalanceado [2,5,3]
--    NodoB (HojaB 2) (NodoB (HojaB 5) (HojaB 3))
```

```
--    ghci> arbolBalanceado [2,5,3,7]
--    NodoB (NodoB (HojaB 2) (HojaB 5)) (NodoB (HojaB 3) (HojaB 7))
-- ---------------------------------------------------------------------

arbolBalanceado :: [Int] -> ArbolB
arbolBalanceado [x] = HojaB x
arbolBalanceado xs  = NodoB (arbolBalanceado ys) (arbolBalanceado zs)
                      where (ys,zs) = mitades xs
```

Relación 19

Tipos de datos algebraicos: árboles binarios

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

-- En esta relación se plantean ejercicios sobre árboles binarios. En
-- concreto, se definen funciones para calcular:
-- * el número de hojas de un árbol,
-- * el número de nodos de un árbol,
-- * la profundidad de un árbol,
-- * el recorrido preorden de un árbol,
-- * el recorrido postorden de un árbol,
-- * el recorrido preorden de forma iterativa,
-- * la imagen especular de un árbol,
-- * el subárbol de profundidad dada,
-- * el árbol infinito generado con un elemento y
-- * el árbol de profundidad dada cuyos nodos son iguales a un elemento.
--
-- Estos ejercicios corresponden al tema 9 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-9.pdf

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

```
import Data.List

-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se trabajará con los árboles
-- binarios definidos como sigue
--    data Arbol a = Hoja
--                 | Nodo a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- En los ejemplos se usará el siguiente árbol
--    arbol = Nodo 9
--                   (Nodo 3
--                         (Nodo 2 Hoja Hoja)
--                         (Nodo 4 Hoja Hoja))
--                   (Nodo 7 Hoja Hoja)
-- ---------------------------------------------------------------------

data Arbol a = Hoja
             | Nodo a (Arbol a) (Arbol a)
             deriving (Show, Eq)

arbol = Nodo 9
               (Nodo 3
                     (Nodo 2 Hoja Hoja)
                     (Nodo 4 Hoja Hoja))
               (Nodo 7 Hoja Hoja)

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    nHojas :: Arbol a -> Int
-- tal que (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--    ghci> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    ghci> nHojas arbol
--    6
-- ---------------------------------------------------------------------

nHojas :: Arbol a -> Int
nHojas Hoja         = 1
nHojas (Nodo x i d) = nHojas i + nHojas d
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    nNodos :: Arbol a -> Int
-- tal que (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--    ghci> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    ghci> nNodos arbol
--    5
-- ---------------------------------------------------------------------

nNodos :: Arbol a -> Int
nNodos Hoja         = 0
nNodos (Nodo x i d) = 1 + nNodos i + nNodos d

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    profundidad :: Arbol a -> Int
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
--    ghci> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    ghci> profundidad arbol
--    3
-- ---------------------------------------------------------------------

profundidad :: Arbol a -> Int
profundidad Hoja = 0
profundidad (Nodo x i d) = 1 + max (profundidad i) (profundidad d)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    preorden :: Arbol a -> [a]
-- tal que (preorden x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--    ghci> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    ghci> preorden arbol
--    [9,3,2,4,7]
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. 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,
--    ghci> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    ghci> postorden arbol
--    [2,4,3,7,9]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. 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,
--    ghci> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    ghci> preordenIt arbol
--    [9,3,2,4,7]
-- Nota: No usar (++) en la definición
-- ---------------------------------------------------------------------

preordenIt :: Arbol a -> [a]
preordenIt x = preordenItAux x []
    where preordenItAux Hoja xs         = xs
          preordenItAux (Nodo x i d) xs =
```

```
              x : preordenItAux i (preordenItAux d xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--    ghci> espejo arbol
--    Nodo 9
--         (Nodo 7 Hoja Hoja)
--         (Nodo 3
--               (Nodo 4 Hoja Hoja)
--               (Nodo 2 Hoja Hoja))
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 8. 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,
--    ghci> takeArbol 0 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--    Hoja
--    ghci> takeArbol 1 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--    Nodo 6 Hoja Hoja
--    ghci> takeArbol 2 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--    Nodo 6 Hoja (Nodo 7 Hoja Hoja)
--    ghci> takeArbol 3 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--    Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja)
--    ghci> takeArbol 4 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--    Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja)
-- ---------------------------------------------------------------------
```

```
takeArbol :: Int -> Arbol a -> Arbol a
takeArbol 0     _  = Hoja
takeArbol _ Hoja   = Hoja
takeArbol n (Nodo x i d) =
    Nodo x (takeArbol (n-1) i) (takeArbol (n-1) d)

-- ---------------------------------------------------------------------
-- Ejercicio 9. La función
--    repeat :: a -> [a]
-- está definida de forma que (repeat x) es la lista formada por
-- infinitos elementos x. Por ejemplo,
--    repeat 3  ==  [3,3,3,3,3,3,3,3,3,3,3,3,3,...
-- La definición de repeat es
--    repeat x = xs where xs = x:xs
-- Definir la función
--    repeatArbol :: a -> Arbol a
-- tal que (repeatArbol x) es es árbol con infinitos nodos x. Por
-- ejemplo,
--    ghci> takeArbol 0 (repeatArbol 3)
--    Hoja
--    ghci> takeArbol 1 (repeatArbol 3)
--    Nodo 3 Hoja Hoja
--    ghci> takeArbol 2 (repeatArbol 3)
--    Nodo 3 (Nodo 3 Hoja Hoja) (Nodo 3 Hoja Hoja)
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10. 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,
--    ghci> replicateArbol 0 5
--    Hoja
--    ghci> replicateArbol 1 5
--    Nodo 5 Hoja Hoja
--    ghci> replicateArbol 2 5
--    Nodo 5 (Nodo 5 Hoja Hoja) (Nodo 5 Hoja Hoja)
-- ---------------------------------------------------------------------

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

Relación 20

Tipos de datos algebraicos: fórmulas proposicionales

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

--  En esta relación se extiende el demostrador proposicional estudiado
--  en el tema 9 para incluir disyunciones y equivalencias.
--
-- Las transparencias del tema 9 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-9.pdf

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

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Extender el procedimiento de decisión de tautologías
-- para incluir las disyunciones (Disj) y las equivalencias (Equi). Por
-- ejemplo,
--    ghci> esTautologia (Equi (Var 'A') (Disj (Var 'A') (Var 'A')))
--    True
--    ghci> esTautologia (Equi (Var 'A') (Disj (Var 'A') (Var 'B')))
--    False
-- Se incluye el código del procedimiento visto en clase para que se
```

```
-- extienda de manera adecuada.
-- ---------------------------------------------------------------------

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Disj FProp FProp   -- Añadido
           | Impl FProp FProp
           | Equi FProp FProp   -- Añadido
           deriving Show

type Interpretacion = [(Char, Bool)]

valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Disj p q) = valor i p || valor i q    -- Añadido
valor i (Impl p q) = valor i p <= valor i q
valor i (Equi p q) = valor i p == valor i q    -- Añadido

busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']

variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Disj p q) = variables p ++ variables q   -- Añadido
variables (Impl p q) = variables p ++ variables q
variables (Equi p q) = variables p ++ variables q   -- Añadido

interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0     = [[]]
interpretacionesVar (n+1) =
    map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar n
```

```
interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
    map (zip vs) (interpretacionesVar (length vs))
    where vs = nub (variables p)

esTautologia :: FProp -> Bool
esTautologia p =
    and [valor i p | i <- interpretaciones p]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    interpretacionesVar' :: Int -> [[Bool]]
-- que sea equivalente a interpretacionesVar pero que en su definición
-- use listas de comprensión en lugar de map. Por ejemplo,
--    ghci> interpretacionesVar' 2
--    [[False,False],[False,True],[True,False],[True,True]]
-- ---------------------------------------------------------------------

interpretacionesVar' :: Int -> [[Bool]]
interpretacionesVar' 0     = [[]]
interpretacionesVar' (n+1) =
    [False:bs | bs <- bss] ++ [True:bs | bs <- bss]
    where bss = interpretacionesVar' n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    interpretaciones' :: FProp -> [Interpretacion]
-- que sea equivalente a interpretaciones pero que en su definición
-- use listas de comprensión en lugar de map. Por ejemplo,
--    ghci> interpretaciones' (Impl (Var 'A') (Conj (Var 'A') (Var 'B')))
--    [[('A',False),('B',False)],
--     [('A',False),('B',True)],
--     [('A',True),('B',False)],
--     [('A',True),('B',True)]]
-- ---------------------------------------------------------------------

interpretaciones' :: FProp -> [Interpretacion]
interpretaciones' p =
    [zip vs i | i <- is]
```

```
    where vs = nub (variables p)
          is = interpretacionesVar (length vs)
```

Relación 21

Tipos de datos algebraicos: Modelización de juego de cartas

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

-- En esta relación se estudia la modelización de un juego de cartas
-- como aplicación de los tipos de datos algebraicos. Además, se definen
-- los generadores correspondientes para comprobar las propiedades con
-- QuickCheck.
--
-- Estos ejercicios corresponden al tema 9 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-9.pdf

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

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

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de datos Palo para representar
-- los cuatro palos de la baraja: picas, corazones, diamantes y
-- tréboles. Hacer que Palo sea instancia de Eq y Show.
```

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

-- La definición es
data Palo = Picas | Corazones | Diamantes | Treboles
            deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Palo se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Palo where
    arbitrary = elements [Picas, Corazones, Diamantes, Treboles]

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de dato Color para representar los
-- colores de las cartas: rojo y negro. Hacer que Color sea instancia de
-- Show.
-- ---------------------------------------------------------------------

data Color = Rojo | Negro
             deriving Show

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    color :: Palo -> Color
-- tal que (color p) es el color del palo p. Por ejemplo,
--    color Corazones ==>  Rojo
-- Nota: Los corazones y los diamantes son rojos. Las picas y los
-- tréboles son negros.
-- ---------------------------------------------------------------------

color :: Palo -> Color
color Picas     = Negro
color Corazones = Rojo
color Diamantes = Rojo
color Treboles  = Negro

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Los valores de las cartas se dividen en los
```

```
-- numéricos (del 2 al 10) y las figuras (sota, reina, rey y
-- as). Definir el tipo -- de datos Valor para representar los valores
-- de las cartas. Hacer que Valor sea instancia de Eq y Show.
--    Main> :type Sota
--    Sota :: Valor
--    Main> :type Reina
--    Reina :: Valor
--    Main> :type Rey
--    Rey :: Valor
--    Main> :type As
--    As :: Valor
--    Main> :type Numerico 3
--    Numerico 3 :: Valor
-- ---------------------------------------------------------------------

data Valor = Numerico Int | Sota | Reina | Rey | As
             deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Valor se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Valor where
  arbitrary =
    oneof $
      [ do return c
      | c <- [Sota,Reina,Rey,As]
      ] ++
      [ do n <- choose (2,10)
           return (Numerico n)
      ]

-- ---------------------------------------------------------------------
-- Ejercicio 2. El orden de valor de las cartas (de mayor a menor) es
-- as, rey, reina, sota y las numéricas según su valor. Definir la
-- función
--    mayor :: Valor -> Valor -> Bool
-- tal que (mayor x y) se verifica si la carta x es de mayor valor que
-- la carta y. Por ejemplo,
```

```
--    mayor Sota (Numerico 7)    ==>  True
--    mayor (Numerico 10) Reina  ==>  False
-- ---------------------------------------------------------------------

mayor :: Valor -> Valor -> Bool
mayor _            As           = False
mayor As           _            = True
mayor _            Rey          = False
mayor Rey          _            = True
mayor _            Reina        = False
mayor Reina        _            = True
mayor _            Sota         = False
mayor Sota         _            = True
mayor (Numerico m) (Numerico n) = m > n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comprobar con QuickCheck si dadas dos cartas, una
-- siempre tiene mayor valor que la otra. En caso de que no se verifique,
-- añadir la menor precondición para que lo haga.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MayorValor1 a b =
    mayor a b || mayor b a

-- La comprobación es
--    Main> quickCheck prop_MayorValor1
--    Falsifiable, after 2 tests:
--    Sota
--    Sota
-- que indica que la propiedad es falsa porque la sota no tiene mayor
-- valor que la sota.

-- La precondición es que las cartas sean distintas:
prop_MayorValor a b =
    a /= b ==> mayor a b || mayor b a

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de datos Carta para representar
-- las cartas mediante un valor y un palo. Hacer que Carta sea instancia
-- de Eq y Show. Por ejemplo,
--    Main> :type Carta Rey Corazones
--    Carta Rey Corazones :: Carta
--    Main> :type Carta (Numerico 4) Corazones
--    Carta (Numerico 4) Corazones :: Carta
-- ---------------------------------------------------------------------

data Carta = Carta Valor Palo
             deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    valor :: Carta -> Valor
-- tal que (valor c) es el valor de la carta c. Por ejemplo,
--    valor (Carta Rey Corazones)  ==>  Rey
-- ---------------------------------------------------------------------

valor :: Carta -> Valor
valor (Carta v p) = v

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    palo :: Carta -> Valor
-- tal que (palo c) es el palo de la carta c. Por ejemplo,
--    palo (Carta Rey Corazones)  ==>  Corazones
-- ---------------------------------------------------------------------

palo :: Carta -> Palo
palo (Carta v p) = p

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Carta se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Carta where
```

```
    arbitrary =
        do v <- arbitrary
           p <- arbitrary
           return (Carta v p)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    ganaCarta :: Palo -> Carta -> Carta -> Bool
-- tal que (ganaCarta p c1 c2) se verifica si la carta c1 le gana a la
-- carta c2 cuando el palo de triunfo es p (es decir, las cartas son del
-- mismo palo y el valor de c1 es mayor que el de c2 o c1 es del palo de
-- triunfo). Por ejemplo,
--    ganaCarta Corazones (Carta Sota Picas) (Carta (Numerico 5) Picas)
--    ==> True
--    ganaCarta Corazones (Carta (Numerico 3) Picas) (Carta Sota Picas)
--    ==> False
--    ganaCarta Corazones (Carta (Numerico 3) Corazones) (Carta Sota Picas)
--    ==> True
--    ganaCarta Treboles (Carta (Numerico 3) Corazones) (Carta Sota Picas)
--    ==> False
-- ---------------------------------------------------------------------

ganaCarta :: Palo -> Carta -> Carta -> Bool
ganaCarta triunfo c c'
    | palo c == palo c' = mayor (valor c) (valor c')
    | palo c == triunfo = True
    | otherwise         = False

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck si dadas dos cartas, una
-- siempre gana a la otra.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_GanaCarta t c1 c2 =
    ganaCarta t c1 c2 || ganaCarta t c2 c1

-- La comprobación es
--    Main> quickCheck prop_GanaCarta
--    Falsifiable, after 0 tests:
```

```
--    Diamantes
--    Carta Rey Corazones
--    Carta As Treboles
-- que indica que la propiedad no se verifica ya que cuando el triunfo
-- es diamantes, ni el rey de corazones le gana al as de tréboles ni el
-- as de tréboles le gana al rey de corazones.

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de datos Mano para representar
-- una  mano en el juego de cartas. Una mano es vacía o se obtiene
-- agregando una carta a una mano. Hacer Mano instancia de Eq y
-- Show. Por ejemplo,
--    Main> :type Agrega (Carta Rey Corazones) Vacia
--    Agrega (Carta Rey Corazones) Vacia :: Mano
-- ---------------------------------------------------------------------

data Mano = Vacia | Agrega Carta Mano
            deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Mano se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Mano where
    arbitrary =
        do cs <- arbitrary
           let mano []     = Vacia
               mano (c:cs) = Agrega c (mano cs)
           return (mano cs)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Una mano gana a una carta c si alguna carta de la mano
-- le gana a c. Definir la función
--    ganaMano :: Palo -> Mano -> Carta -> Bool
-- tal que (gana t m c) se verifica si la mano m le gana a la carta c
-- cuando el triunfo es t. Por ejemplo,
--    ganaMano Picas (Agrega (Carta Sota Picas) Vacia) (Carta Rey Corazones)
--    ==>  True
--    ganaMano Picas (Agrega (Carta Sota Picas) Vacia) (Carta Rey Picas)
```

```
--    ==>  False
-- ---------------------------------------------------------------------

ganaMano :: Palo -> Mano -> Carta -> Bool
ganaMano triunfo Vacia        c' = False
ganaMano triunfo (Agrega c m) c' = ganaCarta triunfo c c' ||
                                   ganaMano triunfo m c'

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    eligeCarta :: Palo -> Carta -> Mano -> Carta
-- tal que (eligeCarta t c1 m) es la mejor carta de la mano m frente a
-- la carta c cuando el triunfo es t. La estrategia para elegir la mejor
-- carta es la siguiente:
-- * Si la mano sólo tiene una carta, se elige dicha carta.
-- * Si la primera carta de la mano es del palo de c1 y la mejor del
--   resto no es del palo de c1, se elige la primera de la mano,
-- * Si la primera carta de la mano no es del palo de c1 y la mejor
--   del resto es del palo de c1, se elige la mejor del resto.
-- * Si la primera carta de la mano le gana a c1 y la mejor del
--   resto no le gana a c1, se elige la primera de la mano,
-- * Si la mejor del resto le gana a c1 y la primera carta de la mano
--   no le gana a c1, se elige la mejor del resto.
-- * Si el valor de la primera carta es mayor que el de la mejor del
--   resto, se elige la mejor del resto.
-- * Si el valor de la primera carta no es mayor que el de la mejor
--   del resto, se elige la primera carta.
-- ---------------------------------------------------------------------

eligeCarta :: Palo -> Carta -> Mano -> Carta
eligeCarta triunfo c1 (Agrega c Vacia) = c                       -- 1
eligeCarta triunfo c1 (Agrega c resto)
  | palo c == palo c1 && palo c' /= palo c1                 = c  -- 2
  | palo c /= palo c1 && palo c' == palo c1                 = c' -- 3
  | ganaCarta triunfo c  c1 && not (ganaCarta triunfo c' c1) = c  -- 4
  | ganaCarta triunfo c' c1 && not (ganaCarta triunfo c c1)  = c' -- 5
  | mayor (valor c) (valor c')                              = c' -- 6
  | otherwise                                               = c  -- 7
 where
  c' = eligeCarta triunfo c1 resto
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 10. Comprobar con QuickCheck que si una mano es ganadora,
-- entonces la carta elegida es ganadora.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_eligeCartaGanaSiEsPosible triunfo c m =
    m /= Vacia ==>
    ganaMano triunfo m c == ganaCarta triunfo (eligeCarta triunfo c m) c

-- La comprobación es
--    Main> quickCheck prop_eligeCartaGanaSiEsPosible
--    Falsifiable, after 12 tests:
--    Corazones
--    Carta Rey Treboles
--    Agrega (Carta (Numerico 6) Diamantes)
--           (Agrega (Carta Sota Picas)
--            (Agrega (Carta Rey Corazones)
--             (Agrega (Carta (Numerico 10) Treboles)
--              Vacia)))
-- La carta elegida es el 10 de tréboles (porque tiene que ser del mismo
-- palo), aunque el mano hay una carta (el rey de corazones) que gana.
```

Relación 22

Cálculo numérico

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

-- En esta relación se definen funciones para resolver los siguientes
-- problemas de cálculo numérico:
-- * diferenciación numérica,
-- * cálculo de la raíz cuadrada mediante el método de Herón,
-- * cálculo de los ceros de una función por el método de Newton y
-- * cálculo de funciones inversas.

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

import Test.QuickCheck

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    derivada :: Double -> (Double -> Double) -> Double -> Double
-- tal que (derivada a f x) es el valor de la derivada de la función f
-- en el punto x con aproximación a. Por ejemplo,
--    derivada 0.001 sin pi  ==  -0.9999998333332315
```

```
--    derivada 0.001 cos pi  ==  4.999999583255033e-4
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir las funciones
--    derivadaBurda :: (Double -> Double) -> Double -> Double
--    derivadaFina  :: (Double -> Double) -> Double -> Double
--    derivadaSuper :: (Double -> Double) -> Double -> Double
-- tales que
--    * (derivadaBurda f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.01,
--    * (derivadaFina f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.0001.
--    * (derivadauperBurda f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.000001.
-- Por ejemplo,
--    derivadaBurda cos pi  ==  4.999958333473664e-3
--    derivadaFina  cos pi  ==  4.999999969612645e-5
--    derivadaSuper cos pi  ==  5.000444502911705e-7
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    derivadaFinaDelSeno :: Double -> Double
-- tal que (derivadaFinaDelSeno x) es el valor de la derivada fina del
-- seno en x. Por ejemplo,
--    derivadaFinaDelSeno pi  ==  -0.9999999983354436
-- ---------------------------------------------------------------------
```

```
derivadaFinaDelSeno :: Double -> Double
derivadaFinaDelSeno = derivadaFina sin

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. En los siguientes apartados de este ejercicio se va a
-- calcular la raíz cuadrada de un número basándose en las siguientes
-- propiedades:
-- * Si y es una aproximación de la raíz cuadrada de x, entonces
--   (y+x/y)/2 es una aproximación mejor.
-- * El límite de la sucesión definida por
--       x_0     = 1
--       x_{n+1} = (x_n+x/x_n)/2
--   es la raíz cuadrada de x.
--
-- Definir, por iteración con until, la función
--    raiz :: Double -> Double
-- tal que (raiz x) es la raíz cuadrada de x calculada usando la
-- propiedad anterior con una aproximación de 0.00001 y tomando como
-- v. Por ejemplo,
--    raiz 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raiz :: Double -> Double
raiz x = raiz' 1
    where raiz' y | aceptable y = y
                  | otherwise   = raiz' (mejora y)
          mejora y    = 0.5*(y+x/y)
          aceptable y = abs(y*y-x) < 0.00001

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir el operador
--    (~=) :: Double -> Double -> Bool
-- tal que (x ~= y) si |x-y| < 0.001. Por ejemplo,
--    3.05 ~= 3.07        ==  False
--    3.00005 ~= 3.00007  == True
```

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

infix 5 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs(x-y) < 0.001

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raiz x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raiz :: Double -> Bool
prop_raiz x =
    (raiz x')^2 ~= x'
    where x' = abs x

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir por recursión la función
--    until' :: (a -> Bool) -> (a -> a) -> a -> a
-- tal que (until' p f x) es el resultado de aplicar la función f a x el
-- menor número posible de veces, hasta alcanzar un valor que satisface
-- el predicado p. Por ejemplo,
--    until' (>1000) (2*) 1  ==  1024
-- Nota: until' es equivalente a la predefinida until.
-- ---------------------------------------------------------------------

until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f x  | p x       = x
              | otherwise = until' p f (f x)

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir, por iteración con until, la función
--    raizI :: Double -> Double
-- tal que (raizI x) es la raíz cuadrada de x calculada usando la
```

```
-- propiedad anterior. Por ejemplo,
--    raizI 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raizI :: Double -> Double
raizI x = until aceptable mejora 1
        where mejora y    = 0.5*(y+x/y)
              aceptable y = abs(y*y-x) < 0.00001

-- ---------------------------------------------------------------------
-- Ejercicio 3.6. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizI x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizI :: Double -> Bool
prop_raizI x =
    (raizI x')^2 ~= x'
    where x' = abs x

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

-- ---------------------------------------------------------------------
-- Ceros de una función                                               --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los ceros de una función pueden calcularse mediante el
-- método de Newton basándose en las siguientes propiedades:
-- * Si b es una aproximación para el punto cero de f, entonces
--   b-f(b)/f'(b) es una mejor aproximación.
-- * el límite de la sucesión x_n definida por
--      x_0     = 1
--      x_{n+1} = x_n-f(x_n)/f'(x_n)
--   es un cero de f.
-- ---------------------------------------------------------------------
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por recursión la función
--    puntoCero :: (Double -> Double) -> Double
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCero cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCero' f 1
    where puntoCero' f x | aceptable x = x
                         | otherwise   = puntoCero' f (mejora x)
          mejora b    = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por iteración con until, la función
--    puntoCeroI :: (Double -> Double) -> Double
-- tal que (puntoCeroI f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCeroI cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

puntoCeroI :: (Double -> Double) -> Double
puntoCeroI f = until aceptable mejora 1
    where mejora b    = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001

-- ---------------------------------------------------------------------
-- Funciones inversas                                                 --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5. En este ejercicio se usará la función puntoCero para
-- definir la inversa de distintas funciones.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, usando puntoCero, la función
--    raizCuadrada :: Double -> Double
```

```
-- tal que (raizCuadrada x) es la raíz cuadrada de x. Por ejemplo,
--    raizCuadrada 9  ==  3.000000002941184
-- ---------------------------------------------------------------------

raizCuadrada :: Double -> Double
raizCuadrada a = puntoCero f
    where f x = x*x-a

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizCuadrada x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizCuadrada :: Double -> Bool
prop_raizCuadrada x =
    (raizCuadrada x')^2 ~= x'
    where x' = abs x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir, usando puntoCero, la función
--    raizCubica :: Double -> Double
-- tal que (raizCubica x) es la raíz cuadrada de x. Por ejemplo,
--    raizCubica 27  ==  3.0000000000196048
-- ---------------------------------------------------------------------

raizCubica :: Double -> Double
raizCubica a = puntoCero f
    where f x = x*x*x-a

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizCubica x)^3 ~= x
-- ---------------------------------------------------------------------
```

```
-- La propiedad es
prop_raizCubica :: Double -> Bool
prop_raizCubica x =
    (raizCubica x)^3 ~= x
    where x' = abs x

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.5. Definir, usando puntoCero, la función
--    arcoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcoseno 1  == 1.5665489428306574
-- ---------------------------------------------------------------------

arcoseno :: Double -> Double
arcoseno a = puntoCero f
    where f x = sin x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.6. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--    sin (arcoseno x) ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_arcoseno :: Double -> Bool
prop_arcoseno x =
    sin (arcoseno x') ~= x'
    where x' = abs (x - fromIntegral (truncate x))

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
```

```
--    arcocoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcocoseno 0  == 1.5707963267949576
-- ---------------------------------------------------------------------

arcocoseno :: Double -> Double
arcocoseno a = puntoCero f
    where f x = cos x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.8. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--    cos (arcocoseno x) ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_arcocoseno :: Double -> Bool
prop_arcocoseno x =
    cos (arcocoseno x') ~= x'
    where x' = abs (x - fromIntegral (truncate x))

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--    inversa :: (Double -> Double) -> Double -> Double
-- tal que (inversa g x) es el valor de la inversa de g en x. Por
-- ejemplo,
--    inversa (^2) 9  ==  3.000000002941184
-- ---------------------------------------------------------------------

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

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

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

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

Relación 23

Ecuación con factoriales

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

-- El objetivo de esta relación de ejercicios es resolver la ecuación
--    a! * b! = a! + b! + c!
-- donde a, b y c son números naturales.

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    factorial :: Integer -> Integer
-- tal que (factorial n) es el factorial de n. Por ejemplo,
--    factorial 5  ==  120
-- ---------------------------------------------------------------------

factorial :: Integer -> Integer
factorial n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la constante
--    factoriales :: [Integer]
```

```
-- tal que factoriales es la lista de los factoriales de los números
-- naturales. Por ejemplo,
--    take 7 factoriales  ==  [1,1,2,6,24,120,720]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, usando factoriales, la función
--    esFactorial :: Integer -> Bool
-- tal que (esFactorial n) se verifica si existe un  número natural m
-- tal que n es m!. Por ejemplo,
--    esFactorial 120  ==  True
--    esFactorial  20  ==  False
-- ---------------------------------------------------------------------

esFactorial :: Integer -> Bool
esFactorial n = n == head (dropWhile (<n) factoriales)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la constante
--    posicionesFactoriales :: [(Integer,Integer)]
-- tal que posicionesFactoriales es la lista de los factoriales con su
-- posición. Por ejemplo,
--    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
-- ---------------------------------------------------------------------
```

```
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)

-- ---------------------------------------------------------------------
-- 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
    ==> not (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

Aplicaciones de la programación funcional con listas infinitas

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

-- En esta relación se estudia distintas aplicaciones de la programación
-- funcional que usan listas infinitas
-- * definición alternativa de la sucesión de Hamming estudiada en el
--   tema 11,
-- * propiedades de la sucesión de Hamming,
-- * problemas 10 y 12 del proyecto Euler y
-- * numero de pares de naturales en un círculo.

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

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    divisoresEn :: Integer -> [Integer] -> Bool
-- tal que (divisoresEn x ys) se verifica si x puede expresarse como un
-- producto de potencias de elementos de ys. Por ejemplo,
--    divisoresEn 12 [2,3,5]  ==  True
```

```
--    divisoresEn 14 [2,3,5]  ==  False
-- ---------------------------------------------------------------------

divisoresEn :: Integer -> [Integer] -> Bool
divisoresEn 1 _                    = True
divisoresEn x []                   = False
divisoresEn x (y:ys) | mod x y == 0 = divisoresEn (div x y) (y:ys)
                     | otherwise    = divisoresEn x ys

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Los números de Hamming forman una sucesión
-- estrictamente creciente de números que cumplen las siguientes
-- condiciones:
--    1. El número 1 está en la sucesión.
--    2. Si x está en la sucesión, entonces 2x, 3x y 5x también están.
--    3. Ningún otro número está en la sucesión.
-- Definir, usando divisoresEn, la constante
--    hamming :: [Integer]
-- tal que hamming es la sucesión de Hamming. Por ejemplo,
--    take 12 hamming  ==  [1,2,3,4,5,6,8,9,10,12,15,16]
-- ---------------------------------------------------------------------

hamming :: [Integer]
hamming = [x | x <- [1..], divisoresEn x [2,3,5]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    cantidadHammingMenores :: Integer -> Int
-- tal que (cantidadHammingMenores x) es la cantidad de números de
-- Hamming menores que x. Por ejemplo,
--    cantidadHammingMenores 6  ==  5
--    cantidadHammingMenores 7  ==  6
--    cantidadHammingMenores 8  ==  6
-- ---------------------------------------------------------------------

cantidadHammingMenores :: Integer -> Int
cantidadHammingMenores x = length (takeWhile (<x) hamming)

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
```

```
--    siguienteHamming :: Integer -> Integer
-- tal que (siguienteHamming x) es el menor número de la sucesión de
-- Hamming mayor que x. Por ejemplo,
--    siguienteHamming 6  ==  8
--    siguienteHamming 21  ==  24
-- ---------------------------------------------------------------------

siguienteHamming :: Integer -> Integer
siguienteHamming x = head (dropWhile (<=x) hamming)

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Definir la función
--    huecoHamming :: Integer -> [(Integer,Integer)]
-- tal que (huecoHamming n) es la lista de pares de números consecutivos
-- en la sucesión de Hamming cuya distancia es mayor que n. Por ejemplo,
--    take 4 (huecoHamming 2)   ==  [(12,15),(20,24),(27,30),(32,36)]
--    take 3 (huecoHamming 2)   ==  [(12,15),(20,24),(27,30)]
--    take 2 (huecoHamming 3)   ==  [(20,24),(32,36)]
--    head (huecoHamming 10)    ==  (108,120)
--    head (huecoHamming 1000)  ==  (34992,36000)
-- ---------------------------------------------------------------------

huecoHamming :: Integer -> [(Integer,Integer)]
huecoHamming n = [(x,y) | x <- hamming,
                          let y = siguienteHamming x,
                          y-x > n]

-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Comprobar con QuickCheck que para todo n, existen
-- pares de números consecutivos en la sucesión de Hamming cuya
-- distancia es mayor o igual que n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_Hamming :: Integer -> Bool
prop_Hamming n = huecoHamming n' /= []
                 where n' = abs n

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

```
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. (Problema 10 del Proyecto Euler)
-- Definir la función
--    sumaPrimoMenores :: Integer -> Integer
-- tal que (sumaPrimoMenores n) es la suma de los primos menores que
-- n. Por ejemplo,
--    sumaPrimoMenores 10  ==  17
-- ---------------------------------------------------------------------

-- La definición es
sumaPrimoMenores :: Integer -> Integer
sumaPrimoMenores n = sumaMenores n primos 0
   where sumaMenores n (x:xs) a | n <= x    = a
                                | otherwise = sumaMenores n xs (a+x)

-- primos es la lista de los número primos obtenida mediante la criba de
-- Erastótenes. Por ejemplo,
--    primos  =>  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,...
primos :: [Integer]
primos = criba [2..]
         where criba (p:ps) = p : criba [n | n<-ps, mod n p /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Problema 12 del Proyecto Euler)
-- La sucesión de los números triangulares se obtiene sumando los
-- números naturales. Así, el 7º número triangular es
--    1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.
-- Los primeros 10 números triangulares son
--    1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
-- Los divisores de los primeros 7 números triangulares son:
--     1: 1
--     3: 1,3
--     6: 1,2,3,6
--    10: 1,2,5,10
--    15: 1,3,5,15
--    21: 1,3,7,21
--    28: 1,2,4,7,14,28
-- Como se puede observar, 28 es el menor número triangular con más de 5
```

```
-- divisores.
--
-- Definir la función
--    euler12 :: Int -> Integer
-- tal que (euler12 n) es el menor número triangular con más de n
-- divisores. Por ejemplo,
--    euler12 5  ==  28
-- ---------------------------------------------------------------------

euler12 :: Int -> Integer
euler12 n = head [x | x <- triangulares, nDivisores x > n]

-- triangulares es la lista de los números triangulares
--    take 10 triangulares  =>  [1,3,6,10,15,21,28,36,45,55]
triangulares :: [Integer]
triangulares = 1:[x+y | (x,y) <- zip [2..] triangulares]

-- Otra definición de triangulares es
triangulares' :: [Integer]
triangulares' = scanl (+) 1 [2..]

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--    divisores 28  ==  [1,2,4,7,14,28]
divisores :: Integer -> [Integer]
divisores x = [y | y <- [1..x], mod x y == 0]

-- (nDivisores n) es el número de los divisores de n. Por ejemplo,
--    nDivisores 28  ==  6
nDivisores :: Integer -> Int
nDivisores x = length (divisores x)

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

```
circulo :: Int -> Int
circulo n = length [(x,y) | x <- [0..n], y <- [0..n], x^2+y^2 < n^2]

-- La eficiencia puede mejorarse con
circulo' :: Int -> Int
circulo' n = length [(x,y) | x <- [0..m], y <- [0..m], x^2+y^2 < n^2]
    where m = raizCuadradaEntera n

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

Relación 25

División y factorización de polinomios mediante la regla de Ruffini

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

-- El objetivo de esta relación de ejercicios es implementar la regla de
-- Ruffini y sus aplicaciones utilizando las implementaciones del TAD de
-- polinomio estudiadas en el tema 21 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/codigos.zip
--
-- Las transparencias del tema 21 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-21.pdf

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

import PolOperaciones
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejemplos                                                           --
-- ---------------------------------------------------------------------

-- Además de los ejemplos de polinomios (ejPol1, ejPol2 y ejPol3) que se
-- encuentran en PolOperaciones, usaremos el siguiente ejemplo.
```

```
ejPol4 :: Polinomio Int
ejPol4 = consPol 3 1
                 (consPol 2 2
                          (consPol 1 (-1)
                                   (consPol 0 (-2) polCero)))

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    divisores :: Int -> [Int]
-- tal que (divisores n) es la lista de todos los divisores enteros de
-- n. Por ejemplo,
--    divisores 4     ==  [1,-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 => Int -> Polinomio a -> a
-- tal que (coeficiente k p) es el coeficiente del término de grado k en
-- p. Por ejemplo:
--     coeficiente 4 ejPol1 == 3
--     coeficiente 3 ejPol1 == 0
--     coeficiente 2 ejPol1 == -5
--     coeficiente 5 ejPol1 == 0
-- ---------------------------------------------------------------------

coeficiente :: Num a => Int -> Polinomio a -> a
coeficiente k p | k == 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 => Polinomio  a -> a
-- tal que (terminoIndep p) es el término independiente del polinomio
```

```
-- p. Por ejemplo,
--    terminoIndep ejPol1 == 3
--    terminoIndep ejPol2 == 0
--    terminoIndep ejPol4 == -2
-- ---------------------------------------------------------------------

terminoIndep :: Num a => Polinomio  a -> a
terminoIndep p = coeficiente 0 p

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    coeficientes :: Num a => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de coeficientes de p, ordenada
-- según el grado. Por ejemplo,
--     coeficientes ejPol1 == [3,0,-5,0,3]
--     coeficientes ejPol4 == [1,2,-1,-2]
--     coeficientes ejPol2 == [1,0,0,5,4,0]
-- ---------------------------------------------------------------------

coeficientes :: Num a => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <- [n,n-1..0]]
    where n = grado p

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    creaPol :: Num a => [a] -> Polinomio a
-- tal que (creaPol cs) es el polinomio cuya lista de coeficientes es
-- cs. Por ejemplo,
--     creaPol [1,0,0,5,4,0] == x^5 + 5*x^2 + 4*x
--     creaPol [1,2,0,3,0]   == x^4 + 2*x^3 + 3*x
-- ---------------------------------------------------------------------

creaPol :: Num a => [a] -> Polinomio a
creaPol []     = polCero
creaPol (a:as) = consPol n a (creaPol as)
    where n = length as

-- ---------------------------------------------------------------------
-- Ejercicio 6. Comprobar con QuickCheck que, dado un polinomio p, el
-- polinomio obtenido mediante creaPol a partir de la lista de
```

```
-- coeficientes de p coincide con p.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_coef:: Polinomio Int -> Bool
prop_coef p =
    creaPol (coeficientes p) == p

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir una función
--    pRuffini:: Int -> [Int] -> [Int]
-- tal que (pRuffini r cs) es la lista que resulta de aplicar un paso
-- del regla de Ruffini al número entero r y a la lista de coeficientes
-- cs. Por ejemplo,
--    pRuffini 2 [1,2,-1,-2] == [1,4,7,12]
--    pRuffini 1 [1,2,-1,-2] == [1,3,2,0]
-- ya que
--      | 1  2  -1  -2           | 1  2  -1  -2
--    2 |    2   8  14         1 |    1   3   2
--    --+--------------        --+-------------
--      | 1  4   7  12           | 1  3   2   0
-- ---------------------------------------------------------------------

pRuffini :: Int -> [Int] -> [Int]
pRuffini r p@(c:cs) =
    c : [x+r*y | (x,y) <- zip cs (pRuffini r p)]

-- Otra forma:
pRuffini' :: Int -> [Int] -> [Int]
pRuffini' r = scanl1 (\s x -> s * r + x)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    cocienteRuffini:: Int -> Polinomio Int -> Polinomio Int
-- tal que (cocienteRuffini r p) es el cociente de dividir el polinomio
-- p por el polinomio x-r. Por ejemplo:
```

```
--     cocienteRuffini 2 ejPol4    == x^2 + 4*x + 7
--     cocienteRuffini (-2) ejPol4 == x^2 + -1
--     cocienteRuffini 3 ejPol4    == x^2 + 5*x + 14
-- ---------------------------------------------------------------------

cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = creaPol (init (pRuffini r (coeficientes p)))

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    restoRuffini:: Int -> Polinomio Int -> Int
-- tal que (restoRuffini r p) es el resto de dividir el polinomio p por
-- el polinomio x-r. Por ejemplo,
--     restoRuffini 2 ejPol4    == 12
--     restoRuffini (-2) ejPol4 == 0
--     restoRuffini 3 ejPol4    == 40
-- ---------------------------------------------------------------------

restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (pRuffini r (coeficientes p))

-- ---------------------------------------------------------------------
-- Ejercicio 10. Comprobar con QuickCheck que, dado un polinomio p y un
-- número entero r, las funciones anteriores verifican la propiedad de
-- la división euclídea.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_diviEuclidea:: Int -> Polinomio Int -> Bool
prop_diviEuclidea r p =
    p == sumaPol (multPol coc div) res
    where coc = cocienteRuffini r p
          div = creaPol [1,-r]
          res = creaTermino 0 (restoRuffini r p)

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

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

```
-- Ejercicio 11. Definir la función
--     esRaizRuffini:: Int -> Polinomio Int -> Bool
-- tal que (esRaizRuffini r p) se verifica si r es una raiz de p, usando
-- para ello el regla de Ruffini. Por ejemplo,
--     esRaizRuffini 0 ejPol3 == True
--     esRaizRuffini 1 ejPol3 == False
-- ---------------------------------------------------------------------

esRaizRuffini:: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--     raicesRuffini :: Polinomio Int -> [Int]
-- tal que (raicesRuffini p) es la lista de las raices enteras de p,
-- calculadas usando el regla de Ruffini. Por ejemplo,
--    raicesRuffini ejPol1  == []
--    raicesRuffini ejPol2  == [0,-1]
--    raicesRuffini ejPol3  == [0]
--    raicesRuffini ejPol4  == [1,-1,-2]
--    raicesRuffini polCero == []
-- ---------------------------------------------------------------------

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
```

Relación 26

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 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/codigos.zip
-- 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.
--
-- Las transparencias del tema 21 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-21.pdf

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

import PolOperaciones
import Test.QuickCheck
import Data.Ratio

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    creaPolDispersa :: Num 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 => [a] -> Polinomio a
creaPolDispersa []     = polCero
creaPolDispersa (x:xs) = consPol (length xs) x (creaPolDispersa xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    creaPolDensa :: Num 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 => [(Int,a)] -> Polinomio a
creaPolDensa [] = polCero
creaPolDensa ((n,a):ps) = consPol n a (creaPolDensa ps)

-- ---------------------------------------------------------------------
-- Nota. En el resto de la sucesión se usará en los ejemplos los
-- los polinomios que se definen a continuación.
-- ---------------------------------------------------------------------

pol1, pol2, pol3 :: Num 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),(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 => 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 => 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]
-- ---------------------------------------------------------------------

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 => 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 => Polinomio a -> [a]
dispersa = densaAdispersa . densa

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    coeficiente :: Num 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 => 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 => Int -> Polinomio a -> a
coeficiente' k p = busca k (densa p)
    where busca k ps = head ([a | (n,a) <- ps, n == k] ++ [0])

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    coeficientes :: Num 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 => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <-[n,n-1..0]]
    where n = grado p

-- Una definición equivalente es
coeficientes' :: Num a => Polinomio a -> [a]
coeficientes' = dispersa

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    potencia :: Num 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 => Polinomio a -> Int -> Polinomio a
potencia p 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))

-- ---------------------------------------------------------------------
-- Ejercicio 9. Mejorar la definición de potencia definiendo la función
--    potenciaM :: Num 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 => Polinomio a -> Int -> Polinomio a
potenciaM p 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 => 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 => 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 => 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
-- ------------------------------------------------------------------------

integralDef :: Fractional 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 => 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 => 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 => 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 => Polinomio a -> Polinomio a -> Polinomio a
cociente p q
    | n2 == 0   = multEscalar (1/a2) p
    | n1 < n2   = polCero
    | otherwise =  consPol n' a' (cociente p' q)
    where n1 = grado p
          a1 = coefLider p
          n2 = grado q
          a2 = coefLider q
          n' = n1-n2
          a' = a1/a2
          p' = restaPol p (multPorTerm (creaTermino n' a') q)

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    resto:: Fractional 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 => Polinomio a -> Polinomio a -> Polinomio a
resto p q = restaPol p (multPol (cociente p q) q)

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    divisiblePol :: Fractional 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 => Polinomio a -> Polinomio a -> Bool
divisiblePol p q = esPolCero (resto p q)

-- ---------------------------------------------------------------------
-- 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
--   ((((a * x + b) * x + c) * x + d) * x + e) * x + f
-- y se evalúa de dentro hacia afuera.
--
-- Definir la función
--    horner:: Num 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 => Polinomio a -> a -> a
horner p x = hornerAux (coeficientes p) 0
    where hornerAux [] v     = v
          hornerAux (a:as) v = hornerAux as (a+v*x)

-- Una defininición equivalente por plegado es
horner' :: Num a => Polinomio a -> a -> a
horner' p x = (foldr (\a b -> a + b*x) 0) (coeficientes p)
```

Relación 27

Operaciones con vectores y matrices

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

-- El objetivo de esta relación es hacer ejercicios sobre vectores y
-- matrices con el tipo de tablas de las tablas, definido en el módulo
-- Data.Array y explicado en el tema 18 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-18t.pdf
-- Además, en algunos ejemplos de usan matrices con números racionales.
-- En Haskell, el número racional x/y se representa por x%y. El TAD de
-- los números racionales está definido en el módulo Data.Ratio.

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

import Data.Array
import Data.Ratio

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    numFilas :: Num a => Matriz a -> Int
-- tal que (numFilas m) es el número de filas de la matriz m. Por
```

```
-- ejemplo,
--    numFilas (listaMatriz [[1,3,5],[2,4,7]])  ==  2
-- ---------------------------------------------------------------------

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

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

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

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

dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas p)

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

separa :: Int -> [a] -> [[a]]
separa _ [] = []
```

```
separa n xs = take n xs : separa n (drop n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    matrizLista :: Num a => Matriz a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
--    ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> m
--    array ((1,1),(2,3)) [((1,1),5),((1,2),1),((1,3),0),
--                         ((2,1),3),((2,2),2),((2,3),6)]
--    ghci> matrizLista m
--    [[5,1,0],[3,2,6]]
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    sumaMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
```

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    prodMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
-- tal que (prodMatrices p q) es el producto de las matrices p y q. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,1],[2,4]]
--    ghci> prodMatrices p p
--    array ((1,1),(2,2)) [((1,1),11),((1,2),7),((2,1),14),((2,2),18)]
--    ghci> matrizLista (prodMatrices p p)
--    [[11,7],[14,18]]
--    ghci> let q = listaMatriz [[7],[5]]
--    ghci> prodMatrices p q
--    array ((1,1),(2,1)) [((1,1),26),((2,1),34)]
--    ghci> matrizLista (prodMatrices p q)
--    [[26],[34]]
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Traspuestas y simétricas                                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    traspuesta :: Num a => Matriz a -> Matriz a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> traspuesta p
--    array ((1,1),(3,2)) [((1,1),5),((1,2),3),
--                         ((2,1),1),((2,2),2),
--                         ((3,1),0),((3,2),6)]
--    ghci> matrizLista (traspuesta p)
--    [[5,3],[1,2],[0,6]]
-- ---------------------------------------------------------------------

traspuesta :: Num a => Matriz a -> Matriz a
traspuesta p =
    array ((1,1),(n,m))
          [((i,j), p!(j,i)) | i <- [1..n], j <- [1..m]]
    where (m,n) = dimension p

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    esCuadrada :: Num a => Matriz a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> esCuadrada p
--    False
```

```
--    ghci> let q = listaMatriz [[5,1],[3,2]]
--    ghci> esCuadrada q
--    True
-- ---------------------------------------------------------------------

esCuadrada :: Num a => Matriz a -> Bool
esCuadrada x = numFilas x == numColumnas x

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    esSimetrica :: Num a => Matriz a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
--    ghci> esSimetrica p
--    True
--    ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
--    ghci> esSimetrica q
--    False
-- ---------------------------------------------------------------------

esSimetrica :: Num a => Matriz a -> Bool
esSimetrica x = x == traspuesta x

-- ---------------------------------------------------------------------
-- Diagonales de una matriz                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    diagonalPral :: Num a => Matriz a -> Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalPral p
--    array (1,2) [(1,5),(2,2)]
--    ghci> vectorLista (diagonalPral p)
--    [5,2]
-- ---------------------------------------------------------------------
```

```
diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    diagonalSec :: Num a => Matriz a -> Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalSec p
--    array (1,2) [(1,1),(2,3)]
--    ghci> vectorLista (diagonalPral p)
--    [5,2]
-- ---------------------------------------------------------------------

diagonalSec :: Num a => Matriz a -> Vector a
diagonalSec p = array (1,n) [(i,p!(i,m+1-i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)
          m = numFilas p

-- ---------------------------------------------------------------------
-- Submatrices                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> submatriz 2 3 p
--    array ((1,1),(2,2)) [((1,1),5),((1,2),1),((2,1),4),((2,2),6)]
--    ghci> matrizLista (submatriz 2 3 p)
--    [[5,1],[4,6]]
-- ---------------------------------------------------------------------

submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
    array ((1,1), (m-1,n -1))
```

```
          [((k,l), p ! f k l) | k <- [1..m-1], l <- [1.. n-1]]
    where (m,n) = dimension p
          f k l | k < i  && l < j  = (k,l)
                | k >= i && l < j  = (k+1,l)
                | k < i  && l >= j = (k,l+1)
                | otherwise        = (k+1,l+1)

-- ---------------------------------------------------------------------
-- Transformaciones elementales                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> intercambiaFilas 1 3 p
--    array ((1,1),(3,3)) [((1,1),4),((1,2),6),((1,3),9),
--                         ((2,1),3),((2,2),2),((2,3),6),
--                         ((3,1),5),((3,2),1),((3,3),0)]
--    ghci> matrizLista (intercambiaFilas 1 3 p)
--    [[4,6,9],[3,2,6],[5,1,0]]
-- ---------------------------------------------------------------------

intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaFilas k l p =
    array ((1,1), (m,n))
          [((i,j), p! f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = (l,j)
                | i == l    = (k,j)
                | otherwise = (i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la función
--    intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaColumnas k l p) es la matriz obtenida
-- intercambiando las columnas k y l de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
```

```
--    ghci> matrizLista (intercambiaColumnas 1 3 p)
--    [[0,1,5],[6,2,3],[9,6,4]]
-- ---------------------------------------------------------------------

intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaColumnas k l p =
    array ((1,1), (m,n))
          [((i,j), p ! f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | j == k    = (i,l)
                | j == l    = (i,k)
                | otherwise = (i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir la función
--    multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
-- tal que (multFilaPor k x p) es a matriz obtenida multiplicando la
-- fila k de la matriz p por el número x. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (multFilaPor 2 3 p)
--    [[5,1,0],[9,6,18],[4,6,9]]
-- ---------------------------------------------------------------------

multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
multFilaPor k x p =
    array ((1,1), (m,n))
          [((i,j), f i j)  | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = x*(p!(i,j))
                | otherwise = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 23. Definir la función
--    sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
-- a la fila k d la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (sumaFilaFila 2 3 p)
--    [[5,1,0],[7,8,15],[4,6,9]]
-- ---------------------------------------------------------------------
```

```
sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
sumaFilaFila k l p =
    array ((1,1), (m,n))
          [((i,j), f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = p!(i,j) + p!(l,j)
                | otherwise = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 24. Definir la función
--    sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (sumaFilaPor 2 3 10 p)
--    [[5,1,0],[43,62,96],[4,6,9]]
-- ---------------------------------------------------------------------

sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
sumaFilaPor k l x p =
    array ((1,1), (m,n))
          [((i,j), f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = p!(i,j) + x*p!(l,j)
                | otherwise = p!(i,j)

-- ---------------------------------------------------------------------
-- Triangularización de matrices                                      --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 25. Definir la función
--    buscaIndiceDesde :: Num a => Matriz a -> Int -> Int -> Maybe Int
-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaIndiceDesde p 3 2
--    Just 2
```

```
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaIndiceDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

buscaIndiceDesde :: Num a => Matriz a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
    | null xs   = Nothing
    | otherwise = Just (head xs)
    where xs = [k | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]

-- ---------------------------------------------------------------------
-- Ejercicio 26. Definir la función
--    buscaPivoteDesde :: Num a => Matriz a -> Int -> Int -> Maybe a
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaPivoteDesde p 3 2
--    Just 6
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaPivoteDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

buscaPivoteDesde :: Num a => Matriz a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
    | null xs   = Nothing
    | otherwise = Just (head xs)
    where xs = [y | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]

-- ---------------------------------------------------------------------
-- Ejercicio 27. Definir la función
--    anuladaColumnaDesde :: Num a => Int -> Int -> Matriz a -> Bool
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> anuladaColumnaDesde q 3 2
--    True
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
```

```
--    ghci> anuladaColumnaDesde p 3 2
--    False
-- ---------------------------------------------------------------------

anuladaColumnaDesde :: Num a => Matriz a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
    buscaIndiceDesde p j (i+1) == Nothing

-- ---------------------------------------------------------------------
-- Ejercicio 28. Definir la función
--    anulaEltoColumnaDesde :: Fractional a =>
--                             Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
--    ghci> let p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matriz Double
--    ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
--    [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]
-- ---------------------------------------------------------------------

anulaEltoColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
anulaEltoColumnaDesde p j i =
    sumaFilaPor l i (-(p!(l,j)/a)) p
    where Just l = buscaIndiceDesde p j (i+1)
          a      = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 29. Definir la función
--    anulaColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando
-- todos los elementos de la columna j de la matriz p por debajo del la
-- posición (i,j) (se supone que el elemnto p_(i,j) es no nulo). Por
-- ejemplo,
--    ghci> let p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matriz Double
--    ghci> matrizLista (anulaColumnaDesde p 2 1)
--    [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
--    ghci> let p = listaMatriz [[4,5],[2,7%2],[6,10]]
--    ghci> matrizLista (anulaColumnaDesde p 1 1)
--    [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
-- ---------------------------------------------------------------------
```

```
anulaColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
anulaColumnaDesde p j i
    | anuladaColumnaDesde p j i = p
    | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i

-- ---------------------------------------------------------------------
-- Algoritmo de Gauss para triangularizar matrices                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 30. Definir la función
--    elementosNoNulosColDesde :: Num a => Matriz a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
--    ghci> let p = listaMatriz [[3,2],[5,1],[0,4]]
--    ghci> elementosNoNulosColDesde p 1 2
--    [5]
-- ---------------------------------------------------------------------

elementosNoNulosColDesde :: Num a => Matriz a -> Int -> Int -> [a]
elementosNoNulosColDesde p j i =
    [x | ((k,j'),x) <- assocs p, x /= 0, j' == j, k >= i]

-- ---------------------------------------------------------------------
-- Ejercicio 31. Definir la función
--    existeColNoNulaDesde :: Num a => Matriz a -> Int -> Int -> Bool
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la j; es decir, si la submatriz de p obtenida eliminando
-- las i-1 primeras filas y las j-1 primeras columnas es no nula. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> existeColNoNulaDesde p 2 2
--    False
--    ghci> let q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> existeColNoNulaDesde q 2 2
-- ---------------------------------------------------------------------

existeColNoNulaDesde :: Num a => Matriz a -> Int -> Int -> Bool
```

```
existeColNoNulaDesde p j i =
    or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
    where n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 32. Definir la función
--    menorIndiceColNoNulaDesde
--      :: Num a => Matriz a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
--    ghci> let p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde p 2 2
--    Just 2
--    ghci> let q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
--    ghci> menorIndiceColNoNulaDesde q 2 2
--    Just 3
--    ghci> let r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde r 2 2
--    Nothing
-- ---------------------------------------------------------------------

menorIndiceColNoNulaDesde :: (Num a) => Matriz a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
    | null js   = Nothing
    | otherwise = Just (head js)
    where n  = numColumnas p
          js = [j' | j' <- [j..n],
                     not (null (elementosNoNulosColDesde p j' i))]

-- ---------------------------------------------------------------------
-- Ejercicio 33. Definir la función
--    gaussAux :: Fractional a => Matriz a -> Int -> Int -> Matriz a
-- tal que (gauss p) es la matriz que en el que las i-1 primeras filas y
-- las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
--    1. Si la dimensión de p es (i,j), entonces p.
--    2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--       primeras columnas es nulas, entonces p.
--    3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
```

```
--    3.1. j' la primera columna a partir de la j donde p tiene
--         algún elemento no nulo a partir de la fila i,
--    3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--         de p,
--    3.3. i' la primera fila a partir de la i donde la columna j de
--         p1 tiene un elemento no nulo,
--    3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--         la matriz p1 y
--    3.5. p' la matriz obtenida anulando todos los elementos de la
--         columna j de p2 por debajo de la fila i.
-- Por ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
--    ghci> matrizLista (gaussAux p 2 2)
--    [[1.0,2.0,3.0],[1.0,2.0,4.0],[2.0,0.0,1.0]]
-- ---------------------------------------------------------------------

gaussAux :: Fractional a => Matriz a -> Int -> Int -> Matriz a
gaussAux p i j
    | dimension p == (i,j)            = p                         -- 1
    | not (existeColNoNulaDesde p j i) = p                         -- 2
    | otherwise                        = gaussAux p' (i+1) (j+1)  -- 3
    where Just j' = menorIndiceColNoNulaDesde p j i               -- 3.1
          p1      = intercambiaColumnas j j' p                     -- 3.2
          Just i' = buscaIndiceDesde p1 j i                        -- 3.3
          p2      = intercambiaFilas i i' p1                       -- 3.4
          p'      = anulaColumnaDesde p2 j i                       -- 3.5

-- ---------------------------------------------------------------------
-- Ejercicio 34. Definir la función
--    gauss :: Fractional a => Matriz a -> Matriz a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--    ghci> gauss p
--    array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                         ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                         ((3,1),0.0),((3,2),0.0),((3,3),0.0)]
--    ghci> matrizLista (gauss p)
--    [[1.0,3.0,2.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
--    ghci> let p = listaMatriz [[3.0,2,3],[1,2,4],[1,2,5]]
```

```
--    ghci> matrizLista (gauss p)
--    [[3.0,2.0,3.0],[0.0,1.3333333333333335,3.0],[0.0,0.0,1.0]]
--    ghci> let p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
--    ghci> matrizLista (gauss p)
--    [[3 % 1,2 % 1,3 % 1],[0 % 1,4 % 3,3 % 1],[0 % 1,0 % 1,1 % 1]]
--    ghci> let p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
--    ghci> matrizLista (gauss p)
--    [[1.0,3.0,0.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
-- ---------------------------------------------------------------------

gauss :: Fractional a => Matriz a -> Matriz a
gauss p = gaussAux p 1 1

-- ---------------------------------------------------------------------
-- Determinante                                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 35. Definir la función
--    determinante :: Fractional a => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--    ghci> determinante p
--    0.0
--    ghci> let p = listaMatriz [[1.0,2,3],[1,3,4],[1,2,5]]
--    ghci> determinante p
--    2.0
-- ---------------------------------------------------------------------

determinante :: Fractional a => Matriz a -> a
determinante p = product (elems (diagonalPral (gauss p)))
```

Relación 28

Ejercicios complementarios

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

-- En esta relación se recogen ejercicios variados de programación
-- funcional que complementan las relaciones anteriores.
--
-- Los problemas incluidos son:
-- * el de los cuadrados mágicos,
-- * el de la codificación de mensajes,
-- * el de pertenencia al rango de una función creciente,
-- * el de los puntos cercanos,
-- * el de la evaluación de expresiones aritméticas y
-- * el del menor número con todos los dígitos en la factorización de su
--   factorial.

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

import Data.List
import Data.Char

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una matriz cuadrada representa un cuadrado mágico de
-- orden n el conjunto de sus elementos es {1,2,...,n^2}, las sumas de
-- cada una de sus filas, columnas y dos diagonales principales
```

```
-- coinciden. Por ejemplo,
--    / 2  9  4 \
--    | 7  5  3 |
--    \ 6  1  8 /
-- es un cuadrado mágico de orden 3, ya que el conjunto de sus
-- elementos es {1,2,...,9}, todas sus filas, columnas y diagonales
-- principales suman 15.
--
-- Representaremos una matriz numérica como una lista cuyos elementos son
-- las filas del cuadrado, en forma de listas. Por ejemplo, el cuadrado
-- anterior vendría representado por la siguiente lista:
--    [[2, 9, 4], [7, 5, 3], [6, 1, 8]]
-- En los distintos apartados de este ejercicio se definirán funciones
-- cuyo objetivo es decidir si una matriz representa un cuadrado
-- mágico y construirlos.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    traspuesta :: [[a]] -> [[a]]
-- tal que (traspuesta m) es la traspuesta de la matriz m. Por ejemplo,
--    traspuesta [[1,2,3],[4,5,6]]    ==  [[1,4],[2,5],[3,6]]
--    traspuesta [[1,4],[2,5],[3,6]]  ==  [[1,2,3],[4,5,6]]
-- ---------------------------------------------------------------------

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

-- Una definición equivalente es
traspuesta' :: [[a]] -> [[a]]
traspuesta' = transpose

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    sumas_de_filas :: Num a => [[a]] -> [a]
-- tal que (sumas_de_filas xss) es la lista de las sumas de las filas de
-- la matriz xss. Por ejemplo,
```

```haskell
--    sumas_de_filas [[2,4,0],[7,1,3],[6,1,8]]  ==  [6,11,15]
-- ---------------------------------------------------------------------

sumas_de_filas :: Num a => [[a]] -> [a]
sumas_de_filas = map sum

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    sumas_de_columnas :: Num a => [[a]] -> [a]
-- tal que (sumas_de_columnas xss) es la lista de las sumas de las
-- columnas de la matriz xss. Por ejemplo,
--    sumas_de_filas [[2,4,0],[7,1,3],[6,1,8]]  ==  [6,11,15]
-- ---------------------------------------------------------------------

sumas_de_columnas :: Num a => [[a]] -> [a]
sumas_de_columnas = sumas_de_filas . traspuesta

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
--    diagonal_pral :: [[a]] -> [a]
-- tal que (diagonal_pral m) es la diagonal principal de la matriz m. Por
-- ejemplo,
--    diagonal_pral [[3,5,2],[4,7,1],[6,9,0]]  ==  [3,7,0]
--    diagonal_pral [[3,5,2],[4,7,1]]          ==  [3,7]
-- ---------------------------------------------------------------------

diagonal_pral :: [[a]] -> [a]
diagonal_pral ((x1:_):xs) = x1 : diagonal_pral [tail x | x <- xs]
diagonal_pral _ = []

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Definir la función
--    diagonal_sec :: [[a]] -> [a]
-- tal que (diagonal_sec m) es la diagonal secundaria de la matriz m
-- Por ejemplo,
--    diagonal_pral [[3,5,2],[4,7,1],[6,9,0]]  ==  [6,7,2]
--    diagonal_pral [[3,5,2],[4,7,1]]          ==  [4,5]
-- ---------------------------------------------------------------------

diagonal_sec :: [[a]] -> [a]
```

```
diagonal_sec = diagonal_pral . reverse

-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Definir la función
--    todos_iguales :: Eq a => [a] -> Bool
-- tal que (todos_iguales xs) se verifica si todos los eleentos de xs
-- son iguales. Por ejemplo,
--    todos_iguales [2,2,2]  ==  True
--    todos_iguales [2,3,2]  ==  False
-- ---------------------------------------------------------------------

todos_iguales :: Eq a => [a] -> Bool
todos_iguales (x:y:ys) = x == y && todos_iguales (y:ys)
todos_iguales _ = True

-- ---------------------------------------------------------------------
-- Ejercicio 1.7. Definir la función
--    matrizCuadrada :: [[Int]] -> Bool
-- tal que (matrizCuadrada xss) se verifica si xss es una matriz
-- cuadrada; es decir, xss es una lista de n elementos y cada elemento
-- de xss es una lista de n elementos. Por ejemplo,
--    matrizCuadrada [[7,3],[1,5]]      ==  True
--    matrizCuadrada [[7,3,1],[1,5,2]]  ==  False
-- ---------------------------------------------------------------------

matrizCuadrada :: [[Int]] -> Bool
matrizCuadrada xss =
    and [length xs == n | xs <- xss]
    where n = length xss

-- ---------------------------------------------------------------------
-- Ejercicio 1.9. Definir la función
--    elementos :: [[a]] -> [a]
-- tal que (elementos xss) es la lista de los elementos de xss. Por
-- ejemplo,
--    elementos [[7,3],[1,5],[3,5]]  ==  [7,3,1,5,3,5]
-- ---------------------------------------------------------------------

elementos :: [[a]] -> [a]
elementos = concat
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 1.10. Definir por recursión la función
--    borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando la primera
-- ocurrencia de x en la lista xs. Por ejemplo,
--    borra 1 [1,2,1]  ==  [2,1]
--    borra 3 [1,2,1]  ==  [1,2,1]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.11. Definir por recursión la función
--    esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutación de
-- ys. Por ejemplo,
--    esPermutacion [1,2,1] [2,1,1]  ==  True
--    esPermutacion [1,2,1] [1,2,2]  ==  False
-- ---------------------------------------------------------------------

esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     []     = True
esPermutacion []     (y:ys) = False
esPermutacion (x:xs) ys     = elem x ys && esPermutacion xs (borra x ys)

-- ---------------------------------------------------------------------
-- Ejercicio 1.12. Definir la función
--    cuadradoMagico :: Num a => [[a]] -> Bool
-- tal que (cuadradoMagico xss) se verifica si xss es un cuadrado
-- mágico. Por ejemplo,
--    ghci> cuadradoMagico [[2,9,4],[7,5,3],[6,1,8]]
--    True
--    ghci> cuadradoMagico [[1,2,3],[4,5,6],[7,8,9]]
--    False
--    ghci> cuadradoMagico [[1,1],[1,1]]
--    False
```

```
--    ghci> cuadradoMagico [[5,8,12,9],[16,13,1,4],[2,10,7,15],[11,3,14,6]]
--    False
-- ---------------------------------------------------------------------

cuadradoMagico xss =
    matrizCuadrada xss &&
    esPermutacion (elementos xss) [1..(length xss)^2]  &&
    todos_iguales ((sumas_de_filas xss )++
                   (sumas_de_columnas xss) ++
                   [(sum (diagonal_pral xss)),
                    (sum (diagonal_sec xss))])

-- ---------------------------------------------------------------------
-- Ejercicio 1.13. Definir la función
--    matriz :: Int-> [a] -> [[a]]
-- tal que (matriz n xs) es la matriz cuadrada de orden nxn cuyos
-- elementos son xs (se supone que la longitud de xs es n^2). Por
-- ejemplo,
--    matriz 3 [1..9]  ==  [[1,2,3],[4,5,6],[7,8,9]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.14. Definir la función
--    cuadradosMagicos :: Int -> [[[Int]]]
-- tal que (cuadradosMagicos n) es la lista de los cuadrados mágicos de
-- orden nxn. Por ejemplo,
--    ghci> take 2 (cuadradosMagicos 3)
--    [[[2,9,4],[7,5,3],[6,1,8]], [[2,7,6],[9,5,1],[4,3,8]]]
-- ---------------------------------------------------------------------

cuadradosMagicos :: Int -> [[[Int]]]
cuadradosMagicos n =
    [m | xs <- permutations [1..n^2],
         let m = matriz n xs,
         cuadradoMagico m]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 1.15. Los cuadrados mágicos de orden 3 tienen la forma
--    +---+---+---+
--    | a | b | c |
--    +---+---+---+
--    | d | e | f |
--    +---+---+---+
--    | g | h | i |
--    +---+---+---+
-- y se pueden construir como sigue:
--    * a es un elemento de [1..9],
--    * b es un elemento de los restantes (es decir, de [1..9] \\ [a]),
--    * c es un elemento de los restantes,
--    * a+b+c tiene que ser igual a 15,
--    * d es un elemento de los restantes,
--    * g es un elemento de los restantes,
--    * a+d+g tiene que ser igual a 15,
-- y así sucesivamente.
--
-- Definir la función
--    cuadradosMagicos_3 :: [[[Int]]]
-- tal que cuadradosMagicos_3  es la lista de los cuadrados mágicos de
-- orden 3 construidos usando el proceso anterior. Por ejemplo,
--    ghci> take 2 cuadradosMagicos_3
--    [[[2,7,6],[9,5,1],[4,3,8]],[[2,9,4],[7,5,3],[6,1,8]]]
-- ---------------------------------------------------------------------

cuadradosMagicos_3 :: [[[Int]]]
cuadradosMagicos_3 =
    [[[a,b,c],[d,e,f],[g,h,i]] |
     a <- [1..9],
     b <- [1..9] \\ [a],
     c <- [1..9] \\ [a,b],
     a+b+c == 15,
     d <- [1..9] \\ [a,b,c],
     g <- [1..9] \\ [a,b,c,d],
     a+d+g == 15,
     e <- [1..9] \\ [a,b,c,d,g],
     c+e+g == 15,
     i <- [1..9] \\ [a,b,c,d,g,e],
```

```
     a+e+i == 15,
     f <- [1..9] \\ [a,b,c,d,g,e,i],
     h <- [1..9] \\ [a,b,c,d,g,e,i,f],
     c+f+i == 15,
     d+e+f == 15]

-- ---------------------------------------------------------------------
-- Ejercicio 1.16. Comprobar que cuadradosMagicos_3 es el mismo conjunto
-- que (cuadradosMagicos 3).
-- ---------------------------------------------------------------------

-- La comprobación es
--    ghci> esPermutacion cuadradosMagicos_3 (cuadradosMagicos 3)
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1.17. Comparar los tiempos utilizados en calcular
-- cuadradosMagicos_3 y (cuadradosMagicos 3).
-- ---------------------------------------------------------------------

-- La comparación es
--    ghci> :set +s
--    ghci> cuadradosMagicos_3
--    [[[2,7,6],[9,5,1],[4,3,8]],[[2,9,4],[7,5,3],[6,1,8]], ...
--    (0.02 secs, 532348 bytes)
--    ghci> (cuadradosMagicos 3)
--    [[[2,9,4],[7,5,3],[6,1,8]],[[2,7,6],[9,5,1],[4,3,8]], ...
--    (50.32 secs, 2616351124 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se desea definir una función que codifique mensajes
-- tales como
--        "eres lo que piensas"
-- del siguiente modo:
-- (a) se separa la cadena en la lista de sus palabras:
--        ["eres","lo","que","piensas"]
-- (b) se cuenta las letras de cada palabra:
--        [4,2,3,7]
-- (c) se une todas las palabras:
--        "eresloquepiensas"
```

```
-- (d) se reagrupa las letras de 4 en 4, dejando el último grupo con el
--     resto:
--        ["eres","loqu","epie","nsas"]
-- (e) se inverte cada palabra:
--        ["sere","uqol","eipe","sasn"]
-- (f) se une todas las palabras:
--        "sereuqoleipesasn"
-- (g) se reagrupan tal como indica la inversa de la lista del apartado
--     (b):
--        ["sereuqo","lei","pe","sasn"]
-- (h) se crea una frase con las palabras anteriores separadas por un
--     espacio en blanco
--        "sereuqo lei pe sasn"
--     obteniendo así el mensaje codificado.
--
-- En los distintos apartados de este ejercicio se definirá el anterior
-- proceso de codificación.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    divide :: (a -> Bool) -> [a] -> ([a], [a])
-- tal que (divide p xs) es el par (ys,zs) donde ys es el mayor prefijo
-- de xs cuyos elementos cumplen p y zs es la lista de los restantes
-- elementos de xs. Por ejemplo,
--    divide (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
--    divide (< 9) [1,2,3]           == ([1,2,3],[])
--    divide (< 0) [1,2,3]           == ([],[1,2,3])
-- ---------------------------------------------------------------------

divide :: (a -> Bool) -> [a] -> ([a], [a])
divide p xs = (takeWhile p xs, dropWhile p xs)

-- Es equivalente a la predefinida span
divide' :: (a -> Bool) -> [a] -> ([a], [a])
divide' = span

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    palabras :: String -> [String]
```

```
-- tal que (palabras cs) es la lista de las palabras de la cadena cs.
-- Por ejemplo,
--    palabras "eres lo que piensas"  ==  ["eres","lo","que","piensas"]
-- ---------------------------------------------------------------------

palabras :: String -> [String]
palabras [] = []
palabras cs = cs1 : palabras cs2
    where cs' = dropWhile (==' ') cs
          (cs1,cs2) = divide (/=' ') cs'

-- Es equivalente a la predefinida words
palabras' :: String -> [String]
palabras' = words

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función
--    longitudes :: [[a]] -> [Int]
-- tal que (longitudes xss) es la lista de las longitudes de los
-- elementos xss. Por ejemplo,
--    longitudes ["eres","lo","que","piensas"]  ==  [4,2,3,7]
-- ---------------------------------------------------------------------

longitudes :: [[a]] -> [Int]
longitudes = map length

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Definir la función
--    une :: [[a]] -> [a]
-- tal que (une xss) es la lista obtenida uniendo los elementos de
-- xss. Por ejemplo,
--    une ["eres","lo","que","piensas"]  ==  "eresloquepiensas"
-- ---------------------------------------------------------------------

une :: [[a]] -> [a]
une = concat

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Definir la función
--    reagrupa :: [a] -> [[a]]
```

```haskell
-- tal que (reagrupa xs) es la lista obtenida agrupando los elementos de
-- xs de 4 en 4. Por ejemplo,
--    reagrupa "eresloquepiensas"  ==  ["eres","loqu","epie","nsas"]
--    reagrupa "erestu"            ==  ["eres","tu"]
-- ---------------------------------------------------------------------

reagrupa :: [a] -> [[a]]
reagrupa [] = []
reagrupa xs = take 4 xs : reagrupa (drop 4 xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2.6. Definir la función
--    inversas :: [[a]] -> [[a]]
-- tal que (inversas xss) es la lista obtenida invirtiendo los elementos
-- de xss. Por ejemplo,
--    ghci> inversas ["eres","loqu","epie","nsas"]
--    ["sere","uqol","eipe","sasn"]
--    ghci> une (inversas ["eres","loqu","epie","nsas"])
--    "sereuqoleipesasn"
-- ---------------------------------------------------------------------

inversas :: [[a]] -> [[a]]
inversas = map reverse

-- ---------------------------------------------------------------------
-- Ejercicio 2.7. Definir la función
--    agrupa :: [a] -> [Int] -> [[a]]
-- tal que (agrupa xs ns) es la lista obtenida agrupando los elementos
-- de xs según las longitudes indicadas en ns. Por ejemplo,
--    ghci> agrupa "sereuqoleipesasn" [7,3,2,4]
--    ["sereuqo","lei","pe","sasn"]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.8. Definir la función
--    frase :: [String] -> String
```

```
-- tal que (frase xs) es la frase obtenida las palabras de xs dejando un
-- espacio en blanco entre ellas. Por ejemplo,
--    frase ["sereuqo","lei","pe","sasn"]  ==  "sereuqo lei pe sasn"
-- ---------------------------------------------------------------------

frase :: [String] -> String
frase [x]    = x
frase (x:xs) = x ++ " " ++ frase xs
frase []     = []

-- La función frase es equivalente a unwords.
frase' :: [String] -> String
frase' = unwords

-- ---------------------------------------------------------------------
-- Ejercicio 2.9. Definir la función
--    clave :: String -> String
-- que realice el proceso completo. Por ejemplo,
--    clave "eres lo que piensas"  ==  "sereuqo lei pe sasn"
-- ---------------------------------------------------------------------

clave :: String -> String
clave xss = frase (agrupa (une (inversas (reagrupa (une ps))))
                          (reverse (longitudes ps)))
    where ps = palabras xss

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    perteneceRango:: Int -> (Int -> Int) -> Bool
-- tal que (perteneceRango x f) se verifica si x pertenece al rango de
-- la función f, suponiendo que f es una función creciente cuyo dominio
-- es el conjunto de los números naturales. Por ejemplo,
--    perteneceRango 5 (\x -> 2*x+1)     ==  True
--    perteneceRango 1234 (\x -> 2*x+1)  ==  False
-- ---------------------------------------------------------------------

perteneceRango:: Int -> (Int -> Int) -> Bool
perteneceRango y f = elem y (takeWhile (<=y) (imagenes f))
    where imagenes f = [f x | x <- [0..]]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. Los puntos del plano se pueden representar por pares de
-- números como se indica a continuación
--    type Punto = (Double,Double)
-- Definir la función
--    cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
-- tal que (cercanos ps qs) es un par de puntos, el primero de ps y el
-- segundo de qs, que son los más cercanos (es decir, no hay otro par
-- (p',q') con p' en ps y q' en qs tales que la distancia entre p' y q'
-- sea menor que la que hay entre p y q). Por ejemplo,
--    cercanos [(2,5),(3,6)] [(4,3),(1,0),(7,9)] == ((2.0,5.0),(4.0,3.0))
-- ---------------------------------------------------------------------

type Punto = (Double,Double)

cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
cercanos ps qs = (p,q)
    where (d,p,q) = minimum [(distancia p q, p, q) | p <- ps, q <-qs]
          distancia (x,y) (u,v) = sqrt ((x-u)^2+(y-v)^2)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Las expresiones aritméticas pueden representarse usando
-- el siguiente tipo de datos
--    data Expr = N Int | V Char | S Expr Expr | P Expr Expr
--              deriving Show
-- Por ejemplo, la expresión 2*(a+5) se representa por
--    P (N 2) (S (V 'a') (N 5))
-- Definir la función
--    valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- deir, el valor de la expresión donde las variables de x se sustituyen
-- por los valores según se indican en el entorno e). Por ejemplo,
--    ghci> valor (P (N 2) (S (V 'a') (V 'b'))) [('a',2),('b',5)]
--    14
-- ---------------------------------------------------------------------

data Expr = N Int | V Char | S Expr Expr | P Expr Expr
          deriving Show

valor :: Expr -> [(Char,Int)] -> Int
```

```
valor (N x)   e = x
valor (V x)   e = head [y | (z,y) <- e, z == x]
valor (S x y) e = (valor x e) + (valor y e)
valor (P x y) e = (valor x e) * (valor y e)

-- ---------------------------------------------------------------------
-- Ejercicio 6. El enunciado del problema 652 de "Números y algo más" es
-- el siguiente:
--    Si factorizamos los factoriales de un número en función de sus
--    divisores primos y sus potencias, ¿cuál es el menor número N tal
--    que entre los factores primos y los exponentes de la factorización
--    de N! están todos los dígitos del cero al nueve?
--    Por ejemplo
--        6! = 2^4*3^2*5^1, le faltan los dígitos 0,6,7,8 y 9
--       12! = 2^10*3^5*5^2*7^1*11^1, le faltan los dígitos 4,6,8 y 9
-- Definir la función
--    digitosDeFactorizacion :: Integer -> [Integer]
-- tal que (digitosDeFactorizacion n) es el conjunto de los dígitos que
-- aparecen en la factorización de n. Por ejemplo,
--    digitosDeFactorizacion (factorial 6)   ==  [1,2,3,4,5]
--    digitosDeFactorizacion (factorial 12)  ==  [0,1,2,3,5,7]
-- Usando la función anterior, calcular la solución del problema.
-- ---------------------------------------------------------------------

digitosDeFactorizacion :: Integer -> [Integer]
digitosDeFactorizacion n =
   sort (nub (concat [digitos x | x <- numerosDeFactorizacion n]))

-- (digitos n) es la lista de los digitos del número n. Por ejemplo,
--    digitos 320274  ==  [3,2,0,2,7,4]
digitos :: Integer -> [Integer]
digitos n = [read [x] | x <- show n]

-- (numerosDeFactorizacion n) es el conjunto de los números en la
-- factorización de n. Por ejemplo,
--    numerosDeFactorizacion 60  ==  [1,2,3,5]
numerosDeFactorizacion :: Integer -> [Integer]
numerosDeFactorizacion n =
   sort (nub (aux (factorizacion n)))
   where aux [] = []
```

```
         aux ((x,y):zs) = x : y : aux zs

-- (factorización n) es la factorización de n. Por ejemplo,
--    factorizacion 300  ==  [(2,2),(3,1),(5,2)]
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n =
    [(head xs, fromIntegral (length xs)) | xs <- group (factorizacion' n)]

-- (factorizacion' n) es la lista de todos los factores primos de n; es
-- decir, es una lista de números primos cuyo producto es n. Por ejemplo,
--    factorizacion 300  ==  [2,2,3,5,5]
factorizacion' :: Integer -> [Integer]
factorizacion' n | n == 1    = []
                 | otherwise = x : factorizacion' (div n x)
                 where x = menorFactor n

-- (menorFactor n) es el menor factor primo de n. Por ejemplo,
--    menorFactor 15  ==  3
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 5  ==  120
factorial :: Integer -> Integer
factorial n = product [1..n]

-- Para calcular la solución, se define la constante
solucion =
    head [n | n <- [1..], digitosDeFactorizacion (factorial n) == [0..9]]

-- El cálculo de la solución es
--    ghci> solucion
--    49
```

Relación 29

Relaciones binarias

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

-- El objetivo de esta relación de ejercicios es definir propiedades y
-- operaciones sobre las relaciones binarias (homogéneas).
--
-- Como referencia se puede usar el artículo de la wikipedia
-- http://bit.ly/HVHOPS

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

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una relación binaria R sobre un conjunto A puede
-- representar mediante un par (xs,ps) donde xs es la lista de los
-- elementos de A (el universo de R) y ps es la lista de pares de R (el
-- grafo de R). Definir el tipo de dato (Rel a) para representar las
-- relaciones binarias sobre a.
-- ---------------------------------------------------------------------

type Rel a = ([a],[(a,a)])
```

```
-- ---------------------------------------------------------------------
-- Nota. En los ejemplos usaremos las siguientes relaciones binarias:
--    r1, r2, r3 :: Rel Int
--    r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
--    r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
--    r3 = ([1..9],[(1,3), (2,6), (8,9), (3,6)])
-- ---------------------------------------------------------------------

r1, r2, r3 :: Rel Int
r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
r3 = ([1..9],[(1,3), (2,6), (8,9), (3,6)])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    universo :: Eq a => Rel a -> [a]
-- tal que (universo r) es el universo de la relación r. Por ejemplo,
--    r1           ==  ([1,2,3,4,5,6,7,8,9],[(1,3),(2,6),(8,9),(2,7)])
--    universo r1  ==  [1,2,3,4,5,6,7,8,9]
-- ---------------------------------------------------------------------

universo :: Eq a => Rel a -> [a]
universo (us,_) = us

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    grafo :: Eq a => ([a],[(a,a)]) -> [(a,a)]
-- tal que (grafo r) es el grafo de la relación r. Por ejemplo,
--    r1        ==  ([1,2,3,4,5,6,7,8,9],[(1,3),(2,6),(8,9),(2,7)])
--    grafo r1  ==  [(1,3),(2,6),(8,9),(2,7)]
-- ---------------------------------------------------------------------

grafo :: Eq a => ([a],[(a,a)]) -> [(a,a)]
grafo (_,ps) = ps

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    reflexiva :: Eq a => Rel a -> Bool
-- tal que (reflexiva r) se verifica si la relación r es reflexiva. Por
-- ejemplo,
```

```haskell
--    reflexiva ([1,3],[(1,1),(1,3),(3,3)])    ==  True
--    reflexiva ([1,2,3],[(1,1),(1,3),(3,3)])  ==  False
-- ---------------------------------------------------------------------

reflexiva :: Eq a => Rel a -> Bool
reflexiva (us,ps) = and [elem (x,x) ps | x <- us]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    simetrica :: Eq a => Rel a -> Bool
-- tal que (simetrica r) se verifica si la relación r es simétrica. Por
-- ejemplo,
--    simetrica ([1,3],[(1,1),(1,3),(3,1)])  ==  True
--    simetrica ([1,3],[(1,1),(1,3),(3,2)])  ==  False
--    simetrica ([1,3],[])                   ==  True
-- ---------------------------------------------------------------------

simetrica :: Eq a => Rel a -> Bool
simetrica (us,ps) = and [(y,x) `elem` ps | (x,y) <- ps]

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

subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    composicion :: Eq a => Rel a -> Rel a -> Rel a
-- tal que (composicion r s) es la composición de las relaciones r y
-- s. Por ejemplo,
--    ghci> composicion ([1,2],[(1,2),(2,2)]) ([1,2],[(2,1)])
--    ([1,2],[(1,1),(2,1)])
-- ---------------------------------------------------------------------
```

```
composicion :: Eq a => Rel a -> Rel a -> Rel a
composicion (xs,ps) (_,qs) =
    (xs,[(x,z) | (x,y) <- ps, (y',z) <- qs, y == y'])

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    transitiva :: Eq a => Rel a -> Bool
-- tal que (transitiva r) se verifica si la relación r es transitiva.
-- Por ejemplo,
--    transitiva ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])  ==  True
--    transitiva ([1,3,5],[(1,1),(1,3),(3,1),(5,5)])        ==  False
-- ---------------------------------------------------------------------

transitiva :: Eq a => Rel a -> Bool
transitiva r@(xs,ps) =
    subconjunto (grafo (composicion r r)) ps

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    esEquivalencia :: Eq a => Rel a -> Bool
-- tal que (esEquivalencia r) se verifica si la relación r es de
-- equivalencia. Por ejemplo,
--    ghci> esEquivalencia ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])
--    True
--    ghci> esEquivalencia ([1,2,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])
--    False
--    ghci> esEquivalencia ([1,3,5],[(1,1),(1,3),(3,3),(5,5)])
--    False
-- ---------------------------------------------------------------------

esEquivalencia :: Eq a => Rel a -> Bool
esEquivalencia r = reflexiva r && simetrica r && transitiva r

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    irreflexiva :: Eq 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,
```

```
--    irreflexiva ([1,2,3],[(1,2),(2,1),(2,3)])  ==  True
--    irreflexiva ([1,2,3],[(1,2),(2,1),(3,3)])  ==  False
-- ---------------------------------------------------------------------

irreflexiva :: Eq a => Rel a -> Bool
irreflexiva (xs,ps) = and [(x,x) `notElem` ps | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    antisimetrica :: Eq 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 ([1,2],[(1,2)])        ==  True
--    antisimetrica ([1,2],[(1,2),(2,1)])  ==  False
--    antisimetrica ([1,2],[(1,1),(2,1)])  ==  True
-- ---------------------------------------------------------------------

antisimetrica :: Eq a => Rel a -> Bool
antisimetrica (_,ps) =
    null [(x,y) | (x,y) <- ps, x /= y, (y,x) `elem` ps]

-- Otra definición es
antisimetrica' :: Eq a => Rel a -> Bool
antisimetrica' (xs,ps) =
    and [((x,y) `elem` ps && (y,x) `elem` ps) --> (x == y)
         | x <- xs, y <- xs]
    where p --> q = not p || q

-- Las dos definiciones son equivalentes
prop_antisimetrica :: Rel Int -> Bool
prop_antisimetrica r =
    antisimetrica r == antisimetrica' r

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

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
```

```
--    total :: Eq a => Rel a -> Bool
-- tal que (total 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 etá relacionado con x. Por ejemplo,
--    total ([1,3],[(1,1),(3,1),(3,3)])  ==  True
--    total ([1,3],[(1,1),(3,1)])        ==  False
--    total ([1,3],[(1,1),(3,3)])        ==  False
-- ---------------------------------------------------------------------

total :: Eq a => Rel a -> Bool
total (xs,ps) =
    and [(x,y) `elem` ps || (y,x) `elem` ps | x <- xs, y <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Comprobar con QuickCheck que las relaciones totales son
-- reflexivas.
-- ---------------------------------------------------------------------

prop_total_reflexiva :: Rel Int -> Property
prop_total_reflexiva r =
    total r ==> reflexiva r

-- La comprobación es
--    ghci> quickCheck prop_total_reflexiva
--    *** Gave up! Passed only 19 tests.

-- ---------------------------------------------------------------------
-- § Clausuras                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    clausuraReflexiva :: Eq 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 ([1,3],[(1,1),(3,1)])
--    ([1,3],[(1,1),(3,1),(3,3)])
-- ---------------------------------------------------------------------

clausuraReflexiva :: Eq a => Rel a -> Rel a
```

```
clausuraReflexiva (xs,ps) =
  (xs, ps `union` [(x,x) | x <- xs])

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

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    clausuraSimetrica :: Eq 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 ([1,3,5],[(1,1),(3,1),(1,5)])
--    ([1,3,5],[(1,1),(3,1),(1,5),(1,3),(5,1)])
-- ---------------------------------------------------------------------

clausuraSimetrica :: Eq a => Rel a -> Rel a
clausuraSimetrica (xs,ps) =
    (xs, ps `union` [(y,x) | (x,y) <- ps])

-- ---------------------------------------------------------------------
-- 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 :: Eq 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 ([1..6],[(1,2),(2,5),(5,6)])
--    ([1,2,3,4,5,6],[(1,2),(2,5),(5,6),(1,5),(2,6),(1,6)])
-- ---------------------------------------------------------------------

clausuraTransitiva :: Eq a => Rel a -> Rel a
clausuraTransitiva (xs,ps) = (xs, aux ps)
    where aux xs | cerradoTr xs = xs
                 | otherwise    = aux (xs `union` (comp xs xs))
          cerradoTr r = subconjunto (comp r r) r
          comp r s    = [(x,z) | (x,y) <- r, (y',z) <- s, y == y']

-- ---------------------------------------------------------------------
-- Ejercicio 19. Comprobar con QuickCheck que clausuraTransitiva es
-- transitiva.
-- ---------------------------------------------------------------------

prop_ClausuraTransitiva :: Rel Int -> Bool
prop_ClausuraTransitiva r =
    transitiva (clausuraTransitiva r)

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

Relación 30

Operaciones con conjuntos

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

-- 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/temas/tema-17t.pdf
-- ---------------------------------------------------------------------

{-# LANGUAGE FlexibleInstances #-}

-- ---------------------------------------------------------------------
-- § 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
```

```
-- Procedimiento de escritura de los conjuntos.
instance (Show a) => Show (Conj a) where
    showsPrec _ (Cj s) cad = showConj s cad

showConj []     cad = showString "{}" cad
showConj (x:xs) cad = showChar '{' (shows x (showl xs cad))
     where showl []     cad = showChar '}' cad
           showl (x:xs) cad = showChar ',' (shows x (showl xs cad))

-- Ejemplo de conjunto:
--    ghci> c1
--    {0,1,2,3,5,7,9}
c1, c2, c3, c4 :: Conj Int
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]
c2 = foldr inserta vacio [2,6,8,6,1,2,1,9,6]
c3 = Cj [2..100000]
c4 = Cj [1..100000]

-- vacio es el conjunto vacío. Por ejemplo,
--    ghci> 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,
--    c1              ==  {0,1,2,3,5,7,9}
--    pertenece 3 c1  ==  True
--    pertenece 4 c1  ==  False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = x `elem` takeWhile (<= x) s

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--    c1            ==  {0,1,2,3,5,7,9}
```

```
--    inserta 5 c1  ==  {0,1,2,3,5,7,9}
--    inserta 4 c1  ==  {0,1,2,3,4,5,7,9}
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (agrega x s)
    where agrega x []                     = [x]
          agrega x s@(y:ys) | x > y       = y : agrega x ys
                            | x < y       = x : s
                            | otherwise   = s

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--    c1            ==  {0,1,2,3,5,7,9}
--    elimina 3 c1  ==  {0,1,2,5,7,9}
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (elimina x s)
    where elimina x []                     = []
          elimina x s@(y:ys) | x > y       = y : elimina x ys
                             | x < y       = s
                             | otherwise = ys

-- ---------------------------------------------------------------------
-- § 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
-- ---------------------------------------------------------------------

-- Se presentan distintas definiciones y se compara su eficiencia.

-- 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]

-- 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:
--    ghci> subconjunto1 (Cj [2..100000]) (Cj [1..1000000])
--      C-c C-cInterrupted.
--    ghci> subconjunto2 (Cj [2..100000]) (Cj [1..1000000])
--      C-c C-cInterrupted.
--    ghci> subconjunto3 (Cj [2..100000]) (Cj [1..1000000])
--    True
--    (0.52 secs, 26097076 bytes)
--    ghci> subconjunto4 (Cj [2..100000]) (Cj [1..1000000])
--    True
--    (0.66 secs, 32236700 bytes)
--    ghci> subconjunto1 (Cj [2..100000]) (Cj [1..10000])
--    False
--    (0.54 secs, 3679024 bytes)
--    ghci> subconjunto2 (Cj [2..100000]) (Cj [1..10000])
--    False
```

```
--    (38.19 secs, 1415562032 bytes)
--    ghci> subconjunto3 (Cj [2..100000]) (Cj [1..10000])
--    False
--    (0.08 secs, 3201112 bytes)
--    ghci> 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
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
-- ---------------------------------------------------------------------

-- Se considera distintas definiciones y se compara la eficiencia.

-- 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
--    ghci> :set +s
--    ghci> let c = Cj [1..1000]
--    ghci> cardinal (union1 c c)
--    1000
--    (1.04 secs, 56914332 bytes)
--    ghci> 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
--    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]) == {}
-- ---------------------------------------------------------------------

-- Se da distintas definiciones y se compara su eficiencia.

-- 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
-- ghci> interseccion1 (Cj [2..1000000]) (Cj [1])
-- {}
-- (0.32 secs, 80396188 bytes)
-- ghci> 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
-- 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
-- elemsntos 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

-- ---------------------------------------------------------------------
-- 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
--    ghci> 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
--    ghci> quickCheck propSubconjuntoAntisimetrica
--    *** Gave up! Passed only 13 tests.

propSubconjuntoTransitiva :: Conj Int -> Conj Int -> Conj Int -> Property
```

```haskell
propSubconjuntoTransitiva c1 c2 c3 =
    subconjunto c1 c2 && subconjunto c2 c3 ==> subconjunto c1 c3

-- La comprobación es
--    ghci> quickCheck propSubconjuntoTransitiva
--    *** Gave up! Passed only 7 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 21. Comprobar con QuickCheck que el conjunto vacío está
-- contenido en cualquier conjunto.
-- ---------------------------------------------------------------------

propSubconjuntoVacio:: Conj Int -> Bool
propSubconjuntoVacio c = subconjunto vacio c

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

propVacioNeutroUnion:: Conj Int -> Bool
propVacioNeutroUnion c =
```

```
    union c vacio == c

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

propUnionCommutativa:: Conj Int -> Conj Int -> Bool
propUnionCommutativa c1 c2 =
    union c1 c2 == union c2 c1

-- La comprobación es
--    ghci> 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
--    ghci> 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
--    ghci> 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
--    ghci> 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)
-- ---------------------------------------------------------------------


propInterseccionIdempotente :: Conj Int -> Bool
propInterseccionIdempotente c =
    interseccion c c == c

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

propVacioInterseccion:: Conj Int -> Bool
propVacioInterseccion c =
    interseccion c vacio == vacio

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

propInterseccionCommutativa:: Conj Int -> Conj Int -> Bool
propInterseccionCommutativa c1 c2 =
    interseccion c1 c2 == interseccion c2 c1

-- La comprobación es
--    ghci> 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
--    ghci> 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
--    ghci> 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
--    ghci> 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
--    ghci> 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
--    ghci> quickCheck propDiferenciaVacio2
--    +++ OK, passed 100 tests.

propDiferenciaVacio2 :: Conj Int -> Bool
propDiferenciaVacio2 c = diferencia vacio c == vacio

-- La comprobación es
--    ghci> 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
--    ghci> 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
--    ghci> quickCheck propDiferenciaDif2
--    +++ OK, passed 100 tests.

propDiferenciaSubc:: Conj Int -> Conj Int -> Bool
propDiferenciaSubc c1 c2 =
  subconjunto (diferencia c1 c2) c1

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

```
propDiferenciaDisj:: Conj Int -> Conj Int -> Bool
propDiferenciaDisj c1 c2 =
  disjuntos c1 (diferencia c2 c1)

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

propDiferenciaUI:: Conj Int -> Conj Int -> Bool
propDiferenciaUI c1 c2 =
  diferencia (union c1 c2) c1 == diferencia c2 (interseccion c1 c2)

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

-- ---------------------------------------------------------------------
-- Generador de conjuntos                                             --
-- ---------------------------------------------------------------------

-- genConjunto es un generador de conjuntos. Por ejemplo,
--    ghci> 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 31

Implementación del TAD de los grafos mediante listas

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

-- El objetivo de esta relación es implementar el TAD de los grafos
-- mediante listas, de manera análoga a las implementaciones estudiadas
-- en el tema 22 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-22.pdf
-- y usando la mismas signatura.

-- ---------------------------------------------------------------------
-- Signatura                                                           --
-- ---------------------------------------------------------------------

module Rel_31_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

-- ---------------------------------------------------------------------
-- 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 d 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 32

Problemas básicos con el TAD de los grafos

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

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de los grafos, utilizando las implementaciones estudiadas
-- en el tema 22 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/codigos.zip
--
-- Las transparencias del tema 22 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-22.pdf

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

{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

import Data.Array
import Data.List (nub)
import Test.QuickCheck

-- Hay que seleccionar una implementación del TAD de los grafos
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

```
-- ---------------------------------------------------------------------
-- Ejemplos                                                           --
-- ---------------------------------------------------------------------

-- Para los ejemplos se usarán los siguientes grafos.
g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11 :: 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)]

-- ---------------------------------------------------------------------
-- Ejercicio 1. El grafo completo de orden n, K(n), es un grafo no
-- dirigido cuyos conjunto de vértices es {1,..n} y tiene una arista
-- entre par de vértices distintos. Definir la función,
--    completo :: Int -> Grafo Int Int
-- tal que (completo n) es el grafo completo de orden n. Por ejemplo,
--    ghci> completo 4
--    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) xs
```

```
    where xs = [(x,y,0) | x <- [1..n], y <- [1..n], x < y]

completo' :: Int -> Grafo Int Int
completo' n = creaGrafo ND (1,n) [(a,b,0)|a<-[1..n],b<-[1..a-1]]

-- ---------------------------------------------------------------------
-- 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) xs
    where xs = [(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    nVertices :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nVertices g) es el número de vértices del grafo g. Por
-- ejemplo,
--    nVertices (completo 4)  ==  4
--    nVertices (completo 5)  ==  5
-- ---------------------------------------------------------------------

nVertices :: (Ix v,Num p) => Grafo v p ->  Int
nVertices = length . nodos

-- ---------------------------------------------------------------------
-- Ejercicio 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]
-- ---------------------------------------------------------------------

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 y los lazos se cuentan dos veces. Por ejemplo,
--    nAristas g1            ==  8
--    nAristas g2            ==  7
--    nAristas g10           ==  4
--    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) `div` 2) + nLazos g

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

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

gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 12. El grado negativo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g incidentes con v. Definir la función
--    gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (gradoNeg g v) es el grado negativo del vértice v en el grafo
-- g. Por ejemplo,
--    gradoNeg g1 5  ==  4
--    gradoNeg g2 5  ==  3
--    gradoNeg g2 1  ==  0
-- ---------------------------------------------------------------------
```

```
gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 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  ==  3
--    grado g10 3 == 4
--    grado g11 3 == 4
-- ---------------------------------------------------------------------

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.

-- ---------------------------------------------------------------------
-- 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
-- ---------------------------------------------------------------------

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 m son regulares y
-- usarla para comprobar que todos los grafos completo desde el de orden
-- 1 hasta el de orden 30 son regulares.
-- ---------------------------------------------------------------------

prop_CompletoRegular :: Int -> Int -> Bool
prop_CompletoRegular m n = and [regular (completo x) | x <- [m..n]]

-- La comprobación es
--    ghci> prop_CompletoRegular 1 30
```

```
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 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

-- ---------------------------------------------------------------------
-- 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                                             --
-- ---------------------------------------------------------------------

-- (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)

-- 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
```

Relación 33

Enumeraciones de los números racionales

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

-- El objetivo de esta relación es construir dos enumeraciones de los
-- números racionales. Concretamente,
-- * una enumeración basada en las representaciones hiperbinarias y
-- * una enumeración basada en los los árboles de Calkin-Wilf.
-- También se incluye la comprobación de la igualdad de las dos
-- sucesiones y una forma alternativa de calcular el número de
-- representaciones hiperbinarias mediante la función fucs.
--
-- Esta relación se basa en los siguientes artículos:
-- * Gaussianos "Sorpresa sumando potencias de 2" http://goo.gl/AHdAG
-- * N. Calkin y H.S. Wilf "Recounting the rationals" http://goo.gl/gVZtW
-- * Wikipedia "Calkin-Wilf tree" http://goo.gl/cB3vn

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

import Data.List
import Test.QuickCheck

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

```haskell
-- Numeración de los racionales mediante representaciones hiperbinarias
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la constante
--    potenciasDeDos :: [Integer]
-- tal que potenciasDeDos es la lista de las potencias de 2. Por
-- ejemplo,
--    take 10 potenciasDeDos  ==  [1,2,4,8,16,32,64,128,256,512]
-- ---------------------------------------------------------------------

potenciasDeDos :: [Integer]
potenciasDeDos = [2^n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    empiezaConDos :: Eq a => a -> [a] -> Bool
-- tal que (empiezaConDos x ys) se verifica si los dos primeros
-- elementos de ys son iguales a x. Por ejemplo,
--    empiezaConDos 5 [5,5,3,7]  ==  True
--    empiezaConDos 5 [5,3,5,7]  ==  False
--    empiezaConDos 5 [5,5,5,7]  ==  True
-- ---------------------------------------------------------------------

empiezaConDos x (y1:y2:ys) = y1 == x && y2 == x
empiezaConDos x (y1:y2:ys) = y1 == x && y2 == x
empiezaConDos x _          = False

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    representacionesHB :: Integer -> [[Integer]]
-- tal que (representacionesHB n) es la lista de las representaciones
-- hiperbinarias del número n como suma de potencias de 2 donde cada
-- sumando aparece como máximo 2 veces. Por ejemplo
--    representacionesHB 5  ==  [[1,2,2],[1,4]]
--    representacionesHB 6  ==  [[1,1,2,2],[1,1,4],[2,4]]
-- ---------------------------------------------------------------------

representacionesHB :: Integer -> [[Integer]]
representacionesHB n = representacionesHB' n potenciasDeDos
```

```
representacionesHB' n (x:xs)
    | n == 0    = [[]]
    | x == n    = [[x]]
    | x <  n    = [x:ys | ys <- representacionesHB' (n-x) (x:xs),
                          not (empiezaConDos x ys)] ++
                  representacionesHB' n xs
    | otherwise = []

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    nRepresentacionesHB :: Integer -> Integer
-- tal que (nRepresentacionesHB n) es el número de las representaciones
-- hiperbinarias del número n como suma de potencias de 2 donde cada
-- sumando aparece como máximo 2 veces. Por ejemplo,
--    ghci> [nRepresentacionesHB n | n <- [0..20]]
--    [1,1,2,1,3,2,3,1,4,3,5,2,5,3,4,1,5,4,7,3,8]
-- ---------------------------------------------------------------------

nRepresentacionesHB :: Integer -> Integer
nRepresentacionesHB = genericLength . representacionesHB

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    termino :: Integer -> (Integer,Integer)
-- tal que (termino n) es el par formado por el número de
-- representaciones hiperbinarias de n y de n+1 (que se interpreta como
-- su cociente). Por ejemplo,
--    termino 4  ==  (3,2)
-- ---------------------------------------------------------------------

termino :: Integer -> (Integer,Integer)
termino n = (nRepresentacionesHB n, nRepresentacionesHB (n+1))

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    sucesionHB :: [(Integer,Integer)]
-- sucesionHB es la la sucesión cuyo témino n-ésimo es (termino n); es
-- decir, el par formado por el número de representaciones hiperbinarias
-- de n y de n+1. Por ejemplo,
--    ghci> take 10 sucesionHB
```

```
--    [(1,1),(1,2),(2,1),(1,3),(3,2),(2,3),(3,1),(1,4),(4,3),(3,5)]
-- ---------------------------------------------------------------------

sucesionHB :: [(Integer,Integer)]
sucesionHB = [termino n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck que, para todo n,
-- (nRepresentacionesHB n) y  (nRepresentacionesHB (n+1)) son primos
-- entre sí.
-- ---------------------------------------------------------------------

prop_irreducibles :: Integer -> Property
prop_irreducibles n =
    n >= 0 ==>
    gcd (nRepresentacionesHB n) (nRepresentacionesHB (n+1)) == 1

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

-- ---------------------------------------------------------------------
-- Ejercicio 8. Comprobar con QuickCheck que todos los elementos de la
-- sucesionHB son distintos.
-- ---------------------------------------------------------------------

prop_distintos :: Integer -> Integer -> Bool
prop_distintos n m =
    termino n' /= termino m'
    where n' = abs n
          m' = n' + abs m + 1

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

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    contenido :: Integer -> Integer -> Bool
-- tal que (contenido n) se verifica si la expresiones reducidas de
```

```
-- todas las fracciones x/y, con x e y entre 1 y n, pertenecen a la
-- sucesionHB. Por ejemplo,
--    contenidos 5  ==  True
-- ---------------------------------------------------------------------

contenido :: Integer -> Bool
contenido n =
    and [pertenece (reducida (x,y)) sucesionHB |
         x <- [1..n], y <- [1..n]]
    where pertenece x (y:ys) = x == y || pertenece x ys
          reducida (x,y) = (x `div` z, y `div` z)
              where z = gcd x y

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    indice :: (Integer,Integer) -> Integer
-- tal que (indice (a,b)) es el índice del par (a,b) en la sucesión de
-- los racionales. Por ejemplo,
--    indice (3,2)  ==  4
-- ---------------------------------------------------------------------

indice :: (Integer,Integer) -> Integer
indice (a,b) = head [n | (n,(x,y)) <- zip [0..] sucesionHB,
                         (x,y) == (a,b)]

-- ---------------------------------------------------------------------
-- Numeraciones mediante árboles de Calkin-Wilf                       --
-- ---------------------------------------------------------------------

-- El árbol de Calkin-Wilf es el árbol definido por las siguientes
-- reglas:
--    * El nodo raíz es el (1,1)
--    * Los hijos del nodo (x,y) son (x,x+y) y (x+y,y)
-- Por ejemplo, los 4 primeros niveles del árbol de Calkin-Wilf son
--                        (1,1)
--                          |
--              +-----------+-----------+
--              |                       |
--            (1,2)                   (2,1)
--              |                       |
```

```
--         +-----+-----+           +-----+-----+
--         |           |           |           |
--       (1,3)       (3,2)       (2,3)       (3,1)
--         |           |           |           |
--      +--+--+     +--+--+     +--+--+     +--+--+
--      |     |     |     |     |     |     |     |
--    (1,4) (4,3) (3,5) (5,2) (2,5) (5,3) (3,4) (4,1)

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    sucesores :: (Integer,Integer) -> [(Integer,Integer)]
-- tal que (sucesores (x,y)) es la lista de los hijos del par (x,y) en
-- el árbol de Calkin-Wilf. Por ejemplo,
--    sucesores (3,2)  ==  [(3,5),(5,2)]
-- ---------------------------------------------------------------------

sucesores :: (Integer,Integer) -> [(Integer,Integer)]
sucesores (x,y) = [(x,x+y),(x+y,y)]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
-- tal que (siguiente xs) es la lista formada por los hijos de los
-- elementos de xs en el árbol de Calkin-Wilf. Por ejemplo,
--    ghci> siguiente [(1,3),(3,2),(2,3),(3,1)]
--    [(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]
-- ---------------------------------------------------------------------

siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
siguiente xs = [p | x <- xs, p <- sucesores x]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la constante
--    nivelesCalkinWilf:: [[(Integer,Integer)]]
-- tal que nivelesCalkinWilf es la lista de los niveles del árbol de
-- Calkin-Wilf. Por ejemplo,
--    ghci> take 4 nivelesCalkinWilf
--    [[(1,1)],
--     [(1,2),(2,1)],
--     [(1,3),(3,2),(2,3),(3,1)],
```

```
--     [(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]]
-- ---------------------------------------------------------------------

nivelesCalkinWilf:: [[(Integer,Integer)]]
nivelesCalkinWilf = iterate siguiente [(1,1)]

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la constante
--    sucesionCalkinWilf :: [(Integer,Integer)]
-- tal que sucesionCalkinWilf es la lista correspondiente al recorrido
-- en anchura del árbol de Calkin-Wilf. Por ejemplo,
--    ghci> take 10 sucesionCalkinWilf
--    [(1,1),(1,2),(2,1),(1,3),(3,2),(2,3),(3,1),(1,4),(4,3),(3,5)]
-- ---------------------------------------------------------------------

sucesionCalkinWilf :: [(Integer,Integer)]
sucesionCalkinWilf = concat nivelesCalkinWilf

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    igual_sucesion_HB_CalkinWilf :: Int -> Bool
-- tal que (igual_sucesion_HB_CalkinWilf n) se verifica si los n
-- primeros términos de la sucesión HB son iguales que los de la
-- sucesión de Calkin-Wilf. Por ejemplo,
--    igual_sucesion_HB_CalkinWilf 20  ==  True
-- ---------------------------------------------------------------------

igual_sucesion_HB_CalkinWilf :: Int -> Bool
igual_sucesion_HB_CalkinWilf n =
    take n sucesionCalkinWilf == take n sucesionHB

-- ---------------------------------------------------------------------
-- Número de representaciones hiperbinarias mediante la función fusc
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    fusc :: Integer -> Integer
-- tal que
--    fusc(0)    = 1
```

```
--    fusc(2n+1) = fusc(n)
--    fusc(2n+2) = fusc(n+1)+fusc(n)
-- Por ejemplo,
--    fusc 4  ==  3
-- ---------------------------------------------------------------------

fusc :: Integer -> Integer
fusc 0 = 1
fusc  n | odd n     = fusc ((n-1) `div` 2)
        | otherwise = fusc(m+1) + fusc m
                      where m = (n-2) `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 17. Comprobar con QuickCheck que, para todo n, (fusc n) es
-- el número de las representaciones hiperbinarias del número n como
-- suma de potencias de 2 donde cada sumando aparece como máximo 2
-- veces; es decir, que las funciones fusc y nRepresentacionesHB son
-- equivalentes.
-- ---------------------------------------------------------------------

prop_fusc :: Integer -> Bool
prop_fusc n = nRepresentacionesHB n' == fusc n'
              where n' = abs n

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

Apéndice A

Exámenes

En este apéndice se recopila las soluciones de los exámenes realizados durante el curso.

A.1. Examen 1 (26 de Octubre de 2011)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (26 de octubre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función numeroDeRaices 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
-- ---------------------------------------------------------------------

numeroDeRaices a b c | d < 0     = 0
                     | d == 0    = 1
                     | otherwise = 2
               where d = b^2-4*a*c

-- ---------------------------------------------------------------------
-- Ejercicio 2. 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 tal que
-- (mayorRectangulo r1 r2) es el rectángulo de mayor área ente 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)
-- ---------------------------------------------------------------------

mayorRectanglo (a,b) (c,d) | a*b >= c*d = (a,b)
                           | otherwise = (c,d)

-- ---------------------------------------------------------------------
-- Ejercicio 3. 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)
```

A.2. Examen 2 (30 de Noviembre de 2011)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (30 de noviembre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [Problema 357 del Project Euler] Un número natural n
-- es especial si para todo divisor d de n, d+n/d es primo. Definir la
-- función
--    especial :: Integer -> Bool
-- tal que (especial x) se verifica si x es especial. Por ejemplo,
--    especial 30  ==  True
--    especial 20  ==  False
-- ---------------------------------------------------------------------

especial :: Integer -> Bool
especial x = and [esPrimo (d + x `div` d) | d <- divisores x]

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

```
divisores x = [d | d <- [1..x], x `rem` d == 0]

-- (esPrimo x) se verifica si x es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 8  ==  False
esPrimo :: Integer -> Bool
esPrimo x = divisores x == [1,x]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    sumaEspeciales :: Integer -> Integer
-- tal que (sumaEspeciales n) es la suma de los números especiales
-- menores o iguales que n. Por ejemplo,
--    sumaEspeciales 100  ==  401
-- ---------------------------------------------------------------------

-- Por comprensión
sumaEspeciales :: Integer -> Integer
sumaEspeciales n = sum [x | x <- [1..n], especial x]

-- Por recursión
sumaEspecialesR :: Integer -> Integer
sumaEspecialesR 0 = 0
sumaEspecialesR n | especial n = n + sumaEspecialesR (n-1)
                  | otherwise  = sumaEspecialesR (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    refinada :: [Float] -> [Float]
-- tal que (refinada xs) es la lista obtenida intercalando entre cada
-- dos elementos consecutivos de xs su media aritmética. Por ejemplo,
--    refinada [2,7,1,8]  ==  [2.0,4.5,7.0,4.0,1.0,4.5,8.0]
--    refinada [2]        ==  [2.0]
--    refinada []         ==  []
-- ---------------------------------------------------------------------

refinada :: [Float] -> [Float]
refinada (x:y:zs) = x : (x+y)/2 : refinada (y:zs)
refinada xs       = xs
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. En este ejercicio vamos a comprobar que la ecuación
-- diofántica
--    1/x_1 + 1/x_2 + ... + 1/x_n = 1
-- tiene solución; es decir, que para todo n >= 1 se puede construir una
-- lista de números enteros de longitud n tal que la suma de sus
-- inversos es 1. Para ello, basta observar que si
--    [x_1, x_2, ..., x_n]
-- es una solución, entonces
--    [2, 2*x_1, 2*x_2, ..., 2*x_n]
-- también lo es. Definir la función solucion tal que (solucion n) es la
-- solución de longitud n construida mediante el método anterior. Por
-- ejemplo,
--    solucion 1  ==  [1]
--    solucion 2  ==  [2,2]
--    solucion 3  ==  [2,4,4]
--    solucion 4  ==  [2,4,8,8]
--    solucion 5  ==  [2,4,8,16,16]
-- ---------------------------------------------------------------------

solucion 1 = [1]
solucion n = 2 : [2*x | x <- solucion (n-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función esSolucion tal que (esSolucion xs)
-- se verifica si la suma de los inversos de xs es 1. Por ejemplo,
--    esSolucion [4,2,4]       ==  True
--    esSolucion [2,3,4]       ==  False
--    esSolucion (solucion 5)  ==  True
-- ---------------------------------------------------------------------

esSolucion xs = sum [1/x | x <- xs] == 1
```

A.3. Examen 3 (25 de Enero de 2012)

```
-- Informática (1º del Grado en Matemáticas)
-- 3º examen de evaluación continua (25 de enero de 2012)
-- ---------------------------------------------------------------------

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

```
-- Ejercicio 1.1. [2 puntos] Un número es muy compuesto si tiene más
-- divisores que sus anteriores. Por ejemplo, 12 es muy compuesto porque
-- tiene 6 divisores (1, 2, 3, 4, 6, 12) y todos los números del 1 al 11
-- tienen menos de 6 divisores.
--
-- Definir la función
--    esMuyCompuesto :: Int -> Bool
-- tal que (esMuyCompuesto x) se verifica si x es un número muy
-- compuesto. Por ejemplo,
--    esMuyCompuesto 24  ==  True
--    esMuyCompuesto 25  ==  False
-- Calcular  el menor número muy compuesto de 4 cifras.
-- ---------------------------------------------------------------------

esMuyCompuesto :: Int -> Bool
esMuyCompuesto x =
    and [numeroDivisores y < n | y <- [1..x-1]]
    where n = numeroDivisores x

-- (numeroDivisores x) es el número de divisores de x. Por ejemplo,
--    numeroDivisores 24  ==  8
numeroDivisores :: Int -> Int
numeroDivisores = length . divisores

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--    divisores 24  ==  [1,2,3,4,6,8,12,24]
divisores :: Int -> [Int]
divisores x = [y | y <- [1..x], mod x y == 0]

-- Los primeros números muy compuestos son
--    ghci> take 14 [x | x <- [1..], esMuyCompuesto x]
--    [1,2,4,6,12,24,36,48,60,120,180,240,360,720]

-- El cálculo del menor número muy compuesto de 4 cifras es
--    ghci> head [x | x <- [1000..], esMuyCompuesto x]
--    1260

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. [1 punto] Definir la función
--    muyCompuesto :: Int -> Int
```

```
-- tal que (muyCompuesto n) es el n-ésimo número muy compuesto. Por
-- ejemplo,
--    muyCompuesto 10  ==  180
-- ---------------------------------------------------------------------

muyCompuesto :: Int -> Int
muyCompuesto n =
    [x | x <- [1..], esMuyCompuesto x] !! n

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. [2 puntos] [Problema 37 del proyecto Euler] Un número
-- primo es truncable si los números que se obtienen eliminado cifras,
-- de derecha a izquierda, son primos. Por ejemplo, 599 es un primo
-- truncable porque 599, 59 y 5 son primos; en cambio, 577 es un primo
-- no truncable porque 57 no es primo.
--
-- Definir la función
--    primoTruncable :: Int -> Bool
-- tal que (primoTruncable x) se verifica si x es un primo
-- truncable. Por ejemplo,
--    primoTruncable 599  ==  True
--    primoTruncable 577  ==  False
-- ---------------------------------------------------------------------

primoTruncable :: Int -> Bool
primoTruncable x
    | x < 10    = primo x
    | otherwise = primo x && primoTruncable (x `div` 10)

-- (primo x) se verifica si x es primo.
primo :: Int -> Bool
primo x = x == head (dropWhile (<x) primos)

-- primos es la lista de los números primos.
primos :: [Int ]
primos = criba [2..]
    where criba :: [Int] -> [Int]
          criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]

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

```
-- Ejercicio 2.2. [1.5 puntos] Definir la función
--    sumaPrimosTruncables :: Int -> Int
-- tal que (sumaPrimosTruncables n) es la suma de los n primeros primos
-- truncables. Por ejemplo,
--    sumaPrimosTruncables 10  ==  249
-- Calcular la suma de los 20 primos truncables.
-- ---------------------------------------------------------------------

sumaPrimosTruncables :: Int -> Int
sumaPrimosTruncables n =
    sum (take n [x | x <- primos, primoTruncable x])

-- El cálculo es
--    ghci> sumaPrimosTruncables 20
--    2551

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. [2 puntos] Los números enteros se pueden ordenar como
-- sigue
--    0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5, -6, 6, -7, 7, ...
-- Definir 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]
-- ---------------------------------------------------------------------

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

-- Otra definicición, por iteración, es
enteros_1 :: [Int]
enteros_1 = iterate siguiente 0
    where siguiente x | x >= 0    = -x-1
                      | otherwise = -x

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. [1.5 puntos] 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
-- ---------------------------------------------------------------------

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

-- Definición por recursión
posicion_1 :: Int -> Int
posicion_1 x = aux enteros 0
    where aux (y:ys) n | x == y    = n
                       | otherwise = aux ys (n+1)

-- Definición por comprensión
posicion_2 :: Int -> Int
posicion_2 x = head [n | (n,y) <- zip [0..] enteros, y == x]

-- Definición directa
posicion_3 :: Int -> Int
posicion_3 x | x >= 0    = 2*x
             | otherwise = 2*(-x)-1
```

A.4. Examen 4 (29 de Febrero de 2012)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (29 de febrero de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] En el enunciado de uno de los problemas de
-- las Olimpiadas matemáticas de Brasil se define el primitivo de un
-- número como sigue:
--    Dado un número natural N, multiplicamos todos sus dígitos,
--    repetimos este procedimiento hasta que quede un solo dígito al
--    cual llamamos primitivo de N. Por ejemplo para 327: 3x2x7 = 42 y
--    4x2 = 8. Por lo tanto, el primitivo de 327 es 8.
--
-- Definir la función
--    primitivo :: Integer -> Integer
-- tal que (primitivo n) es el primitivo de n. Por ejemplo.
```

```haskell
--    primitivo 327  ==  8
-- ---------------------------------------------------------------------

primitivo :: Integer -> Integer
primitivo n | n < 10    = n
            | otherwise = primitivo (producto n)

-- (producto n) es el producto de las cifras de n. Por ejemplo,
--    producto 327  ==  42
producto :: Integer -> Integer
producto = product . cifras

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 327  ==  [3,2,7]
cifras :: Integer -> [Integer]
cifras n = [read [y] | y <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Definir la función
--    sumas :: Int -> [Int] -> [Int]
-- tal que (sumas n xs) es la lista de los números que se pueden obtener
-- como suma de n, o menos, elementos de xs. Por ejemplo,
--    sumas 0 [2,5]    ==  [0]
--    sumas 1 [2,5]    ==  [2,5,0]
--    sumas 2 [2,5]    ==  [4,7,2,10,5,0]
--    sumas 3 [2,5]    ==  [6,9,4,12,7,2,15,10,5,0]
--    sumas 2 [2,3,5]  ==  [4,5,7,2,6,8,3,10,5,0]
-- ---------------------------------------------------------------------

sumas :: Int -> [Int] -> [Int]
sumas 0 _  = [0]
sumas _ [] = [0]
sumas n (x:xs) = [x+y | y <- sumas (n-1) (x:xs)] ++ sumas n xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Los árboles binarios se pueden representar
-- mediante el siguiente tipo de datos
--    data Arbol = H
--               | N Int Arbol Arbol
-- Por ejemplo, el árbol
```

```
--           9
--          / \
--         /   \
--        3     7
--       / \   / \
--      /   \ H   H
--     2     4
--    / \   / \
--   H   H H   H
-- se representa por
--    N 9 (N 3 (N 2 H H) (N 4 H H)) (N 7 H H)
-- Definir la función
--    ramaIzquierda :: Arbol -> [Int]
-- tal que (ramaIzquierda a) es la lista de los valores de los nodos de
-- la rama izquierda del árbol a. Por ejemplo,
--    ghci> ramaIzquierda (N 9 (N 3 (N 2 H H) (N 4 H H)) (N 7 H H))
--    [9,3,2]
-- ---------------------------------------------------------------------

data Arbol = H
           | N Int Arbol Arbol

ramaIzquierda :: Arbol -> [Int]
ramaIzquierda H         = []
ramaIzquierda (N x i d) = x : ramaIzquierda i

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] Un primo permutable es un número primo tal
-- que todos los números obtenidos permutando sus cifras son primos. Por
-- ejemplo, 337 es un primo permutable ya que 337, 373 y 733 son
-- primos.
--
-- Definir la función
--    primoPermutable :: Integer -> Bool
-- tal que (primoPermutable x) se verifica si x es un primo
-- permutable. Por ejemplo,
--    primoPermutable 17  ==  True
--    primoPermutable 19  ==  False
-- ---------------------------------------------------------------------
```

```
primoPermutable :: Integer -> Bool
primoPermutable x = and [primo y | y <- permutacionesN x]

-- (permutacionesN x) es la lista de los números obtenidos permutando
-- las cifras de x. Por ejemplo,
permutacionesN :: Integer -> [Integer]
permutacionesN x = [read ys | ys <- permutaciones (show x)]

-- (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]]
intercala :: a -> [a] -> [[a]]
intercala x [] = [[x]]
intercala x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala x ys]

-- (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 []     = [[]]
permutaciones (x:xs) =
    concat [intercala x ys | ys <- permutaciones xs]

-- (primo x) se verifica si x es primo.
primo :: Integer -> Bool
primo x = x == head (dropWhile (<x) primos)

-- primos es la lista de los números primos.
primos :: [Integer ]
primos = criba [2..]
    where criba :: [Integer] -> [Integer]
          criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]
```

A.5. Examen 5 (21 de Marzo de 2012)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (21 de marzo de 2012)
-- ---------------------------------------------------------------------
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Dos números son equivalentes si la media de
-- sus cifras son iguales. Por ejemplo, 3205 y 41 son equvalentes ya que
-- (3+2+0+5)/4 = (4+1)/2. Definir la función
--    equivalentes :: Int -> Int -> Bool
-- tal que (equivalentes x y) se verifica si los números x e y son
-- equivalentes. Por ejemplo,
--    equivalentes 3205 41  ==  True
--    equivalentes 3205 25  ==  False
-- ---------------------------------------------------------------------

equivalentes :: Int -> Int -> Bool
equivalentes x y = media (cifras x) == media (cifras y)

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 3205  ==  [3,2,0,5]
cifras :: Int -> [Int]
cifras n = [read [y] | y <- show n]

-- (media xs) es la media de la lista xs. Por ejemplo,
--    media [3,2,0,5]  ==  2.5
media :: [Int] -> Float
media xs = (fromIntegral (sum xs)) / (fromIntegral (length xs))

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Definir la función
--    relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
--    relacionados (<) [2,3,7,9]                ==  True
--    relacionados (<) [2,3,1,9]                ==  False
--    relacionados equivalentes [3205,50,5014]  ==  True
-- ---------------------------------------------------------------------

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

-- Una definición alternativa es
relacionados' :: (a -> a -> Bool) -> [a] -> Bool
```

```
relacionados' r xs = and [r x y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Definir la función
--    primosEquivalentes :: Int -> [[Int]]
-- tal que (primosEquivalentes n) es la lista de las sucesiones de n
-- números primos consecutivos equivalentes. Por ejemplo,
--    take 2 (primosEquivalentes 2)  ==  [[523,541],[1069,1087]]
--    head (primosEquivalentes 3)    ==  [22193,22229,22247]
-- ---------------------------------------------------------------------

primosEquivalentes :: Int -> [[Int]]
primosEquivalentes n = aux primos
    where aux (x:xs) | relacionados equivalentes ys = ys : aux xs
                     | otherwise                    = aux xs
                     where ys = take n (x:xs)

-- primos es la lista de los números primos.
primos :: [Int]
primos = criba [2..]
    where criba :: [Int] -> [Int]
          criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] 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
```

A.6. Examen 6 (2 de Mayo de 2012)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (2 de mayo de 2012)
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número x es especial si el número de ocurrencia de
-- cada dígito d de x en x^2 es el doble del número de ocurrencia de d
-- en x. Por ejemplo, 72576 es especial porque tiene un 2, un 5, un 6 y
-- dos 7 y su cuadrado es 5267275776 que tiene exactamente dos 2, dos 5,
-- dos 6 y cuatro 7.
--
-- Definir la función
--    especial :: Integer -> Bool
-- tal que (especial x) se verifica si x es un número especial. Por
-- ejemplo,
--    especial 72576  ==  True
--    especial 12     ==  False
-- Calcular el menor número especial mayor que 72576.
-- ---------------------------------------------------------------------

especial :: Integer -> Bool
especial x =
    sort (ys ++ ys) == sort (show (x^2))
    where ys = show x

-- EL cálculo es
--    ghci> head [x | x <- [72577..], especial x]
--    406512

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    posiciones :: Eq a => a -> Array (Int,Int) a -> [(Int,Int)]
-- tal que (posiciones x p) es la lista de las posiciones de la matriz p
-- cuyo valor es x. Por ejemplo,
--    ghci> let p = listArray ((1,1),(2,3)) [1,2,3,2,4,6]
--    ghci> p
```

```
--    array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
--                         ((2,1),2),((2,2),4),((2,3),6)]
--    ghci> posiciones 2 p
--    [(1,2),(2,1)]
--    ghci> posiciones 6 p
--    [(2,3)]
--    ghci> posiciones 7 p
--    []
-- ---------------------------------------------------------------------

posiciones :: Eq a => a -> Array (Int,Int) a -> [(Int,Int)]
posiciones x p = [(i,j) | (i,j) <- indices p, p!(i,j) == x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. 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, ... de forma que las longitudes
-- de las lista del resultado sean iguales a la más corta de xss. 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 4. [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.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. 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 n = sucGolomb !! (n-1)

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

sucGolomb :: [Int]
sucGolomb = subSucGolomb 1

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. 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 1 = [1] ++ subSucGolomb 2
```

```
subSucGolomb 2 = [2,2] ++ subSucGolomb 3
subSucGolomb x = (replicate (golomb x) x) ++ subSucGolomb (x+1)

-- Nota: La sucesión de Golomb puede definirse de forma más compacta
-- como se muestra a continuación.
sucGolomb' :: [Int]
sucGolomb' = 1 : 2 : 2 : g 3
    where g x      = replicate (golomb x) x ++ g (x+1)
          golomb n = sucGolomb !! (n-1)
```