Ejercicios de "Informática de 1º de 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 18 de febrero de 2012)

Índice general

Introducción

Este libro es una recopilación de las soluciones de ejercicios de la asignatura de "Informática" (de 1º del Grado en Matemáticas) correspondientes al curso 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
```

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

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

```
-- 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

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

```
-- 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           == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
--    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
```

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

```
--    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)))
```

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
```