Ejercicios de “Informática de 1º de Matemáticas” (2018–19) (14 primeras relaciones)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 15 de enero de 2019

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 2018–19.

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].

El libro se irá ampliando conforme se vayan comentando las soluciones de los ejercicios. En esta versión, se incluyen las 5 primeras relaciones.

[1] http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/2018-19-IM-temas-PF.pdf
[2] http://www.glc.us.es/~jalonso/ejerciciosI1M2017G4

Relación 1

Definiciones por composición sobre números, listas y booleanos

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

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

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

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

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

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

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

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

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

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

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

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

ultimaCifra x = rem x 10

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

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

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

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

rota1 xs = tail xs ++ [head xs]

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

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

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

rango xs = [minimum xs, maximum xs]

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

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

palindromo xs = xs == reverse xs

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

interior xs = tail (init xs)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relación 2

Definiciones con condicionales, guardas o patrones

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

-- En esta relación se presentan ejercicios con definiciones elementales
-- (no recursivas) de funciones que usan condicionales, guardas o
-- patrones.
--
-- Estos ejercicios se corresponden con el tema 4 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-16/temas/tema-4.html

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
```

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    numeroMayor :: (Num a, Ord a) => a -> a -> a
-- tal que (numeroMayor x y) es el mayor número de dos cifras que puede
```

```haskell
-- construirse con los dígitos x e y. Por ejemplo,
--    numeroMayor 2 5 ==  52
--    numeroMayor 5 2 ==  52
-- ---------------------------------------------------------------------

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    raices :: Double -> Double -> Double -> [Double]
-- tal que (raices a b c) es la lista de las raíces reales de la
-- ecuación ax^2 + bx + c = 0. Por ejemplo,
```

```haskell
--    raices 1 3 2    ==  [-1.0,-2.0]
--    raices 1 (-2) 1 ==  [1.0,1.0]
--    raices 1 0 1    ==  []
-- Nota: Se supone que a es no nulo.
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Los intervalos cerrados se pueden representar mediante
-- una lista de dos números (el primero es el extremo inferior del
-- intervalo y el segundo el superior).
--
-- Definir la función
--    interseccion :: Ord a => [a] -> [a] -> [a]
-- tal que (interseccion i1 i2) es la intersección de los intervalos i1 e
```

```haskell
-- i2. Por ejemplo,
--    interseccion [] [3,5]     ==  []
--    interseccion [3,5] []     ==  []
--    interseccion [2,4] [6,9]  ==  []
--    interseccion [2,6] [6,9]  ==  [6,6]
--    interseccion [2,6] [0,9]  ==  [2,6]
--    interseccion [2,6] [0,4]  ==  [2,4]
--    interseccion [4,6] [0,4]  ==  [4,4]
--    interseccion [5,6] [0,4]  ==  []
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

Relación 3

Definiciones por comprensión

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

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

-- ---------------------------------------------------------------------
-- 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 4 7     ==  [7,7,7,7]
--    replica 3 True  ==  [True, True, True]
-- Nota: La función replica es equivalente a la predefinida replicate.
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    suma :: Integer -> Integer
-- tal (suma n) es la suma de los n primeros números. Por ejemplo,
--    suma 3  ==  6
-- ---------------------------------------------------------------------

suma :: Integer -> Integer
suma n = sum [1..n]

-- Otra definición más eficiente es
suma2 :: Integer -> Integer
suma2 n = (1+n)*n `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Los triángulos aritméticos se forman como sigue
--     1
--     2  3
--     4  5  6
--     7  8  9 10
--    11 12 13 14 15
--    16 17 18 19 20 21
--
-- Definir la función
--    linea :: Integer -> [Integer]
-- tal que (linea n) es la línea n-ésima de los triángulos
-- aritméticos. Por ejemplo,
--    linea 4  ==  [7,8,9,10]
--    linea 5  ==  [11,12,13,14,15]
-- ---------------------------------------------------------------------

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

-- La definición puede mejorarse
```

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

-- Una variante más eficiente es
linea3 :: Integer -> [Integer]
linea3 n = [s+1..s+n]
  where s = suma2 (n-1)

-- La mejora de la eficiencia se puede observar como sigue:
--    ghci> :set +s
--    ghci> head (linea 1000000)
--    499999500001
--    (17.94 secs, 309207420 bytes)
--    ghci> head (linea3 1000000)
--    499999500001
--    (0.01 secs, 525496 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    triangulo :: Integer -> [[Integer]]
-- 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 :: Integer -> [[Integer]]
triangulo n = [linea m | m <- [1..n]]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- La eficiencia puede mejorarse con
circulo2 :: Int -> Int
circulo2 n = length [(x,y) | x <- [0..n-1]
                           , y <- [0..raizCuadradaEntera (n*n - x*x)]
                           , x*x+y*y < n*n]

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

-- Comparación de eficiencia
--    λ> circulo (10^4)
--    78549754
--    (73.44 secs, 44,350,688,480 bytes)
--    λ> circulo2 (10^4)
--    78549754
```

```
--    (59.71 secs, 36,457,043,240 bytes)

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Definir la función
--    errorAproxE :: Double -> Double
-- tal que (errorE x) es el menor número de términos de la sucesión
-- (1+1/m)**m necesarios para obtener su límite con un error menor que
-- x. Por ejemplo,
--    errorAproxE 0.1    ==  13.0
--    errorAproxE 0.01   ==  135.0
--    errorAproxE 0.001  ==  1359.0
-- Indicación: En Haskell, e se calcula como (exp 1).
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir la función
--    aproxLimSeno :: Double -> [Double]
-- tal que (aproxLimSeno n) es la lista cuyos elementos son los términos
-- de la sucesión
```

```haskell
--    sen(1/m)
--    --------
--      1/m
-- desde 1 hasta n. Por ejemplo,
--    aproxLimSeno 1 == [0.8414709848078965]
--    aproxLimSeno 2 == [0.8414709848078965,0.958851077208406]
-- ---------------------------------------------------------------------

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

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

-- El límite es 1.

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Definir la función
--    errorLimSeno :: Double -> Double
-- tal que (errorLimSeno x) es el menor número de términos de la sucesión
-- sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
-- que x. Por ejemplo,
--    errorLimSeno 0.1     ==   2.0
--    errorLimSeno 0.01    ==   5.0
--    errorLimSeno 0.001   ==  13.0
--    errorLimSeno 0.0001  ==  41.0
-- ---------------------------------------------------------------------

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

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

```haskell
--    calculaPi 300  ==  3.1449149035588526
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Una terna (x,y,z) de enteros positivos es pitagórica
-- si x^2 + y^2 = z^2.
--
-- Definir, por comprensión, la función
--    pitagoricas :: Int -> [(Int,Int,Int)]
-- tal que (pitagoricas n) es la lista de todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n. Por ejemplo,
--    pitagoricas 10  ==  [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. Definir la función
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.3. Definir la función
--    conjetura :: Int -> Bool
-- tal que (conjetura n) se verifica si todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n tiene un número impar de números
-- pares. Por ejemplo,
--    conjetura 10  ==  True
-- ---------------------------------------------------------------------

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

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

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

```
-- Caso 4: x e y son impares. Entonces, x^2 e y^2 también son impares y
-- z^2 es par. Luego el número de componentes pares es 1 que es impar.

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

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

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

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

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 13. El producto escalar de dos listas de enteros xs y ys de
-- longitud n viene dado por la suma de los productos de los elementos
-- correspondientes.
--
-- Definir por comprensión la función
--    productoEscalar :: [Int] -> [Int] -> Int
-- tal que (productoEscalar xs ys) es el producto escalar de las listas
-- xs e ys. Por ejemplo,
--    productoEscalar [1,2,3] [4,5,6]  ==  32
-- ---------------------------------------------------------------------

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

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

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

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

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

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

personas :: [(String,String,Int,Int)]
personas = [("Cervantes","Literatura",1547,1616),
            ("Velazquez","Pintura",1599,1660),
            ("Picasso","Pintura",1881,1973),
            ("Beethoven","Musica",1770,1823),
            ("Poincare","Ciencia",1854,1912),
            ("Quevedo","Literatura",1580,1654),
            ("Goya","Pintura",1746,1828),
            ("Einstein","Ciencia",1879,1955),
            ("Mozart","Musica",1756,1791),
            ("Botticelli","Pintura",1445,1510),
            ("Borromini","Arquitectura",1599,1667),
            ("Bach","Musica",1685,1750)]

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 16.5. Definir la función
--    vivas :: [(String,String,Int,Int)] -> Int -> [String]
```

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

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

Relación 4

Definiciones por recursión

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

-- En esta relación se presentan ejercicios con definiciones por
-- recursión correspondientes al tema 6 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-6.html

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

import Test.QuickCheck

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Teniendo en cuenta que buscamos el máximo común
-- divisor de a y b, sería razonable pensar que el máximo común divisor
-- siempre sería igual o menor que la mitad del máximo de a y b. Definir
-- esta propiedad y comprobarla.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd_div :: Integer -> Integer -> Property
prop_mcd_div a b =
  a > 0 && b > 0 ==> mcd a b <= max a b `div` 2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    sumaDigitosR n == sumaDigitosNR n

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relación 5

Operaciones conjuntistas con listas

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

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

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

import Test.QuickCheck

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

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

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

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

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

comprueba_subconjuntoR :: IO ()
comprueba_subconjuntoR =
  quickCheck prop_subconjuntoR

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```haskell
-- 2ª definición (por recursión):
productoR :: [a] -> [a] -> [(a,a)]
productoR []     _  = []
productoR (x:xs) ys = [(x,y) | y <- ys] ++ productoR xs ys

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

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

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

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

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

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

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

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

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

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

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

Relación 6

Funciones sobre cadenas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por recursión, la función
--    tituloRec :: [String] -> [String]
-- tal que (tituloRec ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--    ghci> tituloRec ["eL","arTE","DE","La","proGraMacion"]
--    ["El","Arte","de","la","Programacion"]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relación 7

Funciones de orden superior y definiciones por plegados

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

-- Esta relación tiene contiene ejercicios con funciones de orden
-- superior y definiciones por plegado correspondientes al tema 7
-- http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-7.html

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

import Test.QuickCheck

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

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

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

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

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

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

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

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

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

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

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

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

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

comprueba_agrupa :: IO ()
comprueba_agrupa =
  quickCheck prop_agrupa

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, usando foldr, la función
--    concatP :: [[a]] -> [a]
-- tal que (concatP xss) es la concatenación de las listas de xss. Por
```

```haskell
-- ejemplo,
--    concatP [[1,3],[2,4,6],[1,9]]  ==  [1,3,2,4,6,1,9]
-- ----------------------------------------------------------------------

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

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

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

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

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

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

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

-- ----------------------------------------------------------------------
-- Ejercicio 5.1. Definir, por comprensión, la función
--    filtraAplicaC :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaC f p xs) es la lista obtenida aplicándole a los
```

```haskell
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaC (4+) (<3) [1..7]  =>  [5,6]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

filtraAplicaP :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

```haskell
filtraAplicaP f p = foldr g []
    where g x y | p x       = f x : y
                | otherwise = y

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

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

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

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

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

Relación 8

Tipos de datos algebraicos: Árboles binarios

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

-- En esta relación se presenta ejercicios sobre árboles binarios
-- definidos como tipos de datos algebraicos.
--
-- Los ejercicios corresponden al tema 9 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-9.html

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

import Test.QuickCheck
import Control.Monad

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-- ---------------------------------------------------------------------

-- La propiedad es
```

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que para todo árbol biario
-- x, se tiene que
--    nNodos x <= 2^(profundidad x) - 1
-- ---------------------------------------------------------------------

-- La propiedad es
prop_nNodosProfundidad :: Arbol Int -> Bool
prop_nNodosProfundidad x =
   nNodos x <= 2 ^ profundidad x - 1

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

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

```haskell
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--    preorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [9,3,2,4,7]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

```haskell
-- Ejercicio 3.4. Definir, usando un acumulador, la función
--    preordenIt :: Arbol a -> [a]
-- tal que (preordenIt x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--    preordenIt (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [9,3,2,4,7]
--
-- Nota: No usar (++) en la definición
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Comprobar con QuickCheck que preordenIt es equivalente
-- a preorden.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_preordenIt :: Arbol Int -> Bool
prop_preordenIt x =
    preordenIt x == preorden x

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que para todo árbol binario
-- x, se tiene que
--    reverse (preorden (espejo x)) = postorden x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_reverse_preorden_espejo :: Arbol Int -> Bool
prop_reverse_preorden_espejo x =
  reverse (preorden (espejo x)) == postorden x

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Comprobar con QuickCheck que para todo árbol x,
--    postorden (espejo x) = reverse (preorden x)
-- ---------------------------------------------------------------------

-- La propiedad es
prop_recorrido :: Arbol Int -> Bool
prop_recorrido x =
  postorden (espejo x) == reverse (preorden x)
```

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

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

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

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

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

-- La comprobación es
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. La función
--    repeat :: a -> [a]
-- está definida de forma que (repeat x) es la lista formada por
-- infinitos elementos x. Por ejemplo,
--    repeat 3  ==  [3,3,3,3,3,3,3,3,3,3,3,3,...
-- La definición de repeat es
--    repeat x = xs where xs = x:xs
--
-- Definir la función
--    repeatArbol :: a -> Arbol a
-- tal que (repeatArbol x) es es árbol con infinitos nodos x. Por
-- ejemplo,
--    takeArbol 0 (repeatArbol 3) == H 3
--    takeArbol 1 (repeatArbol 3) == N 3 (H 3) (H 3)
--    takeArbol 2 (repeatArbol 3) == N 3 (N 3 (H 3) (H 3)) (N 3 (H 3) (H 3))
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Nota. Para comprobar propiedades de árboles con QuickCheck se
-- utilizará el siguiente generador.
-- ---------------------------------------------------------------------

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

Relación 9

Tipos de datos algebraicos

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- 1ª definición
-- =============
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

data Op = Su | Re | Mu

data Expr5 = C5 Int | A Op Expr5 Expr5

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

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

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

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

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

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

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

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

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

Relación 10

Listas infinitas y evaluación perezosa

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

-- En esta relación se presentan ejercicios con listas infinitas y
-- evaluación perezosa. Estos ejercicios corresponden al tema 10 que
-- se encuentra en
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-10.html

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    repite :: a -> [a]
-- tal que (repite x) es la lista infinita cuyos elementos son x. Por
-- ejemplo,
--    repite 5           ==  [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
--    take 3 (repite 5)  ==  [5,5,5]
--
-- Nota: La función repite es equivalente a la función repeat definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición:
repite1 :: a -> [a]
repite1 x = x : repite1 x

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- La propiedad es
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir, por comprensión, la función
--    ecoC :: String -> String
-- tal que (ecoC xs) es la cadena obtenida a partir de la cadena xs
-- repitiendo cada elemento tantas veces como indica su posición: el
-- primer elemento se repite 1 vez, el segundo 2 veces y así
-- sucesivamente. Por ejemplo,
--    ecoC "abcd"  ==  "abbcccdddd"
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por recursión, la función
--    ecoR :: String -> String
-- tal que (ecoR xs) es la cadena obtenida a partir de la cadena xs
-- repitiendo cada elemento tantas veces como indica su posición: el
-- primer elemento se repite 1 vez, el segundo 2 veces y así
-- sucesivamente. Por ejemplo,
--    ecoR "abcd"  ==  "abbcccdddd"
-- ---------------------------------------------------------------------

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por recursión, la función
--    itera :: (a -> a) -> a -> [a]
-- tal que (itera f x) es la lista cuyo primer elemento es x y los
-- siguientes elementos se calculan aplicando la función f al elemento
-- anterior. Por ejemplo,
--    ghci> itera (+1) 3
--    [3,4,5,6,7,8,9,10,11,12,{Interrupted!}
--    ghci> itera (*2) 1
--    [1,2,4,8,16,32,64,{Interrupted!}
--    ghci> itera (`div` 10) 1972
--    [1972,197,19,1,0,0,0,0,0,0,{Interrupted!}
--
-- Nota: La función itera es equivalente a la función iterate definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, por recursión, la función
--    agrupaR :: Int -> [a] -> [[a]]
-- tal que (agrupaR n xs) es la lista formada por listas de n elementos
-- consecutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--    ghci> agrupaR 2 [3,1,5,8,2,7]
--    [[3,1],[5,8],[2,7]]
--    ghci> agrupaR 2 [3,1,5,8,2,7,9]
--    [[3,1],[5,8],[2,7],[9]]
--    ghci> agrupaR 5 "todo necio confunde valor y precio"
--    ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-- ---------------------------------------------------------------------

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

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

```haskell
-- Ejercicio 5.2. Definir, de manera no recursiva con iterate, la función
--    agrupa :: Int -> [a] -> [[a]]
-- tal que (agrupa n xs) es la lista formada por listas de n elementos
-- consecutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--    ghci> agrupa 2 [3,1,5,8,2,7]
--    [[3,1],[5,8],[2,7]]
--    ghci> agrupa 2 [3,1,5,8,2,7,9]
--    [[3,1],[5,8],[2,7],[9]]
--    ghci> agrupa 5 "todo necio confunde valor y precio"
--    ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-- ---------------------------------------------------------------------

agrupa :: Int -> [a] -> [[a]]
agrupa n = takeWhile (not . null)
         . map (take n)
         . iterate (drop n)

-- Puede verse su funcionamiento en el siguiente ejemplo,
--    iterate (drop 2) [5..10]
--    ==> [[5,6,7,8,9,10],[7,8,9,10],[9,10],[],[],...
--    map (take 2) (iterate (drop 2) [5..10])
--    ==> [[5,6],[7,8],[9,10],[],[],[],[],...
--    takeWhile (not . null) (map (take 2) (iterate (drop 2) [5..10]))
--    ==> [[5,6],[7,8],[9,10]]

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Sea la siguiente operación, aplicable a cualquier
-- número entero positivo:
--    * Si el número es par, se divide entre 2.
--    * Si el número es impar, se multiplica por 3 y se suma 1.
-- Dado un número cualquiera, podemos considerar su órbita, es decir,
-- las imágenes sucesivas al iterar la función. Por ejemplo, la órbita
-- de 13 es
--    13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1,...
-- Si observamos este ejemplo, la órbita de 13 es periódica, es decir,
-- se repite indefinidamente a partir de un momento dado). La conjetura
-- de Collatz dice que siempre alcanzaremos el 1 para cualquier número
-- con el que comencemos. Ejemplos:
--    * Empezando en n = 6 se obtiene 6, 3, 10, 5, 16, 8, 4, 2, 1.
--    * Empezando en n = 11 se obtiene: 11, 34, 17, 52, 26, 13, 40, 20,
--      10, 5, 16, 8, 4, 2, 1.
--    * Empezando en n = 27, la sucesión tiene 112 pasos, llegando hasta
--      9232 antes de descender a 1:  27, 82, 41, 124, 62, 31, 94, 47,
--      142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274,
--      137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263,
--      790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502,
--      251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
```

```
--      479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644,
--      1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308,
--      1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
--      61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,
--      16, 8, 4, 2, 1.
--
-- Definir la función
--    siguiente :: Integer -> Integer
-- tal que (siguiente n) es el siguiente de n en la sucesión de
-- Collatz. Por ejemplo,
--    siguiente 13  ==  40
--    siguiente 40  ==  20
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir, sin recursión y con iterate, la función
--    collatz :: Integer -> [Integer]
-- tal que (collatz n) es la órbita de Collatz d n hasta alcanzar el
-- 1. Por ejemplo,
--    collatz 13  ==  [13,40,20,10,5,16,8,4,2,1]
-- Indicación: Usar takeWhile e iterate.
-- ---------------------------------------------------------------------

collatz :: Integer -> [Integer]
```

```haskell
collatz n = takeWhile (/=1) (iterate siguiente n) ++ [1]

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.5. Definir la función
--    menorCollatzSupera :: Integer -> Integer
-- tal que (menorCollatzSupera x) es el menor número cuya órbita de
-- Collatz tiene algún elemento mayor que x. Por ejemplo,
--    menorCollatzSupera 100  ==  15
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir, usando takeWhile y map, la función
--    potenciasMenores :: Int -> Int -> [Int]
-- tal que (potenciasMenores x y) es la lista de las potencias de x
-- menores que y. Por ejemplo,
--    potenciasMenores 2 1000  ==  [2,4,8,16,32,64,128,256,512]
-- ---------------------------------------------------------------------

potenciasMenores :: Int -> Int -> [Int]
```

```haskell
potenciasMenores x y = takeWhile (<y) (map (x^) [1..])

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

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

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

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

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

sumaDeDosPrimos :: Int -> [(Int,Int)]
sumaDeDosPrimos n =
  [(x,n-x) | x <- primosN, primo (n-x)]
```

```
  where primosN = takeWhile (<= (n `div` 2)) primos

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

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

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

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

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

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

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

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

```
--    = 1 : take 3 (1 : zipWith (*) [2..] [R1])
--    = 1 : 1 : take 2 (zipWith (*) [2..] [1|R2])       {R2 es drop 2 factoriales
--    = 1 : 1 : take 2 (2 : zipWith (*) [3..] [R2])
--    = 1 : 1 : 2 : take 1 (zipWith (*) [3..] [2|R3])    {R3 es drop 3 factoriale
--    = 1 : 1 : 2 : take 1 (6 : zipWith (*) [4..] [R3])
--    = 1 : 1 : 2 : 6 : take 0 (zipWith (*) [4..] [R3])
--    = 1 : 1 : 2 : 6 : []
--    = [1, 1, 2, 6]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
--    0
--    (0.03 secs, 11965760 bytes)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. El triángulo de Pascal es un triángulo de números
--          1
--         1 1
--        1 2 1
--       1  3 3  1
--      1 4  6  4 1
--    1 5 10 10 5 1
--   ..............
-- construido de la siguiente forma
-- + la primera fila está formada por el número 1;
-- + las filas siguientes se construyen sumando los números adyacentes
--   de la fila superior y añadiendo un 1 al principio y al final de la
--   fila.
--
-- Definir la función
--    pascal1 :: [[Integer]]
-- tal que pascal es la lista de las líneas del triángulo de Pascal. Por
-- ejemplo,
--    ghci> take 6 pascal1
--    [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

```
--  = [1] : [1,1] : [1,2,1] : (take 1 (map f R2))
--  = [1] : [1,1] : [1,2,1] : (take 1 (map f ([1,2,1]:R3)))
--  = [1] : [1,1] : [1,2,1] : (take 1 ((f [1,2,1]) : map f R3))
--  = [1] : [1,1] : [1,2,1] : (take 1 ((zipWith (+) (0:[1,2,1]) ([1,2,1]++[0]))
--                                     : map f R3))
--  = [1] : [1,1] : [1,2,1] : (take 1 ((zipWith (+) [0,1,2,1] [1,2,1,0])
--                                     : map f R3)))
--  = [1] : [1,1] : [1,2,1] : (take 1 ([1,3,3,1] : map f R3)))
--  = [1] : [1,1] : [1,2,1] : [1,3,3,1] : (take 0 (map f R3)))
--  = [1] : [1,1] : [1,2,1] : [1,3,3,1] : []
--  = [[1],[1,1],[1,2,1],[1,3,3,1]]
-- en el cálculo con R1, R2pascal y R3 es el triángulo de
-- Pascal sin el primero, los dos primeros o los tres primeros elementos,
-- respectivamente.
```

Relación 11

Aplicaciones de la programación funcional con listas infinitas

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

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

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

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

-- 1ª definición
enteros :: [Int]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```haskell
sucGolomb :: [Int]
sucGolomb = subSucGolomb 1

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

subSucGolomb :: Int -> [Int]
subSucGolomb 1 = 1 : subSucGolomb 2
subSucGolomb 2 = [2,2] ++ subSucGolomb 3
subSucGolomb x = replicate (golomb x) x ++ subSucGolomb (x+1)

-- Nota: La sucesión de Golomb puede definirse de forma más compacta
-- como se muestra a continuación.
sucGolomb2 :: [Int]
sucGolomb2 = 1 : 2 : 2 : g 3
    where g x      = replicate (golomb2 x) x ++ g (x+1)
          golomb2 n = sucGolomb !! (n-1)

sucGolomb3 :: [Int]
sucGolomb3 = 1 : 2 : 2 :
             concat [replicate n k | (n,k) <-zip (drop 2 sucGolomb3) [3..]]
```

Relación 12

El 2019 es feliz

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

-- Según la Wikipedia (en http://bit.ly/2RBGEnz), un número feliz se
-- define por el siguiente proceso. Se comienza reemplazando el número
-- por la suma del cuadrado de sus dígitos y se repite el proceso hasta
-- que se obtiene el número 1 o se entra en un ciclo que no contiene al
-- 1. Aquellos números para los que el proceso termina en 1 se llaman
-- números felices y los que entran en un ciclo sin 1 se llaman números
-- desgraciados.
--
-- Por ejemplo, 2019 es un número feliz porque
--    2019 ~> 2² + 0² + 1² + 9² =  4 +  1 + 81 = 86
--         ~> 8² + 6²           = 64 + 36      = 100
--         ~> 1² + 0² + 0²                     = 1
-- Pero 17 es un número desgraciado porque
--    17 ~> 1² + 7²      =  1 + 49      =  50
--       ~> 5² + 0²      = 25 +  0      =  25
--       ~> 2² + 5²      =  4 + 25      =  29
--       ~> 2² + 9²      =  4 + 81      =  85
--       ~> 8² + 5²      = 64 + 25      =  89
--       ~> 8² + 9²      = 64 + 81      = 145
--       ~> 1² + 4² + 5² =  1 + 16 + 25 =  42
--       ~> 4² + 2²      = 16 +  4      =  20
--       ~> 2² + 0²      =  4 +  0      =   4
--       ~> 4²                          =  16
--       ~> 1² + 6²      =  1 + 36      =  37
```

```
--       ~> 3² + 7²      =  9 + 49      =  58
--       ~> 5² + 8²      = 25 + 64      =  89
-- que forma un bucle al repetirse el 89.
--
-- El objetivo del ejercicio es definir una función que calcule todos
-- los números felices hasta un límite dado y calcular la posición del
-- 2019 en dicha sucesión.

-- ---------------------------------------------------------------------
-- § Librerias auxiliares                                              --
-- ---------------------------------------------------------------------

import Data.List
import Data.Char
import Test.QuickCheck
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    digitos :: Int -> [Int]
-- tal que (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 325  ==  [3,2,5]
-- ---------------------------------------------------------------------

-- 1ª definición
digitos :: Int -> [Int]
digitos n = [read [x] | x <- show n]

-- 2ª definición
digitos2 :: Int -> [Int]
digitos2 = map digitToInt . show

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    caminoALaFelicidad :: Int -> [Int]
-- tal que (caminoALaFelicidad n) es la lista de los números obtenidos
-- en el proceso de la determinación si n es un número feliz: se
-- comienza con la lista [n], ampliando la lista con la suma del
-- cuadrado de los dígitos de su primer elemento y se repite el proceso
-- hasta que se obtiene el número 1 o se entra en un ciclo que no
```

```haskell
-- contiene al 1. Por ejemplo,
--    λ> caminoALaFelicidad 2019
--    [1,100,86,2019]
--    λ> caminoALaFelicidad 17
--    [89,58,37,16,4,20,42,145,89,85,29,25,50,17]
-- ---------------------------------------------------------------------

caminoALaFelicidad :: Int -> [Int]
caminoALaFelicidad n = f [n]
  where f vs@(x:xs) | x == 1      = vs
                    | x `elem` xs = vs
                    | otherwise   = f (sum [y^2 | y <- digitos x] : vs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    esFeliz :: Int -> Bool
-- tal que (esFeliz n) se verifica si n es un número feliz. Por ejemplo,
--    esFeliz 2019  ==  True
--    esFeliz 17    ==  False
-- ---------------------------------------------------------------------

esFeliz :: Int -> Bool
esFeliz n = head (caminoALaFelicidad n) == 1

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la lista
--    numerosFelices :: [Int]
-- cuyos elementos son los números felices. Por ejemplo,
--    take 10 numerosFelices  ==  [1,7,10,13,19,23,28,31,32,44]
-- ---------------------------------------------------------------------

-- 1ª definición
numerosFelices :: [Int]
numerosFelices = [x | x <- [1..], esFeliz x]

-- 2ª definición
numerosFelices2 :: [Int]
numerosFelices2 = filter esFeliz [1..]

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

```haskell
-- Ejercicio 5. Definir la función
--    posicionFeliz :: Int -> Maybe Int
-- tal que (posicionFeliz n) es justo la posición de n en la sucesión de
-- los números felices, si n es un número feliz y Nothing, en caso
-- contrario. Por ejemplo,
--    posicionFeliz 2019    ==  Just 300
--    posicionFeliz 17      ==  Nothing
--    posicionFeliz (10^5)  ==  Just 14376
-- ---------------------------------------------------------------------

-- 1ª definición
posicionFeliz :: Int -> Maybe Int
posicionFeliz n
  | y == n    = Just (length xs)
  | otherwise = Nothing
  where (xs,y:_) = span (<n) numerosFelices

-- 2ª definición
posicionFeliz2 :: Int -> Maybe Int
posicionFeliz2 n
  | esFeliz n = elemIndex n numerosFelices
  | otherwise = Nothing

-- Comparación de eficiencia
--    λ> posicionFeliz (10^5)
--    Just 14376
--    (6.96 secs, 12,308,572,216 bytes)
--    λ> posicionFeliz2 (10^5)
--    Just 14376
--    (0.01 secs, 424,584 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Comprobar con QuickChek que existen infinitos números
-- primos felices; es decir, que para cada número natural n existen
-- números primos mayores que n.
--
-- Nota: La demostración de la infinitud de los primos felices es un
-- problema abierto.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_primosFelices :: Int -> Property
prop_primosFelices n =
  n >= 0 ==>
  not (null [x | x <- primes, x > n])

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

-- Otra forma de expresar la propiedad es
prop_primosFelices2 :: Positive Int -> Bool
prop_primosFelices2 (Positive n) =
  not (null [x | x <- primes, x > n, esFeliz x])
```

Relación 13

El juego del nim y las funciones de entrada/salida

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

-- En el juego del nim el tablero tiene 5 filas numeradas de estrellas,
-- cuyo contenido inicial es el siguiente
--    1: *****
--    2: ****
--    3: ***
--    4: **
--    5: *
-- Dos jugadores retiran por turno una o más estrellas de una fila. El
-- ganador es el jugador que retire la última estrella. En este
-- ejercicio se va implementar el juego del Nim para practicar con las
-- funciones de entrada y salida estudiadas en el tema 13 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-18/temas/tema-13.html
--
-- Nota: El juego debe de ejecutarse en una consola, no en la shell de
-- emacs.

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

import Data.Char
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

escribeTablero :: Tablero -> IO ()
escribeTablero t =
  sequence_ [escribeFila n (t!!(n-1)) | n <- [1..length t]]

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

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

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

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

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

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

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

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

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

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

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

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

Relación 14

Cálculo del número pi mediante el método de Montecarlo

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

-- El objetivo de esta relación de ejercicios es el uso de los números
-- aleatorios para calcular el número pi mediante el método de
-- Montecarlo. Un ejemplo del método se puede leer en el artículo de
-- Pablo Rodríguez "Calculando Pi con gotas de lluvia" que se encuentra
-- en http://bit.ly/1cNfSR0

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

import System.Random
import System.IO.Unsafe

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    aleatorio :: Random t => t -> t -> t
-- tal que (aleatorio a b) es un número aleatorio entre a y b. Por
-- ejemplo,
--    ghci> aleatorio 0 1000
--    681
--    ghci> aleatorio 0 1000
--    66
```

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

aleatorio :: Random t => t -> t -> t
aleatorio a b = unsafePerformIO $
                getStdRandom (randomR (a,b))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    aleatorios :: Random t => t -> t -> [t]
-- (aleatorios m n) es una lista infinita de números aleatorios entre m y
-- n. Por ejemplo,
--    ghci> take 20 (aleatorios 2 9)
--    [6,5,3,9,6,3,6,6,2,7,9,6,8,6,2,4,2,6,9,4]
--    ghci> take 20 (aleatorios 2 9)
--    [3,7,7,5,7,7,5,8,6,4,7,2,8,8,2,8,7,6,5,5]
-- ---------------------------------------------------------------------

aleatorios :: Random t => t -> t -> [t]
aleatorios m n = aleatorio m n : aleatorios m n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    puntosDelCuadrado :: [(Double,Double)]
-- tal que puntosDelCuadrado es una lista infinita de puntos del
-- cuadrado de vértices opuestos (-1,-1) y (1,1). Por ejemplo,
--    ghci> take 3 puntosDelCuadrado
--    [(0.5389481918223398,0.9385662370820778),
--     (-0.419123718392838,0.9982440984579455),
--     (0.5610432040657063,-0.7648360614536891)]
-- ---------------------------------------------------------------------

puntosDelCuadrado :: [(Double,Double)]
puntosDelCuadrado = zip (aleatorios (-1) 1) (aleatorios (-1) 1)

puntosDelCuadrado2 :: Int -> [(Double,Double)]
puntosDelCuadrado2 n =
  take n (zip (aleatorios (-1) 1) (aleatorios (-1) 1))


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

```haskell
-- Ejercicio 4. Definir la función
--    puntosEnElCirculo :: [(Double,Double)] -> Int
-- tal que (puntosEnElCirculo xs) es el número de puntos de la lista xs
-- que están en el círculo de centro (0,0) y radio 1.
--    ghci> puntosEnElCirculo [(1,0), (0.5,0.9), (0.2,-0.3)]
--    2
-- ---------------------------------------------------------------------

puntosEnElCirculo :: [(Double,Double)] -> Int
puntosEnElCirculo xs = length [(x,y) | (x,y) <- xs
                                     , x^2+y^2 <= 1]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    calculoDePi :: Int -> Double
-- tal que (calculoDePi n) es el cálculo del número pi usando n puntos
-- aleatorios (la probabilidad de que estén en el círculo es pi/4). Por
-- ejemplo,
--    ghci> calculoDePi 1000
--    3.076
--    ghci> calculoDePi 10000
--    3.11
--    ghci> calculoDePi 100000
--    3.13484
-- ---------------------------------------------------------------------

calculoDePi :: Int -> Double
calculoDePi n = 4 * enCirculo / total
    where xs        = take n puntosDelCuadrado
          enCirculo = fromIntegral (puntosEnElCirculo xs)
          total     = fromIntegral n
```