

Ejercicios de “Informática de 1º de Matemáticas” (2014–15)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 12 de junio de 2015

Esta obra está bajo una licencia Reconocimiento–NoComercial–CompartirIgual 2.5 Spain de Creative Commons.

Se permite:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor.



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Índice general

1	Definiciones por composición sobre números, listas y booleanos	7
2	Definiciones con condicionales, guardas o patrones	13
3	Definiciones por comprensión	25
4	Definiciones por comprensión con cadenas: El cifrado César	41
5	Definiciones por recursión	51
6	Ordenación por mezcla	67
7	Funciones sobre cadenas	71
8	Funciones de orden superior y definiciones por plegados	79
9	Listas infinitas y evaluación perezosa	93
10	Tipos de datos algebraicos	115
	10.1 Ejercicios propuestos	115
	10.2 Ejercicios de exámenes	125
11	Programación de dibujos y animaciones con Haskell y Gloss	159
12	Combinatoria	161
13	El TAD de las pilas	175
14	El TAD de las colas	185
15	Vectores y matrices	195
	15.1 Ejercicios propuestos	195
	15.2 Ejercicios de exámenes	214

16 Cálculo numérico	249
17 Operaciones con el TAD de polinomios	259
18 División y factorización de polinomios mediante la regla de Ruffini	269
19 Operaciones con conjuntos	277
20 Relaciones binarias homogéneas	295
21 Ecuaciones con factoriales	303
22 Vectores y matrices con las librerías	307
23 Problemas básicos con el TAD de los grafos	325
23.1 Ejercicios propuestos	325
23.2 Ejercicios de exámenes	337
24 Implementación del TAD de los grafos mediante listas	345
25 Enumeraciones de los números racionales	351
26 Operaciones con el TAD de montículos	359
27 Operaciones con conjuntos usando la librería Data.Set	367
28 Relaciones binarias homogéneas con la librería Data.Set	377
29 Algoritmos de ordenación y complejidad	387
30 El problema del granjero mediante búsqueda en espacio de estado	401
31 El problema del calendario mediante búsqueda en espacio de estado	407
32 Estadística descriptiva	411
33 Números de Lychrel	419
34 Resolución de problemas mediante búsqueda en espacios de estados	425
35 Rompecabeza del triominó mediante divide y vencerás	431
36 Analizadores sintácticos	441
37 El juego del Nim y las funciones de entrada/salida	451

38 Cálculo del número pi mediante el método de Montecarlo	461
39 Demostración de propiedades de programas por inducción sobre números y listas	465
40 Demostración de propiedades de programas por inducción sobre árboles	487
41 Mayorías parlamentarias	501

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 2014-15.

El objetivo de los ejercicios es complementar la introducción a la programación funcional y a la algorítmica con Haskell presentada en los temas del curso. Los apuntes de los temas se encuentran en [Temas de “Programación funcional”](#) ¹.

Los ejercicios sigue el orden de las relaciones de problemas propuestos durante el curso y, resueltos de manera colaborativa, en la wiki.

¹<http://www.cs.us.es/~jalonso/cursos/i1m-14/temas/2014-15-IM-temas-PF.pdf>

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/ZwSMH0  
  
-----  
-- 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
-- verifica si xs es un palíndromo; es decir, es lo mismo leer xs de
-- izquierda a derecha que de derecha a izquierda. Por ejemplo,
--   palindromo [3,2,5,2,3]    == True
--   palindromo [3,2,5,6,2,3] == False
-----
```

```
palindromo xs = xs == reverse xs
```

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

```
interior xs = tail (init xs)
```

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

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

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

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

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

```
-- elementos de xs. Por ejemplo,  
--   extremos 3 [2,6,7,1,2,4,5,8,9,2,3] == [2,6,7,9,2,3]
```

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

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

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

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

```
-----  
tresIguales x y z = x == y && y == z
```

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

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

```
-----  
-- Ejercicio 18. Definir la función cuatroIguales tal que
```

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

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

Relación 2

Definiciones con condicionales, guardas o patrones

```
-- -----  
-- 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 cuyas transparencias  
-- se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-4t.pdf  
  
-- -----  
-- Librerías auxiliares --  
-- -----  
  
import Test.QuickCheck  
  
-- -----  
-- 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.  
-----
```

```
-- 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 2.5. Comprobar con QuickCheck que las cuatro definiciones
```

```
-- de xor son equivalentes.  
-----
```

```
-- La propiedad es
```

```
prop_xor_equivalentes :: Bool -> Bool -> Bool
```

```
prop_xor_equivalentes x y =
```

```
  xor1 x y == xor2 x y &&
```

```
  xor2 x y == xor3 x y &&
```

```
  xor3 x y == xor4 x y
```

```

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

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

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

-----
-- Ejercicio 4.2. Comprobar con QuickCheck que la función intercambia es
-- idempotente; es decir, si se aplica dos veces es lo mismo que no
-- aplicarla ninguna.
-----

```

```

-- La propiedad es
prop_intercambia :: (Int,Int) -> Bool
prop_intercambia p = intercambia (intercambia p) == p

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

-----

-- Ejercicio 5.1. 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 5.2. Comprobar con QuickCheck que se verifica la propiedad
-- triangular de la distancia; es decir, dados tres puntos p1, p2 y p3,
-- la distancia de p1 a p3 es menor o igual que la suma de la distancia
-- de p1 a p2 y la de p2 a p3.
-----

-- La propiedad es
prop_triangular :: (Double,Double) -> (Double,Double) -> (Double,Double)
-> Bool
prop_triangular p1 p2 p3 =
  distancia p1 p3 <= distancia p1 p2 + distancia p2 p3

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

-----

-- Ejercicio 6.1. 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 6.2. Comprobar que la longitud es un invariante de la
-- función ciclo; es decir, la longitud de (ciclo xs) es la misma que la
-- de xs.
-----

-- La propiedad es
prop_ciclo :: [Int] -> Bool
prop_ciclo xs = length (ciclo xs) == length xs

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

-----

-- 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
-- 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 :: (Num 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
-----
```

```
numeroDeRaices :: (Num t, Ord t) => t -> t -> t -> Int
numeroDeRaices a b c | d < 0      = 0
                      | d == 0    = 1
                      | otherwise = 2
  where d = b^2-4*a*c
```

```
-----
-- Ejercicio 9.1. 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,
-- raices 1 3 2 == [-1.0,-2.0]
-- raices 1 (-2) 1 == [1.0,1.0]
-- raices 1 0 1 == []
-----
```

```
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 9.2. Definir el operador
-- (~=) :: (Fractional a, Ord a) => a -> a -> Bool
```

```
-- tal que (x ~= y) se verifica si x e y son casi iguales; es decir si
-- el valor absoluto de su diferencia es menor que una milésima. Por
-- ejemplo,
--     12.3457 ~= 12.3459 == True
--     12.3457 ~= 12.3479 == False
```

```
-----
(~==) :: (Fractional a, Ord a) => a -> a -> Bool
x ~= y = abs (x-y) < 0.001
```

```
-----
-- Ejercicio 9.3. Comprobar con QuickCheck que la suma de las raíces
-- de la ecuación  $ax^2 + bx + c = 0$  (con a no nulo) es  $-b/a$  y su
-- producto es  $c/a$ .
```

```
--
-- Nota. En la comparación usar ~= en lugar de ==
-----
```

```
-- La propiedad es
```

```
prop_raices :: Double -> Double -> Double -> Property
```

```
prop_raices a b c =
```

```
  a /= 0 && not (null xs) ==> sum xs ~= (-b/a) && product xs ~= (c/a)
  where xs = raices a b c
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_raices
```

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

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

```
-----
```

```
-- Ejercicio 11.2. Comprobar con QuickCheck que la intersección de
-- intervalos es conmutativa.
```

```
-----
```

```
-- La propiedad es
```

```

prop_interseccion :: Int -> Int -> Int -> Int -> Property
prop_interseccion a1 b1 a2 b2 =
  a1 <= b1 && a2 <= b2 ==>
  interseccion [a1,b1] [a2,b2] == interseccion [a2,b2] [a1,b1]

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

-----
-- 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 :: (Integer,Integer) -> (Integer,Integer)
-- 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 :: (Integer,Integer) -> (Integer,Integer)
formaReducida (0,_) = (0,1)
formaReducida (a,b) = (a `div` c, b `div` c)
  where c = gcd a b

-----
-- Ejercicio 12.2. Definir la función
--   sumaRacional :: (Integer,Integer) -> (Integer,Integer) -> (Integer,Integer)
-- 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 :: (Integer,Integer) -> (Integer,Integer) -> (Integer,Integer)
sumaRacional (a,b) (c,d) = formaReducida (a*d+b*c, b*d)
-----

```



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

Relación 3

Definiciones por comprensión

```
-----  
-- Introducción --  
-----  
  
-- En esta relación se presentan ejercicios con definiciones por  
-- comprensión correspondientes al tema 5 cuyas transparencias se  
-- encuentran en  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-5.pdf  
-----  
-- Librerías auxiliares --  
-----  
  
import Test.QuickCheck  
  
-----  
-- Ejercicios propuestos --  
-----  
  
-----  
-- 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. por ejemplo,  
--  subconjunto [3,2,3] [2,5,3,5] == True  
--  subconjunto [3,2,3] [2,5,6,5] == False  
-----
```

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

```
subconjunto xs ys =
  [x | x <- xs, x 'elem' ys] == xs
```

```
-----
-- Ejercicio 2. Definir la función
```

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

```
-- tal que (iguales xs ys) se verifica si xs e ys son iguales. Por
```

```
-- ejemplo,
```

```
-- iguales [3,2,3] [2,3] == True
```

```
-- iguales [3,2,3] [2,3,2] == True
```

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

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

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

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

```
-----
-- Ejercicio 3. 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]
-----
```

```
union :: Eq a => [a] -> [a] -> [a]
```

```
union xs ys = xs ++ [y | y <- ys, y 'notElem' xs]
```

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

```
-----
-- Ejercicio 4. Comprobar con QuickCheck que la unión es conmutativa.
-----
```

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

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

```
-----
-- Ejercicio 5. 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] == []
-----
```

```
interseccion :: Eq a => [a] -> [a] -> [a]
interseccion xs ys =
    [x | x <- xs, x `elem` ys]
```

```
-----
-- Ejercicio 6. Comprobar con QuickCheck si se cumple la siguiente
-- propiedad
--    $A \cup (B \cap C) = (A \cup B) \cap C$ 
-- 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 7. Definir la función
-- diferencia :: Eq a => [a] -> [a] -> [a]
-- tal que (diferencia xs ys) es la diferencia entre los conjuntos xs e
-- ys. Por ejemplo,
-- diferencia [3,2,5,6] [5,7,3,4] == [2,6]
-- diferencia [3,2,5] [5,7,3,2] == []

```

```

diferencia :: Eq a => [a] -> [a] -> [a]
diferencia xs ys = [x | x <- xs, x `notElem` ys]

```

```

-- Ejercicio 8. Comprobar con QuickCheck si la diferencia es
-- conmutativa.

```

```

prop_diferencia_conmutativa :: [Int] -> [Int] -> Bool
prop_diferencia_conmutativa xs ys =
  iguales (diferencia xs ys) (diferencia ys xs)

```

```

-- La comprobación es
-- ghci> quickCheck prop_diferencia_conmutativa
-- *** Failed! Falsifiable (after 2 tests and 2 shrinks):
-- [0]
-- []
-- que es un contraejemplo, ya que
-- [0] - [] = [0]
-- [] - [0] = []

```

```
-- Ejercicio 9. Comprobar con QuickCheck si se cumple la siguiente
-- propiedad:  $A \setminus B \subset A$ 
-----

-- La propiedad es
prop_diferencia_subconjunto :: [Int] -> [Int] -> Bool
prop_diferencia_subconjunto xs ys =
    subconjunto (diferencia xs ys) xs

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

-- Ejercicio 10. Comprobar con QuickCheck si se cumple la siguiente
-- propiedad:  $(A \setminus B) \cap B = \emptyset$ .
-----

-- La propiedad es
prop_diferencia_interseccion :: [Int] -> [Int] -> Bool
prop_diferencia_interseccion xs ys =
    interseccion (diferencia xs ys) ys == []

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

-- Ejercicio 11. Definir la función
-- divisoresPrimos :: Int -> [Int]
-- tal que (divisoresPrimos x) es la lista de los divisores primos de x.
-- Por ejemplo,
-- divisoresPrimos 40 == [2,5]
-- divisoresPrimos 70 == [2,5,7]
-----

divisoresPrimos :: Int -> [Int]
divisoresPrimos x = [n | n <- divisores x, primo n]

-- (divisores n) es la lista de los divisores del número n. Por ejemplo,
```

```

--    divisores 30 == [1,2,3,5,6,10,15,30]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `mod` x == 0]

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 30 == False
--    primo 31 == True
primo :: Int -> Bool
primo n = divisores n == [1, n]

-----
-- Ejercicio 12. Un número es libre de cuadrados si no es divisible por
-- el cuadrado de ningún entero mayor que 1. Por ejemplo, 70 es libre de
-- cuadrado porque sólo es divisible por 1, 2, 5, 7 y 70; en cambio, 40
-- no es libre de cuadrados porque es divisible por 2^2.
--
-- Definir la función
--    libreDeCuadrados :: Int -> Bool
-- tal que (libreDeCuadrados x) se verifica si x es libre de cuadrados.
-- Por ejemplo,
--    libreDeCuadrados 70 == True
--    libreDeCuadrados 40 == False
-----

-- 1ª definición:
libreDeCuadrados :: Int -> Bool
libreDeCuadrados x = x == product (divisoresPrimos x)

-- 2ª definición
libreDeCuadrados2 :: Int -> Bool
libreDeCuadrados2 n =
    null [x | x <- [2..n], rem n (x^2) == 0]

-----
-- Ejercicio 13. [Del problema 21 del Proyecto Euler]. Sea  $d(n)$  la suma
-- de los divisores propios de  $n$ . Si  $d(a) = b$  y  $d(b) = a$ , siendo  $a \neq b$ ,
-- decimos que  $a$  y  $b$  son un par de números amigos. Por ejemplo, los
-- divisores propios de 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y
-- 110; por tanto,  $d(220) = 284$ . Los divisores propios de 284 son 1, 2,
-- 4, 71 y 142; por tanto,  $d(284) = 220$ . Luego, 220 y 284 son dos

```



```
-- números amigos.
--
-- Definir la función
-- amigos :: Int -> Int -> Bool
-- tal que (amigos a b) se verifica si a y b son números amigos. Por
-- ejemplo,
-- amigos 6 6      == False
-- amigos 220 248  == False
-- amigos 220 284  == True
-- amigos 100 200  == False
-- amigos 1184 1210 == True
```

```
-----
amigos :: Int -> Int -> Bool
amigos a b = a /= b && sumaDivisores a == b && sumaDivisores b == a
  where sumaDivisores n = sum [x | x <- [1..n-1], n `rem` x == 0]
```

```
-----
-- Ejercicio 14. (Problema 211 del proyecto Euler) Dado un entero
-- positivo n, consideremos la suma de los cuadrados de sus divisores,
-- Por ejemplo,
-- f(10) = 1 + 4 + 25 + 100 = 130
-- f(42) = 1 + 4 + 9 + 36 + 49 + 196 + 441 + 1764 = 2500
-- Decimos que n es especial si f(n) es un cuadrado perfecto. En los
-- ejemplos anteriores, 42 es especial y 10 no lo es.
--
-- Definir la función
-- especial :: Int -> Bool
-- tal que (especial x) se verifica si x es un número es especial. Por
-- ejemplo,
-- especial 42 == True
-- especial 10 == False
-- Calcular todos los números especiales de tres cifras.
```

```
-----
especial :: Int -> Bool
especial n = esCuadrado (sum [x^2 | x <- divisores n])
```

```
-- (esCuadrado n) se verifica si n es un cuadrado perfecto. Por ejemplo,
-- esCuadrado 36 == True
```

```

--     esCuadrado 40 == False
esCuadrado :: Int -> Bool
esCuadrado n = y^2 == n
             where y = floor (sqrt (fromIntegral n))

-----

-- Ejercicio 15. La multiplicidad de x en y es la mayor potencia de x
-- que divide a y. Por ejemplo, la multiplicidad de 2 en 40 es 3 porque
-- 40 es divisible por 2^3 y no lo es por 2^4. Además, la multiplicidad
-- de 1 en cualquier número se supone igual a 1.
--
-- Definir la función
--     multiplicidad :: Integer -> Integer -> Integer
-- tal que (multiplicidad x y) es la
-- multiplicidad de x en y. Por ejemplo,
--     multiplicidad 2 40 == 3
--     multiplicidad 5 40 == 1
--     multiplicidad 3 40 == 0
--     multiplicidad 1 40 == 1
-----

multiplicidad :: Integer -> Integer -> Integer
multiplicidad 1 _ = 1
multiplicidad x y =
  head [n | n <- [0..], y `rem` (x^(n+1)) /= 0]

-----

-- Ejercicio 16.1. Sea t una lista de pares de la forma
-- (nombre, [(asig_1, nota_1), ..., (asig_k, nota.k)])
-- Por ejemplo,
--     t1 = [("Ana", [("Alg",1), ("Cal",3), ("Inf",8), ("Fis",2)]),
--          ("Juan", [("Alg",5), ("Cal",1), ("Inf",2), ("Fis",9)]),
--          ("Alba", [("Alg",5), ("Cal",6), ("Inf",6), ("Fis",5)]),
--          ("Pedro", [("Alg",9), ("Cal",5), ("Inf",3), ("Fis",1)])]
-- Definir la función
--     calificaciones :: [(String, [(String, Int)])] -> String -> [(String, Int)]
-- tal que (calificaciones t p) es la lista de las calificaciones de la
-- persona p en la lista t. Por ejemplo,
--     ghci> calificaciones t1 "Pedro"
--     [("Alg",9), ("Cal",5), ("Inf",3), ("Fis",1)]

```

```

-----
t1 :: [(String,[(String,Int)])]
t1 = [("Ana",[("Alg",1),("Cal",3),("Inf",8),("Fis",2)]),
      ("Juan",[("Alg",5),("Cal",1),("Inf",2),("Fis",9)]),
      ("Alba",[("Alg",5),("Cal",6),("Inf",6),("Fis",5)]),
      ("Pedro",[("Alg",9),("Cal",5),("Inf",3),("Fis",1)])]

calificaciones :: [(String,[(String,Int)])] -> String -> [(String,Int)]
calificaciones t p = head [xs | (x,xs) <- t, x == p]

```

```

-----
-- Ejercicio 16.2. Definir la función
--   todasAprobadas :: [(String,[(String,Int)])] -> String -> Bool
-- tal que (todasAprobadas t p) se cumple si en la lista t, p tiene
-- todas las asignaturas aprobadas. Por ejemplo,
--   todasAprobadas t1 "Alba" == True
--   todasAprobadas t1 "Pedro" == False

```

```

-----
todasAprobadas :: [(String,[(String,Int)])] -> String -> Bool
todasAprobadas t p = and [n >= 5 | (_,n) <- calificaciones t p]

```

```

-----
-- Ejercicio 16.3. Definir la función
--   aprobados :: [(String,[(String,Int)])] -> [String]
-- tal que (aprobados t) es la lista de los alumnos de la lista de notas
-- t que han aprobado todas las asignaturas. Por ejemplo,
--   aprobados t1 == ["Alba"]

```

```

-----
aprobados :: [(String,[(String,Int)])] -> [String]
aprobados t = [p | (p,_) <- t, todasAprobadas t p]

```

```

-----
-- Ejercicio 17. Un número n es de Angelini si n y 2n tienen algún
-- dígito común. Por ejemplo, 2014 es un número de Angelini ya que 2014
-- y su doble (4028) comparten los dígitos 4 y 0.
--
-- Definir la función

```

```
-- angelini :: Integer -> Bool
-- tal que (angelini n) se verifica si n es un número de Angelini. Por
-- ejemplo,
-- angelini 2014 == True
-- angelini 2067 == False
```

```
-----
angelini :: Integer -> Bool
angelini n = not (null [x | x <- show n, x 'elem' show (2*n)])
```

```
-----
-- Ejercicio 18.1. Definir la función
-- unitarios :: Integer -> [Integer]
-- tal (unitarios n) es la lista de números [n,nn,nnn, ...]. Por
-- ejemplo.
-- take 7 (unitarios 3) == [3,33,333,3333,33333,333333,3333333]
-- take 3 (unitarios 1) == [1,11,111]
```

```
-----
unitarios :: Integer -> [Integer]
unitarios x = [x*(div (10^n-1) 9) | n <- [1 ..]]
```

```
-----
-- Ejercicio 18.2. Definir la función
-- multiplosUnitarios :: Integer -> Integer -> Int -> [Integer]
-- tal que (multiplosUnitarios x y n) es la lista de los n primeros
-- múltiplos de x cuyo único dígito es y. Por ejemplo,
-- multiplosUnitarios 7 1 2 == [111111,111111111111]
-- multiplosUnitarios 11 3 5 == [33,3333,333333,33333333,3333333333]
```

```
-----
multiplosUnitarios :: Integer -> Integer -> Int -> [Integer]
multiplosUnitarios x y n = take n [z | z <- unitarios y, mod z x == 0]
```

```
-----
-- Ejercicio 19. Definir la función
-- masOcurrentes :: Eq a => [a] -> [a]
-- tal que (masOcurrentes xs) es la lista de los elementos de xs que
-- ocurren el máximo número de veces. Por ejemplo,
-- masOcurrentes [1,2,3,4,3,2,3,1,4] == [3,3,3]
```

```

--      masOcurrentes [1,2,3,4,5,2,3,1,4] == [1,2,3,4,2,3,1,4]
--      masOcurrentes "Salamanca"         == "aaaa"
-----

masOcurrentes :: Eq a => [a] -> [a]
masOcurrentes xs = [x | x <- xs, ocurrencias x xs == m]
  where m = maximum [ocurrencias x xs | x <-xs]

-- (ocurrencias x xs) es el número de ocurrencias de x en xs. Por
-- ejemplo,
--      ocurrencias 1 [1,2,3,4,3,2,3,1,4] == 2
ocurrencias x xs = length [x' | x' <- xs, x == x']

-----

-- Ejercicio 20. La suma de la serie
--       $1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$ 
-- es  $\pi^2/6$ . Por tanto,  $\pi$  se puede aproximar mediante la raíz cuadrada
-- de 6 por la suma de la serie.
--
-- Definir la función aproximaPi tal que (aproximaPi n) es la aproximación
-- de pi obtenida mediante n términos de la serie. Por ejemplo,
--      aproximaPi 4      == 2.9226129861250305
--      aproximaPi 1000  == 3.1406380562059946
-----

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

-----

-- Ejercicio 21. Una representación de 20 en base 2 es [0,0,1,0,1] pues
--  $20 = 1 \cdot 2^2 + 1 \cdot 2^4$ . Y una representación de 46 en base 3 es [1,0,2,1]
-- pues  $46 = 1 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 + 1 \cdot 3^3$ .
--
-- Definir la función
--      deBaseABase10 :: Int -> [Int] -> Int
-- tal que (deBaseABase10 b xs) es el número n tal que su representación
-- en base b es xs. Por ejemplo,
--      deBaseABase10 2 [0,0,1,0,1]      == 20
--      deBaseABase10 2 [1,1,0,1]       == 11
--      deBaseABase10 3 [1,0,2,1]       == 46
--      deBaseABase10 5 [0,2,1,3,1,4,1] == 29160

```

```

-----
deBaseABase10 :: Int -> [Int] -> Int
deBaseABase10 b xs = sum [y*b^n | (y,n) <- zip xs [0..]]

```

```

-----
-- Ejercicio 22.1. Definir la función
--   conPos :: [a] -> [(a,Int)]
-- tal que (conPos xs) es la lista obtenida a partir de xs especificando
-- las posiciones de sus elementos. Por ejemplo,
--   conPos [1,5,0,7] == [(1,0),(5,1),(0,2),(7,3)]
-----

```

```

conPos :: [a] -> [(a,Int)]
conPos xs = zip xs [0..]

```

```

-----
-- Ejercicio 22.2. Definir la
--   pares :: [a] -> [a]
-- tal que (pares cs) es la cadena formada por los caracteres en
-- posición par de cs. Por ejemplo,
--   pares "el cielo sobre berlin" == "e il or eln"
-----

```

```

pares :: [a] -> [a]
pares cs = [c | (c,n) <- conPos cs, even n]

```

```

-----
-- Ejercicio 23. Dos listas xs, ys de la misma longitud son
-- perpendiculares si el producto escalar de ambas es 0, donde el
-- producto escalar de dos listas de enteros xs e ys viene
-- dado por la suma de los productos de los elementos correspondientes.
--
-- Definir la función
--   perpendiculares :: (Num a, Eq a) => [a] -> [[a]] -> [[a]]
-- tal que (perpendiculares xs yss) es la lista de los elementos de yss
-- que son perpendiculares a xs. Por ejemplo,
--   ghci> perpendiculares [1,0,1] [[0,1,0], [2,3,1], [-1,7,1],[3,1,0]]
--   [[0,1,0],[-1,7,1]]
-----

```

```
perpendiculares :: (Num a, Eq a) => [a] -> [[a]] -> [[a]]
perpendiculares xs yss = [ys | ys <- yss, productoEscalar xs ys == 0]
```

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

```
-----
-- Ejercicio 24. Dada una lista de números enteros, definiremos el
-- mayor salto como el mayor valor de las diferencias (en valor
-- absoluto) entre números consecutivos de la lista. Por ejemplo, dada
-- la lista [2,5,-3] las distancias son
--   3 (valor absoluto de la resta 2 - 5) y
--   8 (valor absoluto de la resta de 5 y (-3))
-- Por tanto, su mayor salto es 8. No está definido el mayor salto para
-- listas con menos de 2 elementos
--
-- Definir la función
--   mayorSalto :: [Integer] -> Integer
-- tal que (mayorSalto xs) es el mayor salto de la lista xs. Por
-- ejemplo,
--   mayorSalto [1,5]           == 4
--   mayorSalto [10,-10,1,4,20,-2] == 22
-----
```

```
mayorSalto :: [Integer] -> Integer
mayorSalto xs = maximum [abs (x-y) | (x,y) <- zip xs (tail xs)]
```

```
-----
-- Ejercicio 25. Definir la función
--   longCamino :: [(Float,Float)] -> Float
-- tal que (longCamino xs) es la longitud del camino determinado por los
-- puntos del plano listados en xs. Por ejemplo,
--   longCamino [(0,0),(1,0),(2,1),(2,0)] == 3.4142137
-----
```

```
longCamino :: [(Float,Float)] -> Float
longCamino xs =
  sum [sqrt ((a-c)^2+(b-d)^2) | ((a,b),(c,d)) <- zip xs (tail xs)]
```

```
longCamino2 xs =
  sum [sqrt ((a-c)^2+(b-d)^2) | ((a,b),(c,d)) <- zip xs (tail xs)]
```

```
longCamino3 :: [(Double,Double)] -> Double
```

```
longCamino3 xs =
  sum [sqrt ((a-c)^2+(b-d)^2) | ((a,b),(c,d)) <- zip xs (tail xs)]
```

```
-----
-- Ejercicio 26. Definir la función
```

```
--   numerosConsecutivos :: (Num a, Eq a) => [a] -> Int
```

```
--   tal que (numerosConsecutivosC xs) es la cantidad de números
```

```
--   consecutivos (es decir, números que son iguales a su predecesor en la
```

```
--   lista más uno) que aparecen en la lista xs. Por ejemplo,
```

```
--   numerosConsecutivos [] == 0
```

```
--   numerosConsecutivos [5,6,9] == 1
```

```
--   numerosConsecutivos [5,6,9,2,3,4] == 3
-----
```

```
numerosConsecutivos :: (Num a, Eq a) => [a] -> Int
```

```
numerosConsecutivos xs =
```

```
  length [x | (x,y) <- zip xs (tail xs), y == x+1]
```

```
-----
-- Ejercicio 27. Definir la función
```

```
--   sumaEquidistantes :: Num a => [a] -> [a]
```

```
--   tal que (sumaEquidistantes xs) es la lista sumando el primer elemento
```

```
--   de xs con el último, el segundo con el penúltimo y así
```

```
--   sucesivamente. Por ejemplo,
```

```
--   sumaEquidistantes [6,5,3,1] == [7,8]
```

```
--   sumaEquidistantes [6,5,3] == [9,10]
```

```
--   sumaEquidistantes [3,2,3,2] == [5,5]
```

```
--   sumaEquidistantes [6,5,3,1,2,0,4,7,8,9] == [15,13,10,5,2]
-----
```

```
sumaEquidistantes :: Num a => [a] -> [a]
```

```
sumaEquidistantes xs = take n [x+y | (x,y) <- zip xs (reverse xs)]
```

```
  where m = length xs
```

```
        n | even m     = m `div` 2
```

```
          | otherwise = 1 + (m `div` 2)
```



```

-----
-- Ejercicio 28. La distancia entre dos números es el valor absoluto de
-- su diferencia. Por ejemplo, la distancia entre 2 y 5 es 3.
--
-- Definir la función
--   cercanos :: [Int] -> [Int] -> [(Int,Int)]
-- tal que (cercanos xs ys) es la lista de pares de elementos de xs e ys
-- cuya distancia es mínima. Por ejemplo,
--   cercanos [3,7,2,1] [5,11,9] == [(3,5),(7,5),(7,9)]
-----

```

```

cercanos :: [Int] -> [Int] -> [(Int,Int)]
cercanos xs ys =
  [(x,y) | (x,y) <- pares, abs (x-y) == m]
  where pares = [(x,y) | x <- xs, y <- ys]
        m     = minimum [abs (x-y) | (x,y) <- pares]

```

```

-----
-- Ejercicio 29. [De la IMO 1996]. Una sucesión [a(0),a(1),...,a(n)]
-- se denomina cuadrática si para cada  $i \in \{1, 2, \dots, n\}$  se cumple que
--  $|a(i) - a(i-1)| = i^2$  .
--
-- Definir la función
--   esCuadratica :: [Int] -> Bool
-- tal que (esCuadratica xs) se verifica si xs cuadrática. Por ejemplo,
--   esCuadratica [2,1,-3,6] == True
--   esCuadratica [2,1,3,5] == False
--   esCuadratica [3,4,8,17,33,58,94,45,-19,-100] == True
-----

```

```

esCuadratica :: [Int] -> Bool
esCuadratica xs =
  and [abs (y-x) == i^2 | ((x,y),i) <- zip (adyacentes xs) [1..]]

adyacentes xs = zip xs (tail xs)

```

```

-----
-- Ejercicio 30.1. Definir las funciones
--   ultima, primera :: Int -> Int

```

```
-- tales que
-- * (ultima n) es la última cifra del número natural n y
-- * (primera n) es la primera cifra del número natural n.
-- Por ejemplo,
--   ultima 711 = 1
--   primera 711 = 7
```

```
-----
ultima, primera :: Int -> Int
ultima n = n `rem` 10
primera n = read [head (show n)]
```

```
-----
-- Ejercicio 30.2. Definir la función
--   encadenadoC :: [Int] -> Bool
-- tal que (encadenadoC xs) se verifica si xs es una lista de enteros
-- positivos encadenados (es decir, la última cifra de cada número
-- coincide con la primera del siguiente en la lista). Por ejemplo,
--   encadenadoC [711,1024,413,367] == True
--   encadenadoC [711,1024,213,367] == False
```

```
-----
encadenadoC :: [Int] -> Bool
encadenadoC xs =
  and [ultima x == primera y | (x,y) <- zip xs (tail xs)]
```

Relación 4

Definiciones por comprensión con cadenas: El cifrado César

```
-- -----  
-- Introducción --  
-- -----  
  
-- En el tema 5, cuyas transparencias se encuentran en  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-5.pdf  
-- se estudió, como aplicación de las definiciones por comprensión, el  
-- cifrado César. El objetivo de esta relación es modificar el programa  
-- de cifrado César para que pueda utilizar también letras  
-- mayúsculas. Por ejemplo,  
-- ghci> descifra "Ytit Ufwf Sfif"  
-- "Todo Para Nada"  
-- Para ello, se propone la modificación de las funciones correspondientes  
-- del tema 5.  
  
-- -----  
-- Importación de librerías auxiliares --  
-- -----  
  
import Data.Char  
import Test.QuickCheck  
  
-- -----  
-- Ejercicio 1. Definir la función  
-- minuscula2int :: Char -> Int
```

```
-- tal que (minuscuala2int c) es el entero correspondiente a la letra
-- minúscula c. Por ejemplo,
--   minuscuala2int 'a' == 0
--   minuscuala2int 'd' == 3
--   minuscuala2int 'z' == 25
-----
```

```
minuscuala2int :: Char -> Int
minuscuala2int c = ord c - ord 'a'
```

```
-- -----
-- Ejercicio 2. Definir la función
--   mayuscuala2int :: Char -> Int
-- tal que (mayuscuala2int c) es el entero correspondiente a la letra
-- mayúscula c. Por ejemplo,
--   mayuscuala2int 'A' == 0
--   mayuscuala2int 'D' == 3
--   mayuscuala2int 'Z' == 25
-----
```

```
mayuscuala2int :: Char -> Int
mayuscuala2int c = ord c - ord 'A'
```

```
-- -----
-- Ejercicio 3. Definir la función
--   int2minuscuala :: Int -> Char
-- tal que (int2minuscuala n) es la letra minúscula correspondiente al
-- entero n. Por ejemplo,
--   int2minuscuala 0 == 'a'
--   int2minuscuala 3 == 'd'
--   int2minuscuala 25 == 'z'
-----
```

```
int2minuscuala :: Int -> Char
int2minuscuala n = chr (ord 'a' + n)
```

```
-- -----
-- Ejercicio 4. Definir la función
--   int2mayuscuala :: Int -> Char
-- tal que (int2mayuscuala n) es la letra minúscula correspondiente al
```

```
-- entero n. Por ejemplo,
--   int2mayuscula 0  == 'A'
--   int2mayuscula 3  == 'D'
--   int2mayuscula 25 == 'Z'
```

```
-----
int2mayuscula :: Int -> Char
int2mayuscula n = chr (ord 'A' + n)
```

```
-----
-- Ejercicio 5. Definir la función
--   desplaza :: Int -> Char -> Char
-- tal que (desplaza n c) es el carácter obtenido desplazando n
-- caracteres el carácter c. Por ejemplo,
--   desplaza 3 'a' == 'd'
--   desplaza 3 'y' == 'b'
--   desplaza (-3) 'd' == 'a'
--   desplaza (-3) 'b' == 'y'
--   desplaza 3 'A' == 'D'
--   desplaza 3 'Y' == 'B'
--   desplaza (-3) 'D' == 'A'
--   desplaza (-3) 'B' == 'Y'
```

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

```
-----
-- Ejercicio 6.1. Definir la función
--   codifica :: Int -> String -> String
-- tal que (codifica n xs) es el resultado de codificar el texto xs con
-- un desplazamiento n. Por ejemplo,
--   ghci> codifica 3 "En Todo La Medida"
--   "Hq Wrgr Od Phlgd"
--   ghci> codifica (-3) "Hq Wrgr Od Phlgd"
--   "En Todo La Medida"
```

```

codifica :: Int -> String -> String
codifica n xs = [desplaza n x | x <- xs]

-----
-- Ejercicio 6.2. Comprobar con QuickCheck que para cualquier entero n y
-- cualquier cadena cs se tiene que (codifica (-n) (codifica n cs)) es
-- igual a cs.
-----

-- La propiedad es
prop_codifica :: Int -> String -> Bool
prop_codifica n cs =
  codifica (-n) (codifica n cs) == cs

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

-----
-- Ejercicio 7. Definir la función
-- tabla :: [Float]
-- tal que tabla es la lista de la frecuencias de las letras en
-- castellano, Por ejemplo, la frecuencia de la 'a' es del 12.53%, la de
-- la 'b' es 1.42%.
-----

tabla :: [Float]
tabla = [12.53, 1.42, 4.68, 5.86, 13.68, 0.69, 1.01,
        0.70, 6.25, 0.44, 0.01, 4.97, 3.15, 6.71,
        8.68, 2.51, 0.88, 6.87, 7.98, 4.63, 3.93,
        0.90, 0.02, 0.22, 0.90, 0.52]

-----
-- Ejercicio 8. Definir la función
-- porcentaje :: Int -> Int -> Float
-- tal que (porcentaje n m) es el porcentaje de n sobre m. Por ejemplo,
-- porcentaje 2 5 == 40.0
-----

```

```
porcentaje :: Int -> Int -> Float
porcentaje n m = (fromIntegral n / fromIntegral m) * 100
```

```
-----
-- Ejercicio 9. Definir la función
--   letras :: String -> String
-- tal que (letras xs) es la cadena formada por las letras de la cadena
-- xs. Por ejemplo,
--   letras "Esto Es Una Prueba" == "EstoEsUnaPrueba"
-----
```

```
letras :: String -> String
letras xs = [x | x <- xs, elem x (['a'..'z']++['A'..'Z'])]
```

```
-----
-- Ejercicio 10.1. Definir la función
--   ocurrencias :: Eq a => a -> [a] -> Int
-- tal que (ocurrencias x xs) es el número de veces que ocurre el
-- elemento x en la lista xs. Por ejemplo,
--   ocurrencias 'a' "Salamanca" == 4
-----
```

```
ocurrencias :: Eq a => a -> [a] -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']
```

```
-----
-- Ejercicio 10.2. Comprobar con QuickCheck si el número de ocurrencias
-- de un elemento x en una lista xs es igual que en su inversa.
-----
```

```
-- La propiedad es
prop_ocurrencia_inv :: Int -> [Int] -> Bool
prop_ocurrencia_inv x xs =
  ocurrencias x xs == ocurrencias x (reverse xs)
```

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

```
-- Ejercicio 10.3. Comprobar con QuickCheck si el número de ocurrencias
-- de un elemento x en la concatenación de las listas xs e ys es igual a
-- la suma del número de ocurrencias de x en xs y en ys.
```

```
-----
-- La propiedad es
prop_ocurrencia_conc :: Int -> [Int] -> [Int] -> Bool
prop_ocurrencia_conc x xs ys =
  ocurrencias x (xs++ys) == ocurrencias x xs + ocurrencias x ys
```

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

```
-----
-- Ejercicio 12. Definir la función
-- frecuencias :: String -> [Float]
-- tal que (frecuencias xs) es la frecuencia de cada una de las letras
-- de la cadena xs. Por ejemplo,
-- ghci> frecuencias "En Todo La Medida"
-- [14.3,0,0,21.4,14.3,0,0,0,7.1,0,0,7.1,
-- 7.1,7.1,14.3,0,0,0,0,7.1,0,0,0,0,0,0]
```

```
-----
frecuencias :: String -> [Float]
frecuencias xs =
  [porcentaje (ocurrencias x xs') n | x <- ['a'..'z']]
  where xs' = [toLower x | x <- xs]
        n   = length (letras xs)
```

```
-----
-- Ejercicio 13.1. Definir la función
-- chiCudad :: [Float] -> [Float] -> Float
-- tal que (chiCudad os es) es la medida chi cuadrado de las
-- distribuciones os y es. Por ejemplo,
-- chiCudad [3,5,6] [3,5,6] == 0.0
-- chiCudad [3,5,6] [5,6,3] == 3.9666667
```

```
-----
chiCudad :: [Float] -> [Float] -> Float
```



```
chiCudad os es = sum [((o-e)^2)/e | (o,e) <- zip os es]
```

```
-----  
-- Ejercicio 13.2, Comprobar con QuickCheck que para cualquier par de  
-- listas xs e ys se verifica que (chiCudad xs ys) es 0 syss xs e ys son  
-- iguales.  
-----
```

```
-- La propiedad es
```

```
prop_chiCudad_1 :: [Float] -> [Float] -> Bool
```

```
prop_chiCudad_1 xs ys =
```

```
    (chiCudad xs ys == 0) == (xs == ys)
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_chiCudad_1
```

```
-- *** Failed! Falsifiable (after 2 tests and 2 shrinks):
```

```
-- [2.0]
```

```
-- []
```

```
-- En efecto,
```

```
-- ghci> chiCudad [2] [] == 0
```

```
-- True
```

```
-- ghci> [2] == []
```

```
-- False
```

```
-----  
-- Ejercicio 13.3. A la vista de los contraejemplos del apartado  
-- anterior, qué condición hay que añadir para que se verifique la  
-- propiedad.  
-----
```

```
-- La propiedad es
```

```
prop_chiCudad_2 :: [Float] -> [Float] -> Property
```

```
prop_chiCudad_2 xs ys =
```

```
    length xs == length ys ==> (chiCudad xs ys == 0) == (xs == ys)
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_chiCudad_2
```

```
-- *** Gave up! Passed only 47 tests.
```

```
-- Ejercicio 13.3. A la vista del apartado anterior, el número de tests
-- que ha pasado puede ser menor que 100. Reescribir la propiedad de
-- forma que se verifique en los 100 tests.
```

```
-----
-- La propiedad es
prop_chiCudad_3 :: [Float] -> [Float] -> Bool
prop_chiCudad_3 xs ys =
  (chiCudad as bs == 0) == (as == bs)
  where n = min (length xs) (length ys)
        as = take n xs
        bs = take n ys
```

```
-- La comprobación es
-- ghci> quickCheck prop_chiCudad_3
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 14.1. Definir la función
--   rota :: Int -> [a] -> [a]
-- tal que (rota n xs) es la lista obtenida rotando n posiciones los
-- elementos de la lista xs. Por ejemplo,
--   rota 2 "manolo"           == "noloma"
--   rota 10 "manolo"          == "lomano"
--   [rota n "abc" | n <- [0..5]] == ["abc","bca","cab","abc","bca","cab"]
```

```
-----
rota :: Int -> [a] -> [a]
rota _ [] = []
rota n xs = drop m xs ++ take m xs
  where m = n `mod` length xs
```

```
-----
-- Ejercicio 14.2. Comprobar con QuickCkeck si para cualquier lista xs
-- si se rota n veces y el resultado se rota m veces se obtiene lo mismo
-- que rotando xs (n+m) veces, donde n y m son números no nulos.
```

```
-----
-- La propiedad es
prop_rota :: Int -> Int -> [Int] -> Property
```

```

prop_rota n m xs =
  n /= 0 && m /= 0 ==> rota m (rota n xs) == rota (n+m) xs

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

-----
-- Ejercicio 15.1. Definir la función
--   descifra :: String -> String
-- tal que (descifra xs) es la cadena obtenida descodificando la cadena
-- xs por el anti-desplazamiento que produce una distribución de letras
-- con la menor desviación chi cuadrado respecto de la tabla de
-- distribución de las letras en castellano. Por ejemplo,
--   ghci> codifica 5 "Todo Para Nada"
--   "Ytit Ufwf Sfif"
--   ghci> descifra "Ytit Ufwf Sfif"
--   "Todo Para Nada"
-----

descifra :: String -> String
descifra xs = codifica (-factor) xs
  where factor = head (posiciones (minimum tabChi) tabChi)
        tabChi = [chiCuad (rota n tabla') tabla | n <- [0..25]]
        tabla' = frecuencias xs

posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
  [i | (x',i) <- zip xs [0..], x == x']

```


Relación 5

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/ilm-14/temas/tema-6.pdf  
-----  
-- Importación de librerías auxiliares --  
-----  
  
import Test.QuickCheck  
import Data.List  
import Data.Char  
  
-----  
-- 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 m 0 = 1  
potencia m n = m*(potencia m (n-1))
```

```

-----
-- Ejercicio 1.2. Comprobar con QuickCheck que la función potencia es
-- 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
-- Falsificable, 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,
-- 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 :: Eq a => [[a]] -> Bool
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 n []         = []
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)

-----
-- 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' 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

-- 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..]]

-- 2ª definición:

```

```

listaNumeroC2 :: [Integer] -> Integer
listaNumeroC2 xs = read [x | x <- show xs, isDigit x]

-----

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

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

capicua :: Integer -> Bool
capicua n = show n == reverse (show n)

-----

-- Ejercicio 11.1. Definir, por recursión, la función
-- mayorExponenteR :: Integer -> Integer -> Integer
-- tal que (mayorExponenteR a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
-- mayorExponenteR 2 8    == 3
-- mayorExponenteR 2 9    == 0
-- mayorExponenteR 5 100  == 2
-- mayorExponenteR 2 60   == 2
-----

```

```
mayorExponenteR :: Integer -> Integer -> Integer
```

```
mayorExponenteR a b
```

```
  | rem b a /= 0 = 0
```

```
  | otherwise   = 1 + mayorExponenteR a (b `div` a)
```

```
-----
-- Ejercicio 11.2. 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
```

```
-----
mayorExponenteC :: Integer -> Integer -> Integer
```

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

```
-----
-- Ejercicio 12.1. La suma de la serie
```

```
--    $1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$ 
```

```
-- es  $\pi^2/6$ . Por tanto,  $\pi$  se puede aproximar mediante la raíz cuadrada
```

```
-- de 6 por la suma de la serie.
```

```
-- Definir, por comprensión, la función aproximaPiC tal que
```

```
-- (aproximaPiC n) es la aproximación de  $\pi$  obtenida mediante n
```

```
-- términos de la serie. Por ejemplo,
```

```
--   aproximaPiC 4    == sqrt(6*(1/1^2 + 1/2^2 + 1/3^2 + 1/4^2))
```

```
--                               == 2.9226129861250305
```

```
--   aproximaPiC 1000 == 3.1406380562059946
```

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

```
-----
-- Ejercicio 12.2. Definir, por recursión, la función aproximaPiR tal
```

```
-- que (aproximaPiR n) es la aproximación de  $\pi$  obtenida mediante n
```

```
-- términos de la serie. Por ejemplo,
```

```
--   aproximaPiR 4    == sqrt(6*(1/1^2 + 1/2^2 + 1/3^2 + 1/4^2))
```

```

--                                     == 2.9226129861250305
--   aproximaPiR 1000 == 3.1406380562059946
-----

aproximaPiR n = sqrt(6*aproximaPiR' n)

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

-----

-- Ejercicio 13.1. Comprobar con QuickCheck si la función mcd definida
-- en el ejercicio 2.1 es equivalente a la función gcd
-----

-- La propiedad es
prop_mcd_gcd :: Integer -> Integer -> Bool
prop_mcd_gcd a b =
  mcd a b == gcd a b

-- La comprobación es
--   ghci> quickCheck prop_mcd_gcd
--   *** Failed! Falsifiable (after 5 tests and 2 shrinks):
--   0
--   -1
-- Efectivamente,
--   ghci> mcd 0 (-1)
--   -1
--   ghci> gcd 0 (-1)
--   1

-----

-- Ejercicio 13.2. Definir la función
--   mcdE :: Integer -> Integer -> Integer
-- tal que (mcdE a b) es el máximo común divisor de a y b calculado
-- mediante el algoritmo de Euclides, pero extendido a los números
-- negativos. Por ejemplo,
--   mcdE 30 45 == 15
--   mcdE (-2) 0 == 2
--   mcdE (-4) 6 == 2
--   mcdE 0 4 == 4

```



```

--      mcdE 0 0      ==  0
-----

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

-----

-- Ejercicio 13.3. Comprobar con QuickCheck si las funciones mcdE y gcd
-- son equivalentes.
-----

-- La propiedad es
prop_mcdE_gcd :: Integer -> Integer -> Bool
prop_mcdE_gcd a b =
    mcdE a b == gcd a b

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

-----

-- Ejercicio 13.4. Comprobar con QuickCheck que (mcd a b) es un divisor
-- de a y de b.
-----

-- La propiedad es
prop_mcdE_esDivisor :: Integer -> Integer -> Property
prop_mcdE_esDivisor a b =
    a > 0 && b > 0 ==> a `rem` m == 0 && b `rem` m == 0
    where m = mcdE a b

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

-----

-- Ejercicio 13.4. Comprobar con QuickCheck que todos los divisores
-- comunes de a y b son divisores de (mcdE a b).
-----

```

```

-- La propiedad es
prop_mcdE_esMaximo :: Integer -> Integer -> Integer -> Property
prop_mcdE_esMaximo a b c =
  a > 0 && b > 0 && c /= 0 && divide c a && divide c b
  ==> divide c (mcdE a b)
  where divide x y = rem y x == 0

-- La comprobación es
-- ghci> quickCheck prop_mcdE_esMaximo
-- *** Gave up! Passed only 26 tests.

-- La propiedad es
prop_mcdE_esMaximo2 :: Integer -> Integer -> Integer -> Property
prop_mcdE_esMaximo2 a b c =
  a > 0 && b > 0
  ==> and [divide x (mcdE a b) | x <- divisores a, divide x b]

-- (divide x y) se verifica si x divide a y. Por ejemplo,
-- divide 2 6 == True
-- divide 2 7 == False
divide :: Integer -> Integer -> Bool
divide x y = rem y x == 0

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
-- divisores 90 == [1,2,3,5,6,9,10,15,18,30,45,90]
divisores :: Integer -> [Integer]
divisores x = [y | y <- [1..x], divide y x]

-- La comprobación es
-- ghci> quickCheck prop_mcdE_esMaximo2
-- +++ OK, passed 100 tests.

-----
-- Ejercicio 14.1. Definir, por comprensión, la función
-- mcdC :: Integer -> Integer -> Integer
-- tal que (mcdC a b) es el máximo común divisor de a y b. Por ejemplo,
-- mcdC 30 45 == 15
-- mcdC (-2) 0 == 2
-- mcdC (-4) 6 == 2

```

```
--      mcdC 0 4    == 4
--      mcdC 0 0    == 0
-----

mcdC :: Integer -> Integer -> Integer
mcdC 0 b = abs b
mcdC a b = head [x | x <- [c,c-1..1], divide x a, divide x b]
           where c = min (abs a, abs b)

-----

-- Ejercicio 14.2. Comprobar con QuickCheck si las funciones mcdC y gcd
-- son equivalentes.
-----

-- La propiedad es
prop_mcdC_gcd :: Integer -> Integer -> Bool
prop_mcdC_gcd a b =
    mcdC a b == gcd a b

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


Relación 6

Ordenación por mezcla

```
-----  
-- Introducción  
-----  
  
-- El objetivo de esta relación es definir la ordenación por mezclas y  
-- comprobar su corrección con QuickCheck.  
  
import Test.QuickCheck  
  
-----  
-- Ejercicio 1. Definir por recursión la función  
-- mezcla :: Ord a => [a] -> [a] -> [a]  
-- tal que (mezcla xs ys) es la lista obtenida mezclando las listas  
-- ordenadas xs e ys. Por ejemplo,  
-- mezcla [2,5,6] [1,3,4] == [1,2,3,4,5,6]  
-----  
  
mezcla :: Ord a => [a] -> [a] -> [a]  
mezcla [] ys = ys  
mezcla xs [] = xs  
mezcla (x:xs) (y:ys) | x <= y = x : mezcla xs (y:ys)  
                    | otherwise = y : mezcla (x:xs) ys  
  
-----  
-- Ejercicio 2. Definir la función  
-- mitades :: [a] -> ([a],[a])  
-- tal que (mitades xs) es el par formado por las dos mitades en que se
```

```
-- divide xs tales que sus longitudes difieren como máximo en uno. Por
-- ejemplo,
--     mitades [2,3,5,7,9] == ([2,3],[5,7,9])
-----
```

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

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

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

```
-- Ejercicio 4. Definir por recursión la función
--     ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada. Por
-- ejemplo,
--     ordenada [2,3,5] == True
--     ordenada [2,5,3] == False
-----
```

```
ordenada :: Ord a => [a] -> Bool
ordenada [] = True
ordenada [_] = True
ordenada (x:y:xs) = x <= y && ordenada (y:xs)
-----
```

```
-- Ejercicio 5. Comprobar con QuickCheck que la ordenación por mezcla
-- de una lista es una lista ordenada.
```

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

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

```
-----
-- Ejercicio 6. Definir por recursión la función
-- borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
-- x en la lista xs. Por ejemplo,
-- borra 1 [1,2,1] == [2,1]
-- borra 3 [1,2,1] == [1,2,1]
```

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

```
-----
-- Ejercicio 7. Definir por recursión la función
-- esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutación de
-- ys. Por ejemplo,
-- esPermutacion [1,2,1] [2,1,1] == True
-- esPermutacion [1,2,1] [1,2,2] == False
```

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

```
-----  
-- Ejercicio 8. Comprobar con QuickCheck que la ordenación por mezcla  
-- de una lista es una permutación de la lista.  
-----
```

```
-- La propiedad es
```

```
prop_ordMezcla_pemutacion :: [Int] -> Bool
```

```
prop_ordMezcla_pemutacion xs = esPermutacion (ordMezcla xs) xs
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_ordMezcla_pemutacion
```

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


Relación 7

Funciones sobre cadenas

```
-----  
-- 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
-- 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"
-- 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 x | x <- 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 :: String -> String
mayusculaInicialRec [] = []
mayusculaInicialRec (x:xs) = toUpper x : aux xs
  where aux (x:xs) = toLower x : aux xs
        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.
-----
```

```
-- Ejercicio 3.1. Se consideran las siguientes reglas de mayúsculas
-- iniciales para los títulos:
```

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

```

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

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

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

-----
-- Ejercicio 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 (p:ps) = transforma p : tituloRecAux ps

-----
-- 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
--   buscaCrucigrama :: Char -> Int -> Int -> [String] -> [String]
-- tal que (buscaCrucigrama l pos lon ps) es la lista de las palabras de
-- la lista de palabras ps que tienen longitud lon y poseen la letra l en
-- la posición pos (comenzando en 0). Por ejemplo,
--   ghci> buscaCrucigrama 'c' 1 7 ["ocaso", "casa", "ocupado"]
--   ["ocupado"]
--   ghci> buscaCrucigrama 'c' 21 7 ["ocaso", "casa", "ocupado"]
--   []
--   ghci> buscaCrucigrama 'c' (-1) 7 ["ocaso", "casa", "ocupado"]
--   []
-- -----
```

```
buscaCrucigrama :: Char -> Int -> Int -> [String] -> [String]
```

```
buscaCrucigrama l pos lon ps =
  [p | p <- ps,
    length p == lon,
    0 <= pos, pos < length p,
    p !! pos == l]
```

```
-- -----
-- Ejercicio 4.2. Definir, por recursión, la función
--   buscaCrucigramaR :: Char -> Int -> Int -> [String] -> [String]
-- tal que (buscaCrucigramaR l pos lon ps) es la lista de las palabras
-- de la lista de palabras ps que tienen longitud lon y poseen la letra l
-- en la posición pos (comenzando en 0). Por ejemplo,
--   ghci> buscaCrucigramaR 'c' 1 7 ["ocaso", "acabado", "ocupado"]
--   ["acabado", "ocupado"]
-- -----
```

```
buscaCrucigramaR :: Char -> Int -> Int -> [String] -> [String]
```

```
buscaCrucigramaR letra pos lon [] = []
buscaCrucigramaR letra pos lon (p:ps)
  | length p == lon && 0 <= pos && pos < length p && p !! pos == letra
  = p : buscaCrucigramaR letra pos lon ps
  | otherwise
  = buscaCrucigramaR letra pos lon ps
```

```

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

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

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

-----
-- Ejercicio 5.1. 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 5.2. 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 [] y n = []
    posicionesAux (x:xs) y n | x == y = n : posicionesAux xs y (n+1)
                              | otherwise = posicionesAux xs y (n+1)

```

```
-----  
-- Ejercicio 5.3. Comprobar con QuickCheck que ambas definiciones son  
-- 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 6.1. 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 [] ys = False  
contieneR xs ys = isPrefixOf ys xs || contieneR (tail xs) ys  
  
-----  
-- Ejercicio 6.2. Definir, por comprensión, la función  
-- contiene :: String -> String -> Bool  
-- tal que (contiene xs ys) se verifica si ys es una subcadena de  
-- xs. Por ejemplo,  
-- contiene "escasamente" "casa" == True  
-- contiene "escasamente" "cante" == False  
-- contiene "casado y casada" "casa" == True
```

```

-- contiene "" "" == True
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica
-- si ys es un prefijo de xs.
-----

contiene :: String -> String -> Bool
contiene xs ys =
  or [isPrefixOf ys 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 = isInfixOf ys xs

-----

-- Ejercicio 6.3. 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 8

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 cuyas  
-- transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-7.pdf
```

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

```
import Data.List  
import Test.QuickCheck
```

```
-- -----  
-- Ejercicio 1.1. Definir, por recursión, la función  
-- takeWhileR :: (a -> Bool) -> [a] -> [a]  
-- tal que (takeWhileR p xs) es la lista de los elemento de xs hasta el  
-- primero que no cumple la propiedad p. Por ejemplo,  
-- takeWhileR (<7) [2,3,9,4,5] == [2,3]  
-- -----
```

```
takeWhileR :: (a -> Bool) -> [a] -> [a]
```

```

takeWhileR _ [] = []
takeWhileR p (x:xs)
  | p x      = x : takeWhileR p xs
  | otherwise = []

-----

-- Ejercicio 1.2. Comprobar con QuickCheck que, para cualquier lista de
-- enteros xs, se verifica que (takeWhileR even xs) es igual que
-- (takeWhile even xs)
-----

-- La propiedad es
prop_takeWhileR :: [Int] -> Bool
prop_takeWhileR xs =
  takeWhileR even xs == takeWhile even xs

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

-----

-- Ejercicio 1.3. Comprobar con QuickCheck que, para cualquier lista de
-- enteros xs, se verifica que todos los elementos de (takeWhileR even xs)
-- son pares.
-----

prop_takeWhileTodos :: [Int] -> Bool
prop_takeWhileTodos xs =
  all even (takeWhileR even xs)

-----

-- Ejercicio 2.1. Definir por recursión la función
-- dropWhileR :: (a -> Bool) -> [a] -> [a]
-- tal que (dropWhileR p xs) es la lista de eliminando los elemento de xs
-- hasta el primero que cumple la propiedad p. Por ejemplo,
-- dropWhileR (<7) [2,3,9,4,5] == [9,4,5]
-----

dropWhileR :: (a -> Bool) -> [a] -> [a]
dropWhileR _ [] = []

```

```

dropWhileR p (x:xs)
  | p x      = dropWhileR p xs
  | otherwise = x:xs

-----

-- Ejercicio 2.2. Comprobar con QuickCheck que, para cualquier lista de
-- enteros xs, se verifica que la concatenación de (takeWhileR even xs)
-- y (dropWhileR even xs) es igual a xs.
-----

-- La propiedad es
prop_takeDrop :: [Int] -> Bool
prop_takeDrop xs =
  takeWhileR even xs ++ dropWhileR even xs == xs

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

-----

-- Ejercicio 3.1. La función
-- divideMedia :: [Double] -> ([Double],[Double])
-- dada una lista numérica, xs, calcula el par (ys,zs), donde ys
-- contiene los elementos de xs estrictamente menores que la media,
-- mientras que zs contiene los elementos de xs estrictamente mayores
-- que la media. Por ejemplo,
-- divideMedia [6,7,2,8,6,3,4] == ([2.0,3.0,4.0],[6.0,7.0,8.0,6.0])
-- divideMedia [1,2,3]         == ([1.0],[3.0])
-- Definir la función divideMedia por filtrado, comprensión y
-- recursión.
-----

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

-- (media xs) es la media de xs. Por ejemplo,
-- media [1,2,3] == 2.0
-- media [1,-2,3.5,4] == 1.625

```

```
-- Nota: En la definición de media se usa la función fromIntegral tal
-- que (fromIntegral x) es el número real correspondiente al número
-- entero x.
```

```
media :: [Double] -> Double
media xs = (sum xs) / fromIntegral (length xs)
```

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

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

```
-----
-- Ejercicio 3.2. Comprobar con QuickCheck que las tres definiciones
-- anteriores divideMediaF, divideMediaC y divideMediaR son
-- equivalentes.
-----
```

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

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

```
-----
-- Ejercicio 3.3. Comprobar con QuickCheck que si (ys,zs) es el par
```

```
-- obtenido aplicándole la función divideMediaF a xs, entonces la suma
-- de las longitudes de ys y zs es menor o igual que la longitud de xs.
-----

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

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

-----

-- Ejercicio 3.4. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces todos los
-- elementos de ys son menores que todos los elementos de zs.
-----

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

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

-----

-- Ejercicio 3.5. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces la
-- media de xs no pertenece a ys ni a zs.
--
-- Nota: Usar la función notElem tal que (notElem x ys) se verifica si y
-- no pertenece a ys.
-----

-- La propiedad es
```

```

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

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

-----

-- Ejercicio 4. Definir la función
-- segmentos :: (a -> Bool) -> [a] -> [[a]]
-- tal que (segmentos p xs) es la lista de los segmentos de xs cuyos
-- elementos verifican la propiedad p. Por ejemplo,
-- segmentos even [1,2,0,4,5,6,48,7,2] == [[],[2,0,4],[6,48],[2]]
-----

segmentos :: (a -> Bool) -> [a] -> [[a]]
segmentos _ [] = []
segmentos p xs =
  takeWhile p xs : (segmentos p (dropWhile (not.p) (dropWhile p xs)))

-----

-- Ejercicio 5. Definir la función
-- relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
-- relacionados (<) [2,3,7,9]           == True
-- relacionados (<) [2,3,1,9]          == False
-----

-- 1ª definición (por recursión):
relacionados :: (a -> a -> Bool) -> [a] -> Bool
relacionados r (x:y:zs) = (r x y) && relacionados r (y:zs)
relacionados _ _ = True

-- 2ª definición (por comprensión):
relacionados2 :: (a -> a -> Bool) -> [a] -> Bool
relacionados2 r xs = and [r x y | (x,y) <- zip xs (tail xs)]

```

```

-----
-- Ejercicio 6.1. 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 6.2. 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.

```

```

-----
-- Ejercicio 7.1. Definir por recursión la función
--   superpar :: Int -> Bool
-- tal que (superpar n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
--   superpar 426 == True
--   superpar 456 == False

```

```
-----
superpar :: Int -> Bool
superpar n | n < 10    = even n
           | otherwise = even n && superpar (n `div` 10)
```

```
-- Otra forma equivalente es
superpar_2 :: Int -> Bool
superpar_2 0 = True
superpar_2 n = even n && superpar_2 (div n 10)
```

```
-----
-- Ejercicio 7.2. Definir por comprensión la función
--   superpar2 :: Int -> Bool
-- tal que (superpar2 n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
--   superpar2 426 == True
--   superpar2 456 == False
-----
```

```
superpar2 :: Int -> Bool
superpar2 n = and [even d | d <- digitos n]
```

```
digitos :: Int -> [Int]
digitos n = [read [d] | d <- show n]
```

```
-----
-- Ejercicio 7.3. Definir, por recursión sobre los dígitos, la función
--   superpar3 :: Int -> Bool
-- tal que (superpar3 n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
--   superpar3 426 == True
--   superpar3 456 == False
-----
```

```
superpar3 :: Int -> Bool
superpar3 n = sonPares (digitos n)
  where sonPares []      = True
        sonPares (d:ds) = even d && sonPares ds
```



```
-----  
-- Ejercicio 7.4. Definir, usando all, la función  
--   superpar4 :: Int -> Bool  
-- tal que (superpar4 n) se verifica si n es un número par tal que todos  
-- sus dígitos son pares. Por ejemplo,  
--   superpar4 426 == True  
--   superpar4 456 == False  
-----
```

```
superpar4 :: Int -> Bool  
superpar4 n = all even (digitos n)
```

```
-----  
-- Ejercicio 7.5. Definir, usando filter, la función  
--   superpar5 :: Int -> Bool  
-- tal que (superpar5 n) se verifica si n es un número par tal que todos  
-- sus dígitos son pares. Por ejemplo,  
--   superpar5 426 == True  
--   superpar5 456 == False  
-----
```

```
superpar5 :: Int -> Bool  
superpar5 n = filter even (digitos n) == digitos n
```

```
-----  
-- Ejercicio 8.1. 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]  
-----
```

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

```
-----  
-- Ejercicio 8.2. Definir, usando foldr, la función  
--   concatP :: [[a]] -> [a]
```

```
-- tal que (concatP xss) es la concatenación de las listas de xss. Por
-- ejemplo,
--   concatP [[1,3],[2,4,6],[1,9]] == [1,3,2,4,6,1,9]
```

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

```
-- Ejercicio 8.3. 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 8.4. 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 9. Se considera la función
--   filtraAplica :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

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

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

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

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

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

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

```
-----
-- Ejercicio 10.1. 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))
```

```
-----
-- Ejercicio 10.2. 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 (maximumP xs) es el máximo de la lista xs. Por ejemplo,
-- maximumP [3,7,2,5] == 7
-- maximumP ["todo","es","falso"] == "todo"
-- maximumP ["menos","alguna","cosa"] == "menos"
-- Nota: La función maximumP es equivalente a la predefinida maximum.
```

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

```
-----
-- Ejercicio 10.3. Comprobar con QuickCheck que, para cualquier lista no
-- vacía xs, (maximumP xs) es un elemento de xs que es mayor o igual que
-- todos los elementos de xs.
```

```
-----
-- La propiedad es
prop_maximumP :: [Int] -> Property
prop_maximumP xs =
  xs /= [] ==> m 'elem' xs && all (<=m) xs
  where m = maximumP xs
```

```
-- La comprobación es
```

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

-----
-- Ejercicio 11.1. Definir, mediante plegado con foldr1, la función
--   minimunP :: Ord a => [a] -> a
-- tal que (minimunR xs) es el máximo de la lista xs. Por ejemplo,
--   minimunP [3,7,2,5]           == 7
--   minimunP ["todo","es","falso"] == "es"
--   minimunP ["menos","alguna","cosa"] == "alguna"
--
-- Nota: La función minimunP es equivalente a la predefinida minimum.
-----

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

-----
-- Ejercicio 11.2. Comprobar con QuickCheck que, para cualquier lista no
-- vacía xs, (minimumP xs) es un elemento de xs que es menor o igual que
-- todos los elementos de xs.
-----

-- La propiedad es
prop_minimumP :: [Int] -> Property
prop_minimumP xs =
  xs /= [] ==> m 'elem' xs && all (>=m) xs
  where m = minimumP xs

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



```
-- 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]]
```

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

```

-----
-- 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
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 = length (repiteFinita n x) == 0

-- 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
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]

-----

-- 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 xs = aux 1 xs
  where aux n [] = []
        aux n (x:xs) = replicate n x ++ aux (n+1) xs

-- 2ª definición
ecoR2 :: String -> String
ecoR2 [x] = [x]
ecoR2 xs = (ecoR2 . init) xs ++ repiteFinita (length xs) (last xs)

```

```

-----
-- 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 repite es equivalente a la función iterate definida
-- en el preludio de Haskell.
-----

```

```

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

```

```

-----
-- Ejercicio 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 n [] = []
agrupaR n xs = take n xs : agrupaR n (drop n xs)
```

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

```
-- La comprobación es
-- ghci> quickCheck prop_AgruparLongitud
-- 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_AgruparCombina :: Int -> [Int] -> Property
prop_AgruparCombina n xs =
  n > 0 ==> concat (agrupa n xs) == xs
```

```
-- La comprobación es
-- ghci> quickCheck prop_AgruparCombina
-- 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 n | even n    = n `div` 2
            | otherwise = 3*n+1
-----

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

collatzR :: Integer -> [Integer]
collatzR 1 = [1]

```

```
collatzR n = n : collatzR (siguiente n)
```

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

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

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

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

```
-- Otra definición alternativa es
```

```
menorCollatzSupera2 :: Integer -> Integer
menorCollatzSupera2 x = head [n | n <- [1..], t <- collatz n, t > x]
```



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

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

```
-----  
-- 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, x <= n-x, elem (n-x) primosN]
  where primosN = takeWhile (<=n) 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 factoriales2}
--   = 1 : 1 : take 2 (2 : zipWith (*) [3..] [R2])
--   = 1 : 1 : 2 : take 1 (zipWith (*) [3..] [2|R3])   {R3 es drop 3 factoriales2}
--   = 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 aux
  where aux = 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]
-----
```

```
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

```

$xs = \text{zipWith } (+) \text{ } ys \text{ } fs$	$ys = 1:xs$	$fs = 0:ys$
	$1:...$	$0:...$
	\wedge	\wedge
$1:...$	$1:1:...$	$0:1:1:...$
	\wedge	\wedge
$1:2:...$	$1:1:2:...$	$0:1:1:2:...$
	\wedge	\wedge
$1:2:3:...$	$1:1:2:3:...$	$0:1:1:2:3:...$
	\wedge	\wedge
$1:2:3:5:...$	$1:1:2:3:5:...$	$0:1:1:2:3:5:...$
	\wedge	\wedge
$1:2:3:5:8:...$	$1:1:2:3:5:8:...$	$0:1:1:2:3:5:8:...$


```

-- +-----+-----+-----+
-- 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)
-----

-- 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])

-- 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]:R1pascal)))
-- = [1] : (take 3 ((f [1]) : map R1pascal))
-- = [1] : (take 3 ((zipWith (+) (0:[1]) ([1]++[0]) : map R1pascal)))

```

```

-- = [1] : (take 3 ((zipWith (+) [0,1] [1,0]) : map R1pascal)))
-- = [1] : (take 3 ([1,1] : map R1pascal))
-- = [1] : [1,1] : (take 2 (map R1pascal))
-- = [1] : [1,1] : (take 2 (map ([1,1]:R2pascal)))
-- = [1] : [1,1] : (take 2 ((f [1,1]) : map R2pascal))
-- = [1] : [1,1] : (take 2 ((zipWith (+) (0:[1,1]) ([1,1]++[0]) : map R2pascal))
-- = [1] : [1,1] : (take 2 ((zipWith (+) [0,1,1] [1,1,0]) : map R2pascal))
-- = [1] : [1,1] : (take 2 ([1,2,1] : map R2pascal))
-- = [1] : [1,1] : [1,2,1] : (take 1 (map R2pascal))
-- = [1] : [1,1] : [1,2,1] : (take 1 (map ([1,2,1]:R3pascal)))
-- = [1] : [1,1] : [1,2,1] : (take 1 ((f [1,2,1]) : map R3pascal))
-- = [1] : [1,1] : [1,2,1] : (take 1 ((zipWith (+) (0:[1,2,1]) ([1,2,1]++[0])
-- = [1] : [1,1] : [1,2,1] : (take 1 ((zipWith (+) [0,1,2,1] [1,2,1,0]) : map
-- = [1] : [1,1] : [1,2,1] : (take 1 ([1,3,3,1] : map R3pascal))
-- = [1] : [1,1] : [1,2,1] : [1,3,3,1] : (take 0 (map R3pascal))
-- = [1] : [1,1] : [1,2,1] : [1,3,3,1] : []
-- = [[1],[1,1],[1,2,1],[1,3,3,1]]
-- en el cálculo con R1pascal, R2pascal y R3pascal es la el triángulo de
-- Pascal si el primero, los dos primeros o los tres primeros elementos,
-- respectivamente.

```


Relación 10

Tipos de datos algebraicos

Contenido

10.1Ejercicios propuestos	115
10.2Ejercicios de exámenes	125
10.2.1Árboles	125
10.2.2Expresiones	147
10.2.3Recorridos	155

10.1. Ejercicios propuestos

```
-- -----  
-- Introducción                                     --  
--  
-- En esta relación se presenta ejercicios sobre árboles binarios  
-- definidos como tipos de datos algebraicos.  
--  
-- Los ejercicios corresponden al tema 9 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-9.pdf  
--  
-- § 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
--      / \
--     /   \
--    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. 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 x i d) = nHojas i + nHojas d
```

```
-----
-- Ejercicio 2.1. 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 x i d) = 1 + nNodos i + nNodos d
```

```
-- Ejercicio 2.2. Comprobar con QuickCheck que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
```

```
-- La propiedad es
prop_nHojas :: Arbol Int -> Bool
prop_nHojas x =
  nHojas x == nNodos x + 1
```

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

```
-- Ejercicio 3.1. 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 :: Arbol a -> Int
profundidad (H _)      = 0
profundidad (N x i d) = 1 + max (profundidad i) (profundidad d)
```

```
-- Ejercicio 3.2. Comprobar con QuickCheck que para todo árbol binario
-- 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 4.1. Definir la función
```

```
-- preorden :: Arbol a -> [a]
```

```
-- tal que (preorden x) es la lista correspondiente al recorrido
```

```
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
```

```
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
```

```
-- subárbol derecho. Por ejemplo,
```

```
-- 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 4.2. 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 5. Definir la función
```

```
-- postorden :: Arbol a -> [a]
```



```

-- tal que (postorden x) es la lista correspondiente al recorrido
-- postorden del árbol x; es decir, primero recorre el subárbol
-- izquierdo, a continuación el subárbol derecho y, finalmente, la raíz
-- del árbol. Por ejemplo,
--   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]

```

```

-----
-- Ejercicio 6.1. 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 6.2. 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 7.1. 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
espejo (N x i d) = N x (espejo d) (espejo i)

-----

-- Ejercicio 7.2. 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 7.3. 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 7.4. 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)

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

-----

-- Ejercicio 8.1. 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 i d) = H x
takeArbol n (N x i d) =
  N x (takeArbol (n-1) i) (takeArbol (n-1) d)

```

```

-----
-- Ejercicio 8.2. 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 9. La función
-- repeat :: a -> [a]
-- está definida de forma que (repeat x) es la lista formada por
-- infinitos elementos x. Por ejemplo,
-- repeat 3 == [3,3,3,3,3,3,3,3,3,3,3,3,3,3,...]
-- 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 10.1. 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)
--   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 10.2. 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 11.1. 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 11.2. Comprobar con QuickCheck que
--   (mapArbol (1+)) . espejo = espejo . (mapArbol (1+))
-----
```

```
-- 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 11.3. 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      = liftM H arbitrary
    arbol n | n>0 = oneof [liftM H arbitrary,
                          liftM3 N arbitrary subarbol subarbol]
                          where subarbol = arbol (div n 2)

```

10.2. Ejercicios de exámenes

10.2.1. Árboles

10.2.1.1. Árboles binarios con valores en los nodos

```

-----
-- En los ejercicios de esta sección se usará el tipo de árboles
-- binarios definidos como sigue
--   data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-----

```

```

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

```

```

-----
-- Ejercicio 1. [12 de abril de 2010] Definir por recursión la función
--   nodos :: Arbol a -> [a]
-- tal que (nodos x) es la lista de los nodos del árbol x. Por ejemplo.
--   nodos (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H)) == [2,5,3,7,4]
-----

```

```

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

```

```

-----
-- Ejercicio 2. [12 de abril de 2010] Definir por recursión la función
-- sumaArbol :: Num a => Arbol a -> a
-- tal (sumaArbol x) es la suma de los valores que hay en el árbol
-- x. Por ejemplo,
-- sumaArbol (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H)) == 21
-----

```

```

sumaArbol :: Num a => Arbol a -> a
sumaArbol H           = 0
sumaArbol (N x i d) = x + sumaArbol i + sumaArbol d

```

```

-----
-- Ejercicio 3. [16 de marzo de 2010] Definir la función
-- mapArbol :: (a -> b) -> Arbol a -> Arbol 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) (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H))
-- N 3 (N 6 (N 4 H H) (N 8 H H)) (N 5 H H)
-----

```

```

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

```

```

-----
-- Ejercicio 4. [29 de febrero de 2012] Definir la función
-- ramaIzquierda :: Arbol t -> [t]
-- tal que (ramaIzquierda a) es la lista de los valores de los nodos de
-- la rama izquierda del árbol a. Por ejemplo,
-- ramaIzquierda (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H)) == [2,5,3]
-----

```

```

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

```

```

-----
-- Ejercicio 5. [7 de mayo de 2012] Diremos que un árbol está
-- balanceado si para cada nodo v la diferencia entre el número de nodos

```



```

-- (con valor) de sus ramas izquierda y derecha es menor o igual que uno.
--
-- Definir la función
--   balanceado :: Arbol a -> Bool
-- tal que (balanceado a) se verifica si el árbol a está balanceado. Por
-- ejemplo,
--   balanceado (N 5 H (N 3 H H))           == True
--   balanceado (N 5 H (N 3 (N 4 H H) H)) == False

```

```

balanceado :: Arbol a -> Bool
balanceado H           = True
balanceado (N _ i d) = abs (numeroNodos i - numeroNodos d) <= 1

```

```

-- (numeroNodos a) es el número de nodos del árbol a. Por ejemplo,
--   numeroNodos (N 5 H (N 3 H H)) == 2

```

```

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

```

```

-- -----
-- Ejercicio 6. [7 de marzo de 2012]. Definir la función
--   cuentaArbol :: Eq t => Arbol t -> t -> Int
-- tal que (cuentaArbol a x) es el número de veces aparece x en el árbol
-- a. Por ejemplo,
--   cuentaArbol (N 5 (N 5 H H) H) 5 == 2
--   cuentaArbol (N 5 (N 5 H H) H) 7 == 0
--   cuentaArbol (N 5 (N 7 H H) H) 7 == 1

```

```

cuentaArbol :: Eq t => Arbol t -> t -> Int
cuentaArbol H _ = 0
cuentaArbol (N x i d) y
  | x == y    = 1 + c1 + c2
  | otherwise = c1 + c2
  where c1 = cuentaArbol i y
        c2 = cuentaArbol d y

```

```

-- -----
-- Ejercicio 7. [13 de junio de 2013] Un árbol binario ordenado es un

```

```

-- árbol binario (ABO) en el que los valores de cada nodo es mayor o
-- igual que los valores de sus hijos. Por ejemplo, de los siguientes
-- árboles
--
--      9              9
--     / \            / \
--    /   \          /   \
--   8     6         8     6
--  / \   / \       / \   / \
-- 3  2 4  5       3  2 4  7
--
-- el de la izquierda es ordenado y el de la derecha no lo es.
--
-- Definir la función esABO
--   esABO :: Ord t => Arbol t -> Bool
-- tal que (esABO a) se verifica si a es un árbol binario ordenado. Por
-- ejemplo.
--   ghci> esABO (N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H)))
--   True
--   ghci> esABO ( N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 7 H H)))
--   False

```

-- 1ª definición

```

esABO :: Ord a => Arbol a -> Bool
esABO H = True
esABO (N x H H) = True
esABO (N x m1@(N x1 a1 b1) H) = x >= x1 && esABO m1
esABO (N x H m2@(N x2 a2 b2)) = x >= x2 && esABO m2
esABO (N x m1@(N x1 a1 b1) m2@(N x2 a2 b2)) =
  x >= x1 && esABO m1 && x >= x2 && esABO m2

```

-- 2ª definición

```

esABO2 :: Ord a => Arbol a -> Bool
esABO2 H = True
esABO2 (N x i d) = mayor x i && mayor x d && esABO2 i && esABO2 d
  where mayor x H = True
        mayor x (N y _ _) = x >= y

```

-- Ejercicio 8. [15 de mayo de 2014]. Definir la función
-- takeArbolWhile :: (a -> Bool) -> Arbol a -> Arbol a

```
-- tal que (takeArbolWhile p ar) es el subárbol de ar empezando desde la
-- raíz mientras se verifique p. Por ejemplo,
--   takeArbolWhile odd arbol1  == N 5 H (N 7 H H)
--   takeArbolWhile even arbol1 == H
--   takeArbolWhile (< 6) arbol1 == N 5 (N 4 (N 1 H H) H) H
-- donde arbol1 está definido por
--   arbol1 = N 5 (N 4 (N 1 H H) H ) (N 7 H (N 8 H H))
-----
```

```
arbol1 = N 5 (N 4 (N 1 H H) H ) (N 7 H (N 8 H H))
```

```
takeArbolWhile :: (a -> Bool) -> Arbol a -> Arbol a
takeArbolWhile p H = H
takeArbolWhile p (N a x y)
  | p a      = N a (takeArbolWhile p x) (takeArbolWhile p y)
  | otherwise = H
```

10.2.1.2. Árboles binarios con valores en las hojas

```
-----
-- En los ejercicios de esta sección se usará el tipo de árboles
-- binarios definidos como sigue
--   data Arbol a = H a
--                 | N (Arbol a) (Arbol 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 :: Arbol Int
--   arbol1 = N (H 1) (N (H 2) (H 3))
--   arbol2 = N (N (H 1) (H 2)) (H 3)
--   arbol3 = N (N (H 1) (H 4)) (H 3)
--   arbol4 = N (N (H 2) (H 3)) (H 1)
-----
```

```
data Arbol a = N (Arbol a) (Arbol a)
            | H a
            deriving Show
```

```
arbol1, arbol2, arbol3, arbol4 :: Arbol Int
arbol1 = N (H 1) (N (H 2) (H 3))
arbol2 = N (N (H 1) (H 2)) (H 3)
arbol3 = N (N (H 1) (H 4)) (H 3)
arbol4 = N (N (H 2) (H 3)) (H 1)
```

```
-----
-- Ejercicio 1. [17 diciembre de 2010] Definir la función
--   igualBorde :: Eq a => Arbol a -> Arbol 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
-----
```

```
igualBorde :: Eq a => Arbol a -> Arbol 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,
--   borde arbol4 == [2,3,1]
borde :: Arbol a -> [a]
borde (N i d) = borde i ++ borde d
borde (H x)   = [x]
```

10.2.1.3. Árboles binarios con valores en las hojas y en los nodos

```
-----
-- En los ejercicios de esta sección se usará el tipo de árboles
-- binarios definidos como sigue
--   data Arbol a = H a
--               | N a (Arbol a) (Arbol a)
--               deriving Show
-----
```

```
import Data.List (nub)
```

```
data Arbol a = H a
             | N a (Arbol a) (Arbol a)
             deriving Show
```

```
-----
-- Ejercicio 1. [15 de abril de 2011] Definir la función
--   arbol2Lista :: Arbol t -> [t]
-- tal que (arbol2Lista a) es la lista de los valores del árbol a. Por
-- ejemplo,
--   ghci> arbol2Lista (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
--   [5,3,1,4,7,6,9]
-----
```

```
arbol2Lista :: Arbol t -> [t]
arbol2Lista (H x)      = [x]
arbol2Lista (N x i d) = x : (arbol2Lista i ++ arbol2Lista d)
```

```
-----
-- Ejercicio 2. [15 de abril de 2011] Definir la función
--   minArbol :: Ord t => Arbol t -> t
-- tal que (minArbol a) es el menor elemento de a. Por ejemplo,
--   minArbol (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9))) == 1
-----
```

```
minArbol :: Ord t => Arbol t -> t
minArbol = minimum . arbol2Lista
```

```
-----
-- Ejercicio 3. [10 de Diciembre de 2012] Definir la función
--   aplica :: (t -> t) -> Arbol t -> Arbol t
-- tal que (aplica f a) es el árbol obtenido aplicando la función f a
-- los elementos del árbol a. Por ejemplo,
--   ghci> aplica (+2) (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
--   N 7 (N 5 (H 3) (H 6)) (N 9 (H 8) (H 11))
--   ghci> aplica (*5) (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
--   N 25 (N 15 (H 5) (H 20)) (N 35 (H 30) (H 45))
-----
```

```

aplica :: (t -> t) -> Arbol t -> Arbol t
aplica f (H x)      = H (f x)
aplica f (N x i d) = N (f x) (aplica f i) (aplica f d)

```

```

-----
-- Ejercicio 4. [13 de junio de 2013]. Definir la función
--   nodosInternos :: Arbol t -> [t]
-- tal que (nodosInternos a) es la lista de los nodos internos del
-- árbol a. Por ejemplo,
--   nodosInternos (N 3 (H 5) ((N 4 (H 6) (H 7)))) == [3,4]
-----

```

```

nodosInternos (H _)      = []
nodosInternos (N x i d) = x : (nodosInternos i ++ nodosInternos d)

```

```

-----
-- Ejercicio 5. [21 de marzo de 2013] Definir la función
--   ramas :: Arbol a -> [[a]]
-- tal que (ramas a) es la lista de las ramas del árbol. Por ejemplo,
--   ramas (N 3 (H 5) ((N 4 (H 6) (H 7)))) == [[3,5],[3,4,6],[3,4,7]]
-----

```

```

ramas :: Arbol a -> [[a]]
ramas (H x)      = [[x]]
ramas (N x i d) = [x:r | r <- ramas i ++ ramas d]

```

```

-----
-- Ejercicio 6. [13 de septiembre de 2013] Definir la función
--   algunoArbol :: Arbol t -> (t -> Bool) -> Bool
-- tal que (algunoArbol a p) se verifica si algún elemento del árbol a
-- cumple la propiedad p. Por ejemplo,
--   algunoArbol (N 5 (N 3 (H 1) (H 4)) (H 2)) (>4) == True
--   algunoArbol (N 5 (N 3 (H 1) (H 4)) (H 2)) (>7) == False
-----

```

```

algunoArbol :: Arbol a -> (a -> Bool) -> Bool
algunoArbol (H x) p      = p x
algunoArbol (N x i d) p = p x || algunoArbol i p || algunoArbol d p

```

```
-- Ejercicio 7. [4 de diciembre de 2014]. Definir la función
-- allArbol :: (t -> Bool) -> Arbol t -> Bool
-- tal que (allArbol p a) se verifica si todos los elementos del árbol
-- verifican p. Por ejemplo,
-- allArbol (>0) (N 9 (N 3 (H 2) (H 4)) (H 7)) == True
-- allArbol even (N 9 (N 3 (H 2) (H 4)) (H 7)) == False
```

```
-----
allArbol :: (t -> Bool) -> Arbol t -> Bool
allArbol p (H x)      = p x
allArbol p (N r i d) = p r && allArbol p i && allArbol p d
```

```
-----
-- Ejercicio 8. [22 de marzo de 2013]. Se puede observar en el
-- siguiente dibujo que los árboles arbol1 y arbol2 tiene la misma
-- estructura, pero los árboles arbol1 y arbol3 o arbol1 y arbol4 no la
-- tienen.
```

```
--           5             8             5             5
--          / \          / \          / \          / \
--         /   \        /   \        /   \        /   \
--        9     7       9     3       9     2       4     7
--       / \   / \     / \   / \     / \           / \
--      1 4 6 8 1 4 6 2 1 4           6     2
```

```
-- Su representación es
```

```
-- arbol1, arbol2, arbol3, arbol4 :: Arbol Int
-- arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
-- arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
-- arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
-- arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))
```

```
-- Definir la función
```

```
-- igualEstructura :: Arbol -> Arbol -> Bool
-- tal que (igualEstructura a1 a2) se verifica si los árboles a1 y a2
-- tienen la misma estructura. Por ejemplo,
-- igualEstructura arbol1 arbol2 == True
-- igualEstructura arbol1 arbol3 == False
-- igualEstructura arbol1 arbol4 == False
```

```
-----
arbol1, arbol2, arbol3, arbol4 :: Arbol Int
```

```

arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))

```

```

igualEstructura :: Arbol a -> Arbol a -> Bool
igualEstructura (H _) (H _) = True
igualEstructura (N r1 i1 d1) (N r2 i2 d2) =
    igualEstructura i1 i2 && igualEstructura d1 d2
igualEstructura _ _ = False

```

```

-----
-- Ejercicio 9. [19 de marzo de 2014]. 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 -> Arbol a -> [a]
-- tal que (nivel k a) es la lista de los elementos de nivel k del árbol
-- a. Por ejemplo,
-- nivel 0 (N 7 (N 2 (H 5) (H 4)) (H 9)) == [7]
-- nivel 1 (N 7 (N 2 (H 5) (H 4)) (H 9)) == [2,9]
-- nivel 2 (N 7 (N 2 (H 5) (H 4)) (H 9)) == [5,4]
-- nivel 3 (N 7 (N 2 (H 5) (H 4)) (H 9)) == []
-----

```

```

nivel :: Int -> Arbol a -> [a]
nivel 0 (H x) = [x]
nivel 0 (N x _ _) = [x]
nivel k (H _) = []
nivel k (N _ i d) = nivel (k-1) i ++ nivel (k-1) d

```

```

-----
-- Ejercicio 10. [20 de marzo de 2014]. Definir la función
-- transforma :: (t -> Bool) -> Arbol t -> Arbol (Maybe t)
-- tal que (transforma p a) es el árbol con la misma estructura que a,
-- en el que cada elemento x que verifica el predicado p se sustituye por
-- (Just x) y los que no lo verifican se sustituyen por Nothing. Por
-- ejemplo,
-- ghci> transforma even (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
-- N Nothing
-----

```



```

--      (N Nothing (H Nothing) (H (Just 4)))
--      (N Nothing (H (Just 6)) (H Nothing))
-----

transforma :: (t -> Bool) -> Arbol t -> Arbol (Maybe t)
transforma p (H r) | p r      = H (Just r)
                  | otherwise = H Nothing
transforma p (N r i d)
  | p r      = N (Just r) (transforma p i) (transforma p d)
  | otherwise = N Nothing (transforma p i) (transforma p d)
-----

-- Ejercicio 11. [19 de marzo de 2014]. Definir la función
-- todosDistintos :: Eq a => Arbol a -> Bool
-- tal que (todosDistintos a) se verifica si todos los elementos del
-- árbol a son distintos entre sí. Por ejemplo,
-- todosDistintos (N 7 (H 3) (N 4 (H 9) (H 2))) == True
-- todosDistintos (N 7 (H 3) (N 4 (H 7) (H 2))) == False
-----

todosDistintos :: Eq a => Arbol a -> Bool
todosDistintos a = xs == nub xs
  where xs = preorden a

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

-- Ejercicio 12. [24 de marzo de 2014]. Definir la función
-- profundidades :: (Num t, Eq t) => Arbol t -> t -> [t]
-- tal que (profundidades a x) es la lista de las profundidades que ocupa x
-- en el árbol a. Por ejemplo,
-- profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 1 == [1,2,3]
-- profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 2 == [3]
-- profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 3 == [2]
-- profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 4 == []
-----

profundidades :: (Num t, Eq t) => Arbol t -> t -> [t]

```

```

profundidades (H y) x | x == y    = [1]
                    | otherwise = []
profundidades (N y i d) x
  | x == y    = 1:[n+1 | n <- profundidades i x ++ profundidades d x]
  | otherwise = [n+1 | n <- profundidades i x ++ profundidades d x]

```

-- *Ejercicio 13. [20 de marzo de 2014]. Las capas de un árbol binario son las listas de elementos que están a la misma profundidad. Por ejemplo, las capas del árbol*

```

--      1
--     / \
--    /   \
--   3     2
--  / \   / \
-- 5  4 6  7
-- son: [1], [3,2] y [5,4,6,7]
--

```

-- *Definir la función*

```

--   capas :: Arbol a -> [[a]]
-- tal que (capas a) es la lista de las capas de dicho árbol ordenadas
-- según la profundidad. Por ejemplo,
--   capas (N 3 (H 1) (H 4))           == [[3],[1,4]]
--   capas (N 2 (H 3) (H 5))           == [[2],[3,5]]
--   capas (N 1 (H 2) (N 2 (H 3) (H 3))) == [[1],[2,2],[3,3]]
--   capas (N 1 (N 2 (H 3) (H 5)) (H 2)) == [[1],[2,2],[3,5]]
--

```

```

capas :: Arbol a -> [[a]]
capas (H n) = [[n]]
capas (N n i d) = [n] : union (capas i) (capas d)

```

-- *(union xss yss) es la lista obtenida concatenando los correspondientes elementos de xss e yss. Por ejemplo,*

```

--   union [[3,4],[2]] [[5],[7,6,8]] == [[3,4,5],[2,7,6,8]]
--   union [[3,4]]      [[5],[7,6,8]] == [[3,4,5],[7,6,8]]
--   union [[3,4],[2]] [[5]]          == [[3,4,5],[2]]

```

```

union :: [[a]] -> [[a]] -> [[a]]
union [] yss = yss
union xss [] = xss

```

```
union (xs:xss) (ys:yss) = (xs ++ ys) : union xss yss
```

```
-----
-- Ejercicio 14. [18 de marzo de 2013]. Definir el predicado
-- contenido :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (contenido a1 a2) es verdadero si todos los elementos que
-- aparecen en el árbol a1 también aparecen en el árbol a2. Por ejemplo,
-- contenido ej1 ej2 == True
-- contenido ej2 ej1 == False
-- donde ej1 y ej2 están definidos por
-- ej1, ej2 :: Arbol Int
-- ej1 = N 9 (N 8 (H 3) (H 2)) (N 8 (H 4) (H 5))
-- ej2 = N 9 (N 4 (H 3) (H 2)) (N 8 (H 5) (H 7))
-----
```

```
ej1, ej2 :: Arbol Int
```

```
ej1 = N 9 (N 8 (H 3) (H 2)) (N 8 (H 4) (H 5))
```

```
ej2 = N 9 (N 4 (H 3) (H 2)) (N 8 (H 5) (H 7))
```

```
contenido :: Eq a => Arbol a -> Arbol a -> Bool
```

```
contenido (H x) a = pertenece x a
```

```
contenido (N x i d) a = pertenece x a && contenido i a && contenido d a
```

```
-- (pertenece x a) se verifica si x pertenece al árbol a. Por ejemplo,
-- pertenece 8 ej1 == True
-- pertenece 7 ej1 == False
pertenece x (H y) = x == y
pertenece x (N y i d) = x == y || pertenece x i || pertenece x d
```

```
-----
-- Ejercicio 15. [3 de julio de 2013]. Definir la función
-- esSubarbol :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (esSubarbol a1 a2) se verifica si a1 es un subárbol de
-- a2. Por ejemplo,
-- esSubarbol (H 2) (N 2 (H 2) (H 4)) == True
-- esSubarbol (H 5) (N 2 (H 2) (H 4)) == False
-- esSubarbol (N 2 (H 2) (H 4)) (N 2 (H 2) (H 4)) == True
-- esSubarbol (N 2 (H 4) (H 2)) (N 2 (H 2) (H 4)) == False
-----
```

```

esSubarbol :: Eq a => Arbol a -> Arbol a -> Bool
esSubarbol (H x) (H y) = x == y
esSubarbol a@(H x) (N y i d) = esSubarbol a i || esSubarbol a d
esSubarbol (N _ _ _) (H _) = False
esSubarbol a@(N r1 i1 d1) (N r2 i2 d2)
  | r1 == r2 = (igualArbol i1 i2 && igualArbol d1 d2) ||
                esSubarbol a i2 || esSubarbol a d2
  | otherwise = esSubarbol a i2 || esSubarbol a d2

```

-- (igualArbol a1 a2) se verifica si los árboles a1 y a2 son iguales.

```

igualArbol :: Eq a => Arbol a -> Arbol a -> Bool
igualArbol (H x) (H y) = x == y
igualArbol (N r1 i1 d1) (N r2 i2 d2) =
  r1 == r2 && igualArbol i1 i2 && igualArbol d1 d2
igualArbol _ _ = False

```

```

-----
-- Ejercicio 16. [10 de septiembre de 2012] Un árbol binario ordenado
-- (ABO) es un árbol binario tal que para cada nodo, los elementos de su
-- subárbol izquierdo son menores y los de su subárbol derecho son
-- mayores. Por ejemplo,
--
--      5
--     / \
--    /   \
--   3     7
--  / \   / \
-- 1  4 6  9
--
-- es una ABO y se representa por
--   ejArbol2 = N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9))
--
-- Definir la función
--   ancestroMasProximo :: Int -> Int -> Arbol Int -> Int
-- tal que (ancestroMasProximo x y a) es el ancestro más próximo de los
-- nodos x e y en el árbol a. Por ejemplo,
--   ancestroMasProximo 4 1 ejArbol2 == 3
--   ancestroMasProximo 4 6 ejArbol2 == 5
-----

```

```

ejArbol2 :: Arbol Int

```

```
ejArbol2 = N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9))
```

```
ancestroMasProximo :: Int -> Int -> Arbol Int -> Int
```

```
ancestroMasProximo x y (N z i d)
  | x < z && y < z = ancestroMasProximo x y i
  | x > z && y > z = ancestroMasProximo x y d
  | otherwise     = z
```

```
-----
-- Ejercicio 17. [4 de julio de 2014] 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
--
-- Los árboles se representarán por
--   data Arbol = H Int
--             | N Int Arbol Arbol
--             deriving Show
--
-- Definir la función
--   arbolFactorizacion :: Int -> Arbol
-- tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
-- ejemplo,
--   arbolFactorizacion 60 == N 60 (N 6 (H 2) (H 3)) (N 10 (H 2) (H 5))
--   arbolFactorizacion 45 == N 45 (H 5) (N 9 (H 3) (H 3))
```

```

--   arbolFactorizacion 7  == H 7
--   arbolFactorizacion 14 == N 14 (H 2) (H 7)
--   arbolFactorizacion 28 == N 28 (N 4 (H 2) (H 2)) (H 7)
--   arbolFactorizacion 84 == N 84 (H 7) (N 12 (H 3) (N 4 (H 2) (H 2)))
--   -----

-- 1ª definición
-- =====
arbolFactorizacion :: Int -> Arbol Int
arbolFactorizacion n
  | esPrimo n = H n
  | otherwise = N 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 :: 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 -> Arbol Int
arbolFactorizacion2 n
  | x == 1    = H n
  | otherwise = N 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)
divisoresMedio2 n = (n `div` x,x)
    where m = ceiling (sqrt (fromIntegral n))
          x = head [y | y <- [m..n], n `rem` y == 0]

-----

-- Ejercicio 18. [13 de junio de 2013]. Definir la función
--   ramaIguales :: Eq t => Arbol t -> Bool
-- tal que (ramaIguales a) se verifica si el árbol a contiene al menos
-- una rama tal que todos sus elementos son iguales. Por ejemplo,
--   ramaIguales (N 3 (H 5) (H 3))           == True
--   ramaIguales (N 3 (H 5) (H 4))           == False
--   ramaIguales (N 3 (H 5) ((N 3 (H 3) (H 7)))) == True
-----

-- 1ª definición:
ramaIguales :: Eq a => Arbol a -> Bool
ramaIguales (H _)      = True
ramaIguales (N x i d) = aux x i || aux x d
    where aux x (H y)    = x == y
          aux x (N y i d) = x == y && (aux x i || aux x d)

-- 2ª definición:
ramaIguales2 :: Eq a => Arbol a -> Bool
ramaIguales2 a = or [iguales xs | xs <- ramas a]

-- (iguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--   iguales [5,5,5] == True
--   iguales [5,2,5] == False
iguales :: Eq a => [a] -> Bool
iguales (x:y:xs) = x == y && iguales (y:xs)
iguales _       = True

```

```
-- 2ª definición de iguales (por comprensión):
```

```
iguales2 :: Eq a => [a] -> Bool
```

```
iguales2 [] = True
```

```
iguales2 (x:xs) = and [x == y | y <- xs]
```

```
-- 3ª definición de iguales (usando nub):
```

```
iguales3 :: Eq a => [a] -> Bool
```

```
iguales3 xs = length (nub xs) <= 1
```

```
-- 3ª solución:
```

```
ramaIguales3 :: Eq a => Arbol a -> Bool
```

```
ramaIguales3 = any iguales . ramas
```

```
-----
```

```
-- Ejercicio 19. [21 de mayo de 2014]. Definir el predicado
```

```
--   ramasDis :: Eq a => Arbol a -> Bool
```

```
-- tal que (ramasDis t) se verifica si todas las ramas del árbol t
```

```
-- contienen, al menos, dos elementos distintos. Por ejemplo:
```

```
--   ramasDis (N 3 (H 5) (H 4))           == True
```

```
--   ramasDis (N 3 (H 5) (H 3))           == False
```

```
--   ramasDis (N 3 (H 5) ((N 3 (H 3) (H 7)))) == False
```

```
-----
```

```
ramasDis :: Eq a => Arbol a -> Bool
```

```
ramasDis = not . ramaIguales
```

```
-----
```

```
-- Ejercicio 20. [3 de abril de 2013]. Decimos que un árbol binario es
```

```
-- par si la mayoría de sus nodos son pares e impar en caso contrario.
```

```
--
```

```
-- Para representar la paridad se define el tipo Paridad
```

```
--   data Paridad = Par | Impar deriving (Eq, Show)
```

```
--
```

```
-- Definir la función
```

```
--   paridad :: Arbol Int -> Paridad
```

```
-- tal que (paridad a) es la paridad del árbol a. Por ejemplo,
```

```
--   paridad (N 8 (N 6 (H 3) (H 4)) (H 5)) == Par
```

```
--   paridad (N 8 (N 9 (H 3) (H 4)) (H 5)) == Impar
```

```
-----
```



```
data Paridad = Par | Impar deriving (Eq, Show)
```

```
paridad :: Arbol Int -> Paridad
paridad a | x > y      = Par
          | otherwise  = Impar
          where (x,y) = paridades a
```

```
-- (paridades a) es un par (x,y) donde x es el número de valores pares
-- en el árbol a e i es el número de valores impares en el árbol a. Por
-- ejemplo,
```

```
-- paridades (N (N (H 3) 6 (H 4)) 8 (H 5)) == (3,2)
-- paridades (N (N (H 3) 9 (H 4)) 8 (H 5)) == (2,3)
```

```
paridades :: Arbol Int -> (Int,Int)
paridades (H x) | even x    = (1,0)
               | otherwise  = (0,1)
paridades (N x i d) | even x    = (1+a1+a2,b1+b2)
                   | otherwise  = (a1+a2,1+b1+b2)
                   where (a1,b1) = paridades i
                         (a2,b2) = paridades d
```

10.2.1.4. Árboles binarios de búsqueda

```
-- -----
-- Un árbol binario de búsqueda (ABB) es un árbol binario tal que el de
-- cada nodo es mayor que los valores de su subárbol izquierdo y es
-- menor que los valores de su subárbol derecho y, además, ambos
-- subárboles son árboles binarios de búsqueda. Por ejemplo, al
-- almacenar los valores de [8,4,2,6,3] en un ABB se puede obtener el
-- siguiente ABB:
```

```
--
--      5
--     / \
--    /   \
--   2     6
--      / \
--     4   8
```

```
-- Los ABB se pueden representar como tipo de dato algebraico:
```

```
-- data ABB = H
--          | N Int ABB ABB
```

```
--           deriving (Eq, Show)
-- Por ejemplo, la definición del ABB anterior es
--   ej :: ABB
--   ej = N 3 (N 2 H H) (N 6 (N 4 H H) (N 8 H H))
-----
```

```
data ABB = H
  | N Int ABB ABB
  deriving (Eq, Show)
```

```
ej :: ABB
ej = N 3 (N 2 H H) (N 6 (N 4 H H) (N 8 H H))
```

```
-----
-- Ejercicio 1. [[22 de noviembre de 2011] Definir la función
--   inserta :: Int -> ABB -> ABB
-- tal que (inserta v a) es el árbol obtenido añadiendo el valor v al
-- ABB a, si no es uno de sus valores. Por ejemplo,
--   ghci> inserta 5 (N 3 (N 2 H H) (N 6 (N 4 H H) (N 8 H H)))
--   N 3 (N 2 H H) (N 6 (N 4 H (N 5 H H)) (N 8 H H))
--   ghci> inserta 1 (N 3 (N 2 H H) (N 6 (N 4 H H) (N 8 H H)))
--   N 3 (N 2 (N 1 H H) H) (N 6 (N 4 H H) (N 8 H H))
--   ghci> inserta 2 (N 3 (N 2 H H) (N 6 (N 4 H H) (N 8 H H)))
--   N 3 (N 2 H H) (N 6 (N 4 H H) (N 8 H H))
-----
```

```
inserta :: Int -> ABB -> ABB
inserta v' H = N v' H H
inserta v' (N v i d)
  | v' == v   = N v i d
  | v' < v   = N v (inserta v' i) d
  | otherwise = N v i (inserta v' d)
```

10.2.1.5. Árboles booleanos

```
-----
-- Ejercicio 1. [18 de junio de 2014]. Se consideran los árboles con
-- operaciones booleanas definidos por
--   data ArbolB = H Bool
--               | Conj ArbolB ArbolB
```

```

--          | Disy ArbolB ArbolB
--          | Neg ArbolB
--
-- Por ejemplo, los árboles
--
--          Conj
--         /  \
--        /    \
--       Disy   Conj
--      /  \   /  \
--     Conj Neg Neg True
--    /  \  |   |
--   True False False False
--
--          Conj
--         /  \
--        /    \
--       Disy   Conj
--      /  \   /  \
--     Conj  Neg Neg True
--    /  \  |   |
--   True False True False
--
-- se definen por
-- ej1, ej2:: ArbolB
-- ej1 = Conj (Disy (Conj (H True) (H False))
--              (Neg (H False)))
--        (Conj (Neg (H False))
--              (H True))
--
-- ej2 = Conj (Disy (Conj (H True) (H False))
--              (Neg (H True)))
--        (Conj (Neg (H False))
--              (H True))
--
-- Definir la función
-- valor:: ArbolB -> Bool
-- tal que (valor ar) es el resultado de procesar el árbol realizando
-- las operaciones booleanas especificadas en los nodos. Por ejemplo,
-- valor ej1 == True
-- valor ej2 == False
-- -----

```

data ArbolB = H Bool

```

| Conj ArbolB ArbolB
| Disy ArbolB ArbolB
| Neg ArbolB

```

ej1, ej2:: ArbolB

```

ej1 = Conj (Disy (Conj (H True) (H False))
            (Neg (H False)))
      (Conj (Neg (H False))
            (H True))

```

```

        (Neg (H False)))
    (Conj (Neg (H False))
          (H True))

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

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

```

10.2.1.6. Árboles generales

```

-----
-- Ejercicio 1. [19 de mayo de 2014]. Los árboles se pueden
-- representar mediante el siguiente tipo de dato
--   data Arbol a = N a [Arbol a]
--               deriving 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
--
-- se representan por
--   ej1, ej2, ej3 :: Arbol Int
--   ej1 = N 1 [N 2 [], N 3 [N 4 []]]
--   ej2 = N 3 [N 5 [N 6 []],
--             N 4 [],
--             N 7 [N 2 [], N 1 []]]

```

```

--      ej3 = 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 :: Arbol a -> Arbol a -> (a -> Bool) -> Arbol 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 (N 3 [N 5 [N 6 []],N 4 [],N 7 [N 2 [],N 1 []]]) (N 8 []) (>5)
--      N 3 [N 5 [N 6 [N 8 []]],N 4 [],N 7 [N 2 [],N 1 [],N 8 []]]
-----

```

```

data Arbol a = N a [Arbol a]
              deriving Show

```

```

ej1, ej2,ej3 :: Arbol Int
ej1 = N 1 [N 2 [],N 3 [N 4 []]]
ej2 = N 3 [N 5 [N 6 []],
          N 4 [],
          N 7 [N 2 [], N 1 []]]
ej3 = N 3 [N 5 [N 6 []],
          N 4 [N 1 [N 2 [],N 3 [N 4 []]]],
          N 7 [N 2 [], N 1 []]]

```

```

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]

```

10.2.2. Expresiones

10.2.2.1. Expresiones aritméticas básicas

```

-----
-- Las expresiones aritméticas básicas pueden representarse usando el
-- siguiente tipo de datos
--      data Expr = N Int | S Expr Expr | P Expr Expr
--              deriving Show
-- Por ejemplo, la expresión 2*(3+7) se representa por
--      P (N 2) (S (N 3) (N 7))

```

```
-----
data Expr = N Int | S Expr Expr | P Expr Expr
deriving Show
```

```
-----
-- Ejercicio 1. [14 de marzo de 2011] Definir la función
--   valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión aritmética e. Por
-- ejemplo,
--   valor (P (N 2) (S (N 3) (N 7))) == 20
-----
```

```
valor :: Expr -> Int
valor (N x)    = x
valor (S x y)  = valor x + valor y
valor (P x y)  = valor x * valor y
```

```
-----
-- Ejercicio 2. [10 de septiembre de 2014]. Definir la función
--   dual :: Expr -> Expr
-- tal que (dual e) es la dual de la expresión e. Por ejemplo,
--   dual (P (N 2) (S (N 3) (N 7))) == S (N 2) (P (N 3) (N 7))
-----
```

```
dual :: Expr -> Expr
dual (N x)      = N x
dual (S e1 e2) = P (dual e1) (dual e2)
dual (P e1 e2) = S (dual e1) (dual e2)
```

```
-----
-- Ejercicio 3. [16 de mayo de 2014]. Definir la función
--   aplica :: (Int -> Int) -> Expr -> Expr
-- 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) (S (P (N 3) (N 5)) (P (N 6) (N 7)))
--   S (P (N 5) (N 7)) (P (N 8) (N 9))
--   ghci> aplica (*2) (S (P (N 3) (N 5)) (P (N 6) (N 7)))
--   S (P (N 6) (N 10)) (P (N 12) (N 14))
-----
```

```

aplica :: (Int -> Int) -> Expr -> Expr
aplica f (N x)      = N (f x)
aplica f (S e1 e2) = S (aplica f e1) (aplica f e2)
aplica f (P e1 e2) = P (aplica f e1) (aplica f e2)

```

10.2.2.2. Expresiones aritméticas con una variable

```

-----
-- 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 Expr definido por
--   data Expr = X
--             | N Int
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión "X*(13+X)" se representa por
-- "P X (S (N 13) X)".
-----

```

```

data Expr = X
          | N Int
          | S Expr Expr
          | P Expr Expr

```

```

-----
-- Ejercicio 1. [16 de septiembre de 2011] Definir la función
--   valorE :: Expr -> Int -> Int
-- tal que (valorE e n) es el valor de la expresión e cuando se
-- sustituye su variable por n. Por ejemplo,
--   valorE exp1 2 == 23
-----

```

```

valorE :: Expr -> Int -> Int
valorE X      n = n
valorE (N a)  n = a
valorE (S e1 e2) n = valorE e1 n + valorE e2 n
valorE (P e1 e2) n = valorE e1 n * valorE e2 n

```

```

-- Ejercicio 2. [25 de junio de 2012] Definir la función
--   numVars :: Expr -> Int
-- tal que (numVars e) es el número de variables en la expresión e. Por
-- ejemplo,
--   numVars (N 3)           == 0
--   numVars X               == 1
--   numVars (P X (S (N 13) X)) == 2
-----

```

```

numVars :: Expr -> Int
numVars X      = 1
numVars (N n)  = 0
numVars (S a b) = numVars a + numVars b
numVars (P a b) = numVars a + numVars b

```

10.2.2.3. Expresiones aritméticas con varias variables.

```

-----
-- Las expresiones aritméticas con variables pueden representarse usando
-- el siguiente tipo de datos
--   data Expr = N Int
--             | V Char
--             | S Expr Expr
--             | P Expr Expr
--             deriving Show
-- Por ejemplo, la expresión 2*(a+5) se representa por
--   P (N 2) (S (V 'a') (N 5))
-----

```

```

data Expr = N Int
          | V Char
          | S Expr Expr
          | P Expr Expr
          deriving Show

```

```

-----
-- Ejercicio 1. Definir la función
--   valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- decir, el valor de la expresión donde las variables de x se sustituyen

```



```
-- por los valores según se indican en el entorno e). Por ejemplo,
-- ghci> valor (P (N 2) (S (V 'a') (V 'b')))) [('a',2),('b',5)]
-- 14
```

```
-----
valor :: Expr -> [(Char,Int)] -> Int
valor (N x) e = x
valor (V x) e = head [y | (z,y) <- e, z == x]
valor (S x y) e = valor x e + valor y e
valor (P x y) e = valor x e * valor y e
```

```
-----
-- Ejercicio 2. [21 de marzo de 2013].Definir la función
-- sumas :: Expr -> Int
-- tal que (sumas e) es el número de sumas en la expresión e. Por
-- ejemplo,
-- sumas (P (V 'z') (S (N 3) (V 'x')))) == 1
-- sumas (S (V 'z') (S (N 3) (V 'x')))) == 2
-- sumas (P (V 'z') (P (N 3) (V 'x')))) == 0
```

```
-----
sumas :: Expr -> Int
sumas (V _) = 0
sumas (N _) = 0
sumas (S x y) = 1 + sumas x + sumas y
sumas (P x y) = sumas x + sumas y
```

```
-----
-- Ejercicio 3. [29 de junio de 2012] Definir la función
-- sustitucion :: Expr -> [(Char, Int)] -> Expr
-- 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 (P (V 'z') (S (N 3) (V 'x')))) [('x',7),('z',9)]
-- P (N 9) (S (N 3) (N 7))
-- ghci> sustitucion (P (V 'z') (S (N 3) (V 'y')))) [('x',7),('z',9)]
-- P (N 9) (S (N 3) (V 'y'))
```

```
-----
sustitucion :: Expr -> [(Char, Int)] -> Expr
```

```

sustitucion e [] = e
sustitucion (V c) ((d,n):ps) | c == d = N n
                              | otherwise = sustitucion (V c) ps

sustitucion (N n) _ = N n
sustitucion (S e1 e2) ps = S (sustitucion e1 ps) (sustitucion e2 ps)
sustitucion (P e1 e2) ps = P (sustitucion e1 ps) (sustitucion e2 ps)

```

```

-----
-- Ejercicio 4. [3 de diciembre de 2014] Definir la función
--   reducible :: Expr -> 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 (S (N 3) (N 4))           == True
--   reducible (S (N 3) (V 'x'))        == False
--   reducible (S (N 3) (P (N 4) (N 5))) == True
--   reducible (S (V 'x') (P (N 4) (N 5))) == True
--   reducible (S (N 3) (P (V 'x') (N 5))) == False
--   reducible (N 3)                    == False
--   reducible (V 'x')                   == False
-----

```

```

reducible :: Expr -> Bool
reducible (N _)           = False
reducible (V _)           = False
reducible (S (N _) (N _)) = True
reducible (S a b)         = reducible a || reducible b
reducible (P (N _) (N _)) = True
reducible (P a b)         = reducible a || reducible b

```

10.2.2.4. Expresiones aritméticas generales

```

-----
-- Las expresiones aritméticas se pueden definir usando el siguiente
-- tipo de datos
--   data Expr = N Int
--             | X
--             | S Expr Expr
--             | R Expr Expr
--             | P Expr Expr
-----

```

```
--          | E Expr Int
--          deriving (Eq, Show)
-- Por ejemplo, la expresión
--    3*x - (x+2)^7
-- se puede definir por
--    R (P (N 3) X) (E (S X (N 2)) 7)
```

```
-----
-- Ejercicio 1. [20 de noviembre de 2012]. Definir la función
--    maximo :: Expr -> [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 (E (S (N 10) (P (R (N 1) X) X)) 2) [-3..3]
--    (100,[0,1])
-----
```

```
data Expr = N Int
  | X
  | S Expr Expr
  | R Expr Expr
  | P Expr Expr
  | E Expr Int
  deriving (Eq, Show)
```

```
maximo :: Expr -> [Int] -> (Int,[Int])
maximo e ns = (m,[n | n <- ns, valor e n == m])
  where m = maximum [valor e n | n <- ns]
```

```
valor :: Expr -> Int -> Int
valor (N x) _ = x
valor X     n = n
valor (S e1 e2) n = (valor e1 n) + (valor e2 n)
valor (R e1 e2) n = (valor e1 n) - (valor e2 n)
valor (P e1 e2) n = (valor e1 n) * (valor e2 n)
valor (E e m) n = (valor e n)^m
```

10.2.2.5. Expresiones aritméticas con tipo de operaciones

```

-----
-- Ejercicio 1. [21 de marzo de 2014] Las operaciones de suma, resta y
-- multiplicación se pueden representar mediante el siguiente tipo de
-- datos
--   data Op = S | R | M
-- La expresiones aritméticas con dichas operaciones se pueden
-- representar mediante el siguiente tipo de dato algebraico
--   data Expr = N Int | A Op Expr Expr
-- Por ejemplo, la expresión
--   (7-3)+(2*5)
-- se representa por
--   A S (A R (N 7) (N 3)) (A M (N 2) (N 5))
--
-- Definir la función
--   valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--   valor (A S (A R (N 7) (N 3)) (A M (N 2) (N 5))) == 14
--   valor (A M (A R (N 7) (N 3)) (A S (N 2) (N 5))) == 28
-----

```

```
data Op = S | R | M
```

```
data Expr = N Int | A Op Expr Expr
```

```
valor :: Expr -> Int
```

```
valor (N x) = x
```

```
valor (A o e1 e2) = aplica o (valor e1) (valor e2)
```

```
aplica :: Op -> Int -> Int -> Int
```

```
aplica S x y = x+y
```

```
aplica R x y = x-y
```

```
aplica M x y = x*y
```

10.2.2.6. Expresiones vectoriales

```

-----
-- Ejercicio 1. [18 de junio de 2014]. 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
--
-- Definir la función
--     valor :: ExpV -> (Int,Int)
-- tal que (valor e) es el valor de la expresión vectorial c. Por
-- ejemplo,
--     valor (Vec 1 2)                               == (1,2)
--     valor (Sum (Vec 1 2 ) (Vec 3 4))              == (4,6)
--     valor (Mul 2 (Vec 3 4))                        == (6,8)
--     valor (Mul 2 (Sum (Vec 1 2 ) (Vec 3 4)))      == (8,12)
--     valor (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

```

```

-- =====

```

```

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

```

```

-- 2ª solución

```

```

-- =====

```

```

valor2 :: ExpV -> (Int,Int)
valor2 (Vec a b)   = (a, b)
valor2 (Sum e1 e2) = suma (valor2 e1) (valor2 e2)
valor2 (Mul n e1)  = multiplica n (valor2 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)
```

10.2.3. Recorridos

10.2.3.1. Recorridos

```
-----
-- Ejercicio 1. [28 de marzo de 2012] Representaremos un recorrido
-- como una secuencia de puntos en el espacio de dos dimensiones. Para
-- ello utilizaremos la siguiente definición
--     data Recorrido = Nodo Double Double Recorrido
--                       | Fin
--                       deriving Show
-- De esta forma, el recorrido que parte del punto (0,0) pasa por el
-- punto (1,2) y termina en el (2,4) se representará como
--     rec0 :: Recorrido
--     rec0 = Nodo 0 0 (Nodo 1 2 (Nodo 2 4 Fin))
-- A continuación se muestran otros ejemplos definidos
--     rec1, rec2, rec3, rec4 :: Recorrido
--     rec1 = Nodo 0 0 (Nodo 1 1 Fin)
--     rec2 = Fin
--     rec3 = Nodo 1 (-1) (Nodo 2 3 (Nodo 5 (-2) (Nodo 1 0 Fin)))
--     rec4 = Nodo 0 0
--             (Nodo 0 2
--             (Nodo 2 0
--             (Nodo 0 0
--             (Nodo 2 2
--             (Nodo 2 0
--             (Nodo 0 0 Fin))))))
--
-- Definir la función
--     distanciaRecorrido :: Recorrido -> Double
-- tal que (distanciaRecorrido ps) es la suma de las distancias de todos
-- los segmentos de un recorrido ps. Por ejemplo,
--     distanciaRecorrido rec0      == 4.4721359549995
--     distanciaRecorrido rec1      == 1.4142135623730951
```

```

-- distanciaRecorrido rec2 == 0.0
-----

data Recorrido = Nodo Double Double Recorrido
               | Fin
               deriving Show

rec0, rec1, rec2, rec3, rec4 :: Recorrido
rec0 = Nodo 0 0 (Nodo 1 2 (Nodo 2 4 Fin))
rec1 = Nodo 0 0 (Nodo 1 1 Fin)
rec2 = Fin
rec3 = Nodo 1 (-1) (Nodo 2 3 (Nodo 5 (-2) (Nodo 1 0 Fin)))
rec4 = Nodo 0 0
      (Nodo 0 2
        (Nodo 2 0
          (Nodo 0 0
            (Nodo 2 2
              (Nodo 2 0
                (Nodo 0 0 Fin))))))

distanciaRecorrido :: Recorrido -> Double
distanciaRecorrido Fin = 0
distanciaRecorrido (Nodo _ _ Fin) = 0
distanciaRecorrido (Nodo x y r@(Nodo x' y' n)) =
  distancia (x,y) (x',y') + distanciaRecorrido r

-- (distancia p q) es la distancia del punto p al q. Por ejemplo,
-- distancia (0,0) (3,4) == 5.0
distancia :: (Double,Double) -> (Double,Double) -> Double
distancia (x,y) (x',y') =
  sqrt ((x-x')^2 + (y-y')^2)

```


Relación 11

Programación de dibujos y animaciones con Haskell y Gloss

En esta relación se propone programar dibujos, fractales, animaciones y simulaciones con Haskell y Gloss o con CodeWorld.

Relación 12

Combinatoria

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es estudiar la generación y el número de  
-- las principales operaciones de la combinatoria. En concreto, se  
-- estudia  
-- * Permutaciones.  
-- * Combinaciones sin repetición..  
-- * Combinaciones con repetición  
-- * Variaciones sin repetición.  
-- * Variaciones con repetición.  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
import Test.QuickCheck  
import Data.List (genericLength)  
  
-- -----  
-- § Subconjuntos  
-- -----  
  
-- -----  
-- Ejercicio 1. Definir, por recursión, la función  
-- subconjunto :: Eq a => [a] -> [a] -> Bool
```

```
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
--   subconjunto [1,3,2,3] [1,2,3] == True
--   subconjunto [1,3,4,3] [1,2,3] == False
-----
```

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

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

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

```
-----
-- Ejercicio 3. Comprobar con QuickCheck que las funciones subconjunto
-- y subconjunto' son equivalentes.
-----
```

```
-- La propiedad es
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
  subconjunto xs ys == subconjunto' xs ys
```

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

```
-- Ejercicio 4. Definir la función
--   igualConjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (igualConjunto xs ys) se verifica si las listas xs e ys,
```

```
-- vistas como conjuntos, son iguales. Por ejemplo,
--   igualConjunto [1..10] [10,9..1] == True
--   igualConjunto [1..10] [11,10..1] == False
```

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

```
-----
-- Ejercicio 5. Definir la función
--   subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de los subconjuntos de la lista
-- xs. Por ejemplo,
--   ghci> subconjuntos [2,3,4]
--   [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
--   ghci> subconjuntos [1,2,3,4]
--   [[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
--     [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
```

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

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

```
-----
-- § Permutaciones
-----
```

```
-----
-- Ejercicio 6. Definir la función
--   intercala :: a -> [a] -> [[a]]
-- tal que (intercala x ys) es la lista de las listas obtenidas
-- intercalando x entre los elementos de ys. Por ejemplo,
```

```
-- intercala 1 [2,3] == [[1,2,3],[2,1,3],[2,3,1]]
```

```
-- Una definición recursiva es
```

```
intercala1 :: a -> [a] -> [[a]]
```

```
intercala1 x [] = [[x]]
```

```
intercala1 x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala1 x ys]
```

```
-- Otra definición, más eficiente, es
```

```
intercala :: a -> [a] -> [[a]]
```

```
intercala y xs =
```

```
  [take n xs ++ (y : drop n xs) | n <- [0..length xs]]
```

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

```
-- permutaciones :: [a] -> [[a]]
```

```
-- tal que (permutaciones xs) es la lista de las permutaciones de la  
-- lista xs. Por ejemplo,
```

```
-- permutaciones "bc" == ["bc","cb"]
```

```
-- permutaciones "abc" == ["abc","bac","bca","acb","cab","cba"]
```

```
permutaciones :: [a] -> [[a]]
```

```
permutaciones [] = [[]]
```

```
permutaciones (x:xs) =
```

```
  concat [intercala x ys | ys <- permutaciones xs]
```

```
-- Ejercicio 8. Definir la función
```

```
-- permutacionesN :: Integer -> [[Integer]]
```

```
-- tal que (permutacionesN n) es la lista de las permutaciones de los n  
-- primeros números. Por ejemplo,
```

```
-- ghci> permutacionesN 3
```

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

```
permutacionesN :: Integer -> [[Integer]]
```

```
permutacionesN n = permutaciones [1..n]
```

```
-- Ejercicio 9. Definir, usando permutacionesN, la función
-- numeroPermutacionesN :: Integer -> Integer
-- tal que (numeroPermutacionesN n) es el número de permutaciones de un
-- conjunto con n elementos. Por ejemplo,
-- numeroPermutacionesN 3 == 6
-- numeroPermutacionesN 4 == 24
-----
```

```
numeroPermutacionesN :: Integer -> Integer
numeroPermutacionesN = genericLength . permutacionesN
-----
```

```
-- Ejercicio 10. Definir la función
-- fact :: Integer -> Integer
-- tal que (fact n) es el factorial de n. Por ejemplo,
-- fact 3 == 6
-----
```

```
fact :: Integer -> Integer
fact n = product [1..n]
-----
```

```
-- Ejercicio 11. Definir, usando fact, la función
-- numeroPermutacionesN' :: Integer -> Integer
-- tal que (numeroPermutacionesN' n) es el número de permutaciones de un
-- conjunto con n elementos. Por ejemplo,
-- numeroPermutacionesN' 3 == 6
-- numeroPermutacionesN' 4 == 24
-----
```

```
numeroPermutacionesN' :: Integer -> Integer
numeroPermutacionesN' = fact
-----
```

```
-- Ejercicio 12. Definir la función
-- prop_numeroPermutacionesN :: Integer -> Bool
-- tal que (prop_numeroPermutacionesN n) se verifica si las funciones
-- numeroPermutacionesN y numeroPermutacionesN' son equivalentes para
-- los n primeros números. Por ejemplo,
-- prop_numeroPermutacionesN 5 == True
```

```

-----
prop_numeroPermutacionesN :: Integer -> Bool
prop_numeroPermutacionesN n =
  and [numeroPermutacionesN x == numeroPermutacionesN' x | x <- [1..n]]

```

```

-----
-- § Combinaciones
-----

```

```

-----
-- Ejercicio 13. Definir la función
--   combinaciones :: Integer -> [a] -> [[a]]
-- tal que (combinaciones k xs) es la lista de las combinaciones de
-- orden k de los elementos de la lista xs. Por ejemplo,
--   ghci> combinaciones 2 "bcde"
--   ["bc","bd","be","cd","ce","de"]
--   ghci> combinaciones 3 "bcde"
--   ["bcd","bce","bde","cde"]
--   ghci> combinaciones 3 "abcde"
--   ["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
-----

```

```

-- 1ª definición

```

```

combinaciones_1 :: Integer -> [a] -> [[a]]
combinaciones_1 n xs =
  [ys | ys <- subconjuntos xs, genericLength ys == n]

```

```

-- 2ª definición

```

```

combinaciones_2 :: Integer -> [a] -> [[a]]
combinaciones_2 0 _ = [[]]
combinaciones_2 _ [] = []
combinaciones_2 k (x:xs) =
  [x:ys | ys <- combinaciones_2 (k-1) xs] ++ combinaciones_2 k xs

```

```

-- La anterior definición se puede escribir usando map:

```

```

combinaciones_3 :: Integer -> [a] -> [[a]]
combinaciones_3 0 _ = [[]]
combinaciones_3 _ [] = []
combinaciones_3 (k+1) (x:xs) =

```



```

map (x:) (combinaciones_3 k xs) ++ combinaciones_3 (k+1) xs

-- Nota. La segunda definición es más eficiente como se comprueba en la
-- siguiente sesión
-- ghci> :set +s
-- ghci> length (combinaciones_1 2 [1..15])
-- 105
-- (0.19 secs, 6373848 bytes)
-- ghci> length (combinaciones_2 2 [1..15])
-- 105
-- (0.01 secs, 525360 bytes)
-- ghci> length (combinaciones_3 2 [1..15])
-- 105
-- (0.02 secs, 528808 bytes)

-- En lo que sigue, usaremos combinaciones como combinaciones_2
combinaciones :: Integer -> [a] -> [[a]]
combinaciones = combinaciones_2

-----
-- Ejercicio 14. Definir la función
-- combinacionesN :: Integer -> Integer -> [[Int]]
-- tal que (combinacionesN n k) es la lista de las combinaciones de
-- orden k de los n primeros números. Por ejemplo,
-- ghci> combinacionesN 4 2
-- [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
-- ghci> combinacionesN 4 3
-- [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
-----

combinacionesN :: Integer -> Integer -> [[Integer]]
combinacionesN n k = combinaciones k [1..n]

-----
-- Ejercicio 15. Definir, usando combinacionesN, la función
-- numeroCombinaciones :: Integer -> Integer -> Integer
-- tal que (numeroCombinaciones n k) es el número de combinaciones de
-- orden k de un conjunto con n elementos. Por ejemplo,
-- numeroCombinaciones 4 2 == 6
-- numeroCombinaciones 4 3 == 4

```

```

-----
numeroCombinaciones :: Integer -> Integer -> Integer
numeroCombinaciones n k = genericLength (combinacionesN n k)

-- Puede definirse por composición
numeroCombinaciones_2 :: Integer -> Integer -> Integer
numeroCombinaciones_2 = (genericLength .) . combinacionesN

-- Para facilitar la escritura de las definiciones por composición de
-- funciones con dos argumentos, se puede definir
(.:) :: (c -> d) -> (a -> b -> c) -> a -> b -> d
(.:) = (.) . (.)

-- con lo que la definición anterior se simplifica a
numeroCombinaciones_3 :: Integer -> Integer -> Integer
numeroCombinaciones_3 = genericLength .: combinacionesN

-----
-- Ejercicio 16. Definir la función
--   comb :: Integer -> Integer -> Integer
-- tal que (comb n k) es el número combinatorio n sobre k; es decir,
--   (comb n k) = n! / (k!(n-k)!).
-- Por ejemplo,
--   comb 4 2 == 6
--   comb 4 3 == 4
-----

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

```

```

-----
-- Ejercicio 17. Definir, usando comb, la función
--   numeroCombinaciones' :: Integer -> Integer -> Integer
-- tal que (numeroCombinaciones' n k) es el número de combinaciones de
-- orden k de un conjunto con n elementos. Por ejemplo,
--   numeroCombinaciones' 4 2 == 6
--   numeroCombinaciones' 4 3 == 4
-----

```

```

numeroCombinaciones' :: Integer -> Integer -> Integer
numeroCombinaciones' = comb

-----

-- Ejercicio 18. Definir la función
--   prop_numeroCombinaciones :: Integer -> Bool
-- tal que (prop_numeroCombinaciones n) se verifica si las funciones
-- numeroCombinaciones y numeroCombinaciones' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--   prop_numeroCombinaciones 5 == True
-----

prop_numeroCombinaciones :: Integer -> Bool
prop_numeroCombinaciones n =
  and [numeroCombinaciones n k == numeroCombinaciones' n k | k <- [1..n]]

-----

-- § Combinaciones con repetición
-----

-----

-- Ejercicio 19. Definir la función
--   combinacionesR :: Integer -> [a] -> [[a]]
-- tal que (combinacionesR k xs) es la lista de las combinaciones orden
-- k de los elementos de xs con repeticiones. Por ejemplo,
--   ghci> combinacionesR 2 "abc"
--   ["aa","ab","ac","bb","bc","cc"]
--   ghci> combinacionesR 3 "bc"
--   ["bbb","bbc","bcc","ccc"]
--   ghci> combinacionesR 3 "abc"
--   ["aaa","aab","aac","abb","abc","acc","bbb","bbc","bcc","ccc"]
-----

combinacionesR :: Integer -> [a] -> [[a]]
combinacionesR _ [] = []
combinacionesR 0 _ = [[]]
combinacionesR k (x:xs) =
  [x:ys | ys <- combinacionesR (k-1) (x:xs)] ++ combinacionesR k xs
-----

```

```
-- Ejercicio 20. Definir la función
-- combinacionesRN :: Integer -> Integer -> [[Integer]]
-- tal que (combinacionesRN n k) es la lista de las combinaciones orden
-- k de los primeros n números naturales. Por ejemplo,
-- ghci> combinacionesRN 3 2
-- [[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]
-- ghci> combinacionesRN 2 3
-- [[1,1,1],[1,1,2],[1,2,2],[2,2,2]]
```

```
-----
combinacionesRN :: Integer -> Integer -> [[Integer]]
combinacionesRN n k = combinacionesR k [1..n]
```

```
-----
-- Ejercicio 21. Definir, usando combinacionesRN, la función
-- numeroCombinacionesR :: Integer -> Integer -> Integer
-- tal que (numeroCombinacionesR n k) es el número de combinaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
-- numeroCombinacionesR 3 2 == 6
-- numeroCombinacionesR 2 3 == 4
```

```
-----
numeroCombinacionesR :: Integer -> Integer -> Integer
numeroCombinacionesR n k = genericLength (combinacionesRN n k)
```

```
-----
-- Ejercicio 22. Definir, usando comb, la función
-- numeroCombinacionesR' :: Integer -> Integer -> Integer
-- tal que (numeroCombinacionesR' n k) es el número de combinaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
-- numeroCombinacionesR' 3 2 == 6
-- numeroCombinacionesR' 2 3 == 4
```

```
-----
numeroCombinacionesR' :: Integer -> Integer -> Integer
numeroCombinacionesR' n k = comb (n+k-1) k
```

```
-----
-- Ejercicio 23. Definir la función
-- prop_numeroCombinacionesR :: Integer -> Bool
```

```
-- tal que (prop_numeroCombinacionesR n) se verifica si las funciones
-- numeroCombinacionesR y numeroCombinacionesR' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--   prop_numeroCombinacionesR 5 == True
```

```
-----
prop_numeroCombinacionesR :: Integer -> Bool
prop_numeroCombinacionesR n =
  and [numeroCombinacionesR n k == numeroCombinacionesR' n k |
       k <- [1..n]]
```

```
-----
-- § Variaciones
-----
```

```
-----
-- Ejercicio 24. Definir la función
--   variaciones :: Integer -> [a] -> [[a]]
-- tal que (variaciones n xs) es la lista de las variaciones n-arias
-- de la lista xs. Por ejemplo,
--   variaciones 2 "abc" == ["ab","ba","ac","ca","bc","cb"]
```

```
-----
variaciones :: Integer -> [a] -> [[a]]
variaciones k xs =
  concat (map permutaciones (combinaciones k xs))
```

```
-----
-- Ejercicio 25. Definir la función
--   variacionesN :: Integer -> Integer -> [[Integer]]
-- tal que (variacionesN n k) es la lista de las variaciones de orden k
-- de los n primeros números. Por ejemplo,
--   variacionesN 3 2 == [[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]]
```

```
-----
variacionesN :: Integer -> Integer -> [[Integer]]
variacionesN n k = variaciones k [1..n]
```

```
-----
-- Ejercicio 26. Definir, usando variacionesN, la función
```

```
-- numeroVariaciones :: Integer -> Integer -> Integer
-- tal que (numeroVariaciones n k) es el número de variaciones de orden
-- k de un conjunto con n elementos. Por ejemplo,
-- numeroVariaciones 4 2 == 12
-- numeroVariaciones 4 3 == 24
-----
```

```
numeroVariaciones :: Integer -> Integer -> Integer
numeroVariaciones n k = genericLength (variacionesN n k)
```

```
-----
-- Ejercicio 27. Definir, usando product, la función
-- numeroVariaciones' :: Integer -> Integer -> Integer
-- tal que (numeroVariaciones' n k) es el número de variaciones de orden
-- k de un conjunto con n elementos. Por ejemplo,
-- numeroVariaciones' 4 2 == 12
-- numeroVariaciones' 4 3 == 24
-----
```

```
numeroVariaciones' :: Integer -> Integer -> Integer
numeroVariaciones' n k = product [(n-k+1)..n]
```

```
-----
-- Ejercicio 28. Definir la función
-- prop_numeroVariaciones :: Integer -> Bool
-- tal que (prop_numeroVariaciones n) se verifica si las funciones
-- numeroVariaciones y numeroVariaciones' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
-- prop_numeroVariaciones 5 == True
-----
```

```
prop_numeroVariaciones :: Integer -> Bool
prop_numeroVariaciones n =
  and [numeroVariaciones n k == numeroVariaciones' n k | k <- [1..n]]
```

```
-----
-- § Variaciones con repetición
-----
```

```

-- Ejercicio 28. Definir la función
--   variacionesR :: Integer -> [a] -> [[a]]
-- tal que (variacionesR k xs) es la lista de las variaciones de orden
-- k de los elementos de xs con repeticiones. Por ejemplo,
--   ghci> variacionesR 1 "ab"
--   ["a","b"]
--   ghci> variacionesR 2 "ab"
--   ["aa","ab","ba","bb"]
--   ghci> variacionesR 3 "ab"
--   ["aaa","aab","aba","abb","baa","bab","bba","bbb"]

```

```

variacionesR :: Integer -> [a] -> [[a]]
variacionesR _ [] = [[]]
variacionesR 0 _ = [[]]
variacionesR k xs =
  [z:ys | z <- xs, ys <- variacionesR (k-1) xs]

```

```

-- Ejercicio 30. Definir la función
--   variacionesRN :: Integer -> Integer -> [[Integer]]
-- tal que (variacionesRN n k) es la lista de las variaciones orden
-- k de los primeros n números naturales. Por ejemplo,
--   ghci> variacionesRN 3 2
--   [[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
--   ghci> variacionesRN 2 3
--   [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]

```

```

variacionesRN :: Integer -> Integer -> [[Integer]]
variacionesRN n k = variacionesR k [1..n]

```

```

-- Ejercicio 31. Definir, usando variacionesR, la función
--   numeroVariacionesR :: Integer -> Integer -> Integer
-- tal que (numeroVariacionesR n k) es el número de variaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
--   numeroVariacionesR 3 2 == 9
--   numeroVariacionesR 2 3 == 8

```

```

numeroVariacionesR :: Integer -> Integer -> Integer
numeroVariacionesR n k = genericLength (variacionesRN n k)

```

```

-----
-- Ejercicio 32. Definir, usando (^), la función
--   numeroVariacionesR' :: Integer -> Integer -> Integer
-- tal que (numeroVariacionesR' n k) es el número de variaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
--   numeroVariacionesR' 3 2 == 9
--   numeroVariacionesR' 2 3 == 8
-----

```

```

numeroVariacionesR' :: Integer -> Integer -> Integer
numeroVariacionesR' n k = n^k

```

```

-----
-- Ejercicio 33. Definir la función
--   prop_numeroVariacionesR :: Integer -> Bool
-- tal que (prop_numeroVariacionesR n) se verifica si las funciones
-- numeroVariacionesR y numeroVariacionesR' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--   prop_numeroVariacionesR 5 == True
-----

```

```

prop_numeroVariacionesR :: Integer -> Bool
prop_numeroVariacionesR n =
  and [numeroVariacionesR n k == numeroVariacionesR' n k |
       k <- [1..n]]

```


Relación 13

El TAD de las pilas

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir funciones sobre  
-- el TAD de las pilas, utilizando las implementaciones estudiadas en el  
-- tema 14 cuyas transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-14.pdf  
--  
-- Para realizar los ejercicios hay que descargar, en el mismo  
-- directorio que el enunciado, el código de los TAD  
-- PilaConTipoDeDatoAlgebraico.hs que está en http://bit.ly/1Aic5Ba  
-- PilaConListas.hs que está en http://bit.ly/1AicnYE  
--  
-- Hay que hacer los ejercicios con la primera implementación  
-- (PilaConTipoDeDatoAlgebraico) y comprobar que las definiciones  
-- también son válidas con la segunda implementación (PilaConListas).  
  
-----  
-- Importación de librerías --  
-----  
  
-- Hay que elegir una implementación del TAD pilas.  
import I1M.Pila  
-- import PilaConTipoDeDatoAlgebraico  
-- import PilaConListas
```

```
import Data.List
import Test.QuickCheck
```

```
-----
-- Ejemplos
-----
```

```
-- A lo largo de esta relación de ejercicios usaremos los siguientes
-- ejemplos de pila
```

```
p1, p2, p3, p4, p5 :: Pila Int
p1 = foldr apila vacia [1..20]
p2 = foldr apila vacia [2,5..18]
p3 = foldr apila vacia [3..10]
p4 = foldr apila vacia [4,-1,7,3,8,10,0,3,3,4]
p5 = foldr apila vacia [1..5]
```

```
-----
-- Ejercicio 1: Definir la función
```

```
--   filtraPila :: (a -> Bool) -> Pila a -> Pila a
-- tal que (filtraPila p q) es la pila obtenida con los elementos de
-- pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
--   ghci> p1
--   1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|-
--   ghci> filtraPila even p1
--   2|4|6|8|10|12|14|16|18|20|-
-----
```

```
filtraPila :: (a -> Bool) -> Pila a -> Pila a
```

```
filtraPila p q
  | esVacia q = vacia
  | p cq      = apila cq (filtraPila p dq)
  | otherwise = filtraPila p dq
  where cq = cima q
        dq = desapila q
```

```
-----
-- Ejercicio 2: Definir la función
```

```
--   mapPila :: (a -> a) -> Pila a -> Pila a
-- tal que (mapPila f p) es la pila formada con las imágenes por f de
-- los elementos de pila p, en el mismo orden. Por ejemplo,
```

```
-- ghci> mapPila (+7) p1
-- 8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|-
```

```
-----
mapPila :: (a -> a) -> Pila a -> Pila a
mapPila f p
  | esVacia p = p
  | otherwise = apila (f cp) (mapPila f dp)
  where cp = cima p
        dp = desapila p
```

```
-----
-- Ejercicio 3: Definir la función
-- pertenecePila :: Eq a => a -> Pila a -> Bool
-- tal que (pertenecePila y p) se verifica si y es un elemento de la
-- pila p. Por ejemplo,
-- pertenecePila 7 p1 == True
-- pertenecePila 70 p1 == False
```

```
-----
pertenecePila :: Eq a => a -> Pila a -> Bool
pertenecePila x p
  | esVacia p = False
  | otherwise = x == cp || pertenecePila x dp
  where cp = cima p
        dp = desapila p
```

```
-----
-- Ejercicio 4: Definir la función
-- contenidaPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (contenidaPila p1 p2) se verifica si todos los elementos de
-- de la pila p1 son elementos de la pila p2. Por ejemplo,
-- contenidaPila p2 p1 == True
-- contenidaPila p1 p2 == False
```

```
-----
contenidaPila :: Eq a => Pila a -> Pila a -> Bool
contenidaPila p1 p2
  | esVacia p1 = True
  | otherwise = pertenecePila cp1 p2 && contenidaPila dp1 p2
```

```

where cp1 = cima p1
        dp1 = desapila p1

```

```

-----
-- Ejercicio 4: Definir la función
--   prefijoPila :: Eq a => Pila a -> Pila a -> Bool
--   tal que (prefijoPila p1 p2) se verifica si la pila p1 es justamente
--   un prefijo de la pila p2. Por ejemplo,
--     prefijoPila p3 p2 == False
--     prefijoPila p5 p1 == True
-----

```

```

prefijoPila :: Eq a => Pila a -> Pila a -> Bool
prefijoPila p1 p2
  | esVacía p1 = True
  | esVacía p2 = False
  | otherwise  = cp1 == cp2 && prefijoPila dp1 dp2
where cp1 = cima p1
        dp1 = desapila p1
        cp2 = cima p2
        dp2 = desapila p2

```

```

-----
-- Ejercicio 5. Definir la función
--   subPila :: Eq a => Pila a -> Pila a -> Bool
--   tal que (subPila p1 p2) se verifica si p1 es una subpila de p2. Por
--   ejemplo,
--     subPila p2 p1 == False
--     subPila p3 p1 == True
-----

```

```

subPila :: Eq a => Pila a -> Pila a -> Bool
subPila p1 p2
  | esVacía p1 = True
  | esVacía p2 = False
  | cp1 == cp2 = prefijoPila dp1 dp2 || subPila p1 dp2
  | otherwise  = subPila p1 dp2
where cp1 = cima p1
        dp1 = desapila p1
        cp2 = cima p2

```

```
dp2 = desapila p2
```

```
-----  
-- Ejercicio 6. Definir la función  
--   ordenadaPila :: Ord a => Pila a -> Bool  
-- tal que (ordenadaPila p) se verifica si los elementos de la pila p  
-- están ordenados en orden creciente. Por ejemplo,  
--   ordenadaPila p1 == True  
--   ordenadaPila p4 == False  
-----
```

```
ordenadaPila :: Ord a => Pila a -> Bool  
ordenadaPila p  
  | esVacia p = True  
  | esVacia dp = True  
  | otherwise = cp <= cdp && ordenadaPila dp  
where cp = cima p  
      dp = desapila p  
      cdp = cima dp
```

```
-----  
-- Ejercicio 7.1. Definir la función  
--   lista2Pila :: [a] -> Pila a  
-- tal que (lista2Pila xs) es la pila formada por los elementos de  
-- xs. Por ejemplo,  
--   lista2Pila [1..6] == 1|2|3|4|5|6|  
-----
```

```
lista2Pila :: [a] -> Pila a  
lista2Pila = foldr apila vacia
```

```
-----  
-- Ejercicio 7.2. Definir la función  
--   pila2Lista :: Pila a -> [a]  
-- tal que (pila2Lista p) es la lista formada por los elementos de la  
-- lista p. Por ejemplo,  
--   pila2Lista p2 == [2,5,8,11,14,17]  
-----
```

```
pila2Lista :: Pila a -> [a]
```

```
pila2Lista p
  | esVacia p = []
  | otherwise = cp : pila2Lista dp
  where cp = cima p
        dp = desapila p
```

```
-----
-- Ejercicio 7.3. Comprobar con QuickCheck que la función pila2Lista es
-- la inversa de lista2Pila, y recíprocamente.
-----
```

```
prop_pila2Lista p =
  lista2Pila (pila2Lista p) == p
```

```
-- ghci> quickCheck prop_pila2Lista
-- +++ OK, passed 100 tests.
```

```
prop_lista2Pila xs =
  pila2Lista (lista2Pila xs) == xs
```

```
-- ghci> quickCheck prop_lista2Pila
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 8.1. Definir la función
--   ordenaInserPila :: Ord a => Pila a -> Pila a
-- tal que (ordenaInserPila p) es la pila obtenida ordenando por
-- inserción los los elementos de la pila p. Por ejemplo,
--   ghci> ordenaInserPila p4
--   -1|0|3|3|3|4|4|7|8|10|-
-----
```

```
ordenaInserPila :: Ord a => Pila a -> Pila a
ordenaInserPila p
  | esVacia p = p
  | otherwise = insertaPila cp (ordenaInserPila dp)
  where cp = cima p
        dp = desapila p
```

```
insertaPila :: Ord a => a -> Pila a -> Pila a
```

```

insertaPila x p
  | esVacia p = apila x p
  | x < cp    = apila x p
  | otherwise = apila cp (insertaPila x dp)
where cp = cima p
      dp = desapila p

```

```

-----
-- Ejercicio 8.2. Comprobar con QuickCheck que la pila
--   (ordenaInserPila p)
--   está ordenada correctamente.
-----

```

```

prop_ordenaInserPila p =
  pila2Lista (ordenaInserPila p) == sort (pila2Lista p)

```

```

-- ghci> quickCheck prop_ordenaInserPila
-- +++ OK, passed 100 tests.

```

```

-----
-- Ejercicio 9.1. Definir la función
--   nubPila :: Eq a => Pila a -> Pila a
--   tal que (nubPila p) es la pila con los elementos de p sin repeticiones.
--   Por ejemplo,
--   ghci> p4
--   4|-1|7|3|8|10|0|3|3|4|-
--   ghci> nubPila p4
--   -1|7|8|10|0|3|4|-
-----

```

```

nubPila :: (Eq a) => Pila a -> Pila a
nubPila p
  | esVacia p          = vacia
  | pertenecePila cp dp = nubPila dp
  | otherwise          = apila cp (nubPila dp)
where cp = cima p
      dp = desapila p

```

```

-----
-- Ejercicio 9.2. Definir la propiedad siguiente: "la composición de

```

```
-- las funciones nub y pila2Lista coincide con la composición de las
-- funciones pila2Lista y nubPila", y comprobarla con quickCheck.
-- En caso de ser falsa, redefinir la función nubPila para que se
-- verifique la propiedad.
```

```
-----
-- La propiedad es
prop_nubPila p =
  nub (pila2Lista p) == pila2Lista (nubPila p)

-- La comprobación es
-- ghci> quickCheck prop_nubPila
-- *** Failed! Falsifiable (after 8 tests):
-- -7|-2|0|-5|-7|-
-- ghci> let p = foldr apila vacia [-7,-2,0,-5,-7]
-- ghci> p
-- -7|-2|0|-5|-7|-
-- ghci> pila2Lista p
-- [-7,-2,0,-5,-7]
-- ghci> nub (pila2Lista p)
-- [-7,-2,0,-5]
-- ghci> nubPila p
-- -2|0|-5|-7|-
-- ghci> pila2Lista (nubPila p)
-- [-2,0,-5,-7]
```

```
-- Falla porque nub quita el último de los elementos repetidos de la
-- lista, mientras que nubPila quita el primero de ellos.
```

```
-- La redefinimos
nubPila' :: Eq a => Pila a -> Pila a
nubPila' p
  | esVacia p           = p
  | pertenecePila cp dp = apila cp (nubPila' (eliminaPila cp dp))
  | otherwise           = apila cp (nubPila' dp)
  where cp = cima p
        dp = desapila p
```

```
eliminaPila :: Eq a => a -> Pila a -> Pila a
eliminaPila x p
```



```

    | esVacia p = p
    | x == cp   = eliminaPila x dp
    | otherwise = apila cp (eliminaPila x dp)
  where cp = cima p
        dp = desapila p

-- La propiedad es
prop_nubPila' p =
  nub (pila2Lista p) == pila2Lista (nubPila' p)

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

-----

-- Ejercicio 10: Definir la función
--   maxPila :: Ord a => Pila a -> a
-- tal que (maxPila p) sea el mayor de los elementos de la pila p. Por
-- ejemplo,
-- ghci> p4
-- 4|-1|7|3|8|10|0|3|3|4|-
-- ghci> maxPila p4
-- 10
-----

maxPila :: Ord a => Pila a -> a
maxPila p
  | esVacia p = error "pila vacia"
  | esVacia dp = cp
  | otherwise = max cp (maxPila dp)
  where cp = cima p
        dp = desapila p

-----

-- Generador de pilas --
-----

-- genPila es un generador de pilas. Por ejemplo,
-- ghci> sample genPila
-- -

```

```

-- 0|0|-
-- -
-- -6|4|-3|3|0|-
-- -
-- 9|5|-1|-3|0|-8|-5|-7|2|-
-- -3|-10|-3|-12|11|6|1|-2|0|-12|-6|-
-- 2|-14|-5|2|-
-- 5|9|-
-- -1|-14|5|-
-- 6|13|0|17|-12|-7|-8|-19|-14|-5|10|14|3|-18|2|-14|-11|-6|-
genPila :: (Num a, Arbitrary a) => Gen (Pila a)
genPila = do xs <- listOf arbitrary
            return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
    arbitrary = genPila

```

Relación 14

El TAD de las colas

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir funciones sobre  
-- el TAD de las colas, utilizando las implementaciones estudiadas en el  
-- tema 15 transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-15.pdf  
--  
-- Para realizar los ejercicios hay que descargar, en el mismo  
-- directorio que el enunciado, el código de los TAD  
-- ColasConListas.hs que está en http://bit.ly/1AieXy6  
-- ColasConDosListas.hs que está en http://bit.ly/1AieZ8X  
--  
-- Hay que hacer los ejercicios con la primera implementación  
-- (ColasConListas) y comprobar que las definiciones también son válidas  
-- con la segunda implementación (ColasConDosListas).  
  
-----  
-- Importación de librerías --  
-----  
  
-- Hay que elegir una implementación del TAD colas:  
import ColaConListas  
-- import ColaConDosListas  
  
import Data.List
```

```
import Test.QuickCheck
```

```
-- -----
-- Nota. A lo largo de la relación de ejercicios usaremos los siguientes
-- ejemplos de colas:
```

```
c1, c2, c3, c4, c5, c6 :: Cola Int
c1 = foldr inserta vacia [1..20]
c2 = foldr inserta vacia [2,5..18]
c3 = foldr inserta vacia [3..10]
c4 = foldr inserta vacia [4,-1,7,3,8,10,0,3,3,4]
c5 = foldr inserta vacia [15..20]
c6 = foldr inserta vacia (reverse [1..20])
-- -----
```

```
-- -----
-- Ejercicio 1: Definir la función
--   ultimoCola :: Cola a -> a
-- tal que (ultimoCola c) es el último elemento de la cola c. Por
-- ejemplo:
--   ultimoCola c4 == 4
--   ultimoCola c5 == 15
-- -----
```

```
ultimoCola :: Cola a -> a
ultimoCola c
  | esVacia c = error "cola vacia"
  | esVacia rc = pc
  | otherwise = ultimoCola rc
  where pc = primero c
        rc = resto c
-- -----
```

```
-- -----
-- Ejercicio 2: Definir la función
--   longitudCola :: Cola a -> Int
-- tal que (longitudCola c) es el número de elementos de la cola c. Por
-- ejemplo,
--   longitudCola c2 == 6
-- -----
```

```
longitudCola :: Cola a -> Int
```

```
longitudCola c
  | esVacia c = 0
  | otherwise = 1 + longitudCola rc
  where rc = resto c

-----

-- Ejercicio 3: Definir la función
--   todosVerifican :: (a -> Bool) -> Cola a -> Bool
-- tal que (todosVerifican p c) se verifica si todos los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--   todosVerifican (>0) c1 == True
--   todosVerifican (>0) c4 == False
-----

todosVerifican :: (a -> Bool) -> Cola a -> Bool
todosVerifican p c
  | esVacia c = True
  | otherwise = p pc && todosVerifican p rc
  where pc = primero c
        rc = resto c

-----

-- Ejercicio 4: Definir la función
--   algunoVerifica :: (a -> Bool) -> Cola a -> Bool
-- tal que (algunoVerifica p c) se verifica si algún elemento de la cola
-- c cumple la propiedad p. Por ejemplo,
--   algunoVerifica (<0) c1 == False
--   algunoVerifica (<0) c4 == True
-----

algunoVerifica :: (a -> Bool) -> Cola a -> Bool
algunoVerifica p c
  | esVacia c = False
  | otherwise = p pc || algunoVerifica p rc
  where pc = primero c
        rc = resto c

-----

-- Ejercicio 5: Definir la función
--   ponAlaCola :: Cola a -> Cola a -> Cola a
```

```
-- tal que (ponAlaCola c1 c2) es la cola que resulta de poner los
-- elementos de c2 a la cola de c1. Por ejemplo,
--   ponAlaCola c2 c3 == C [17,14,11,8,5,2,10,9,8,7,6,5,4,3]
```

```
-----
ponAlaCola :: Cola a -> Cola a -> Cola a
ponAlaCola c1 c2
  | esVacia c2 = c1
  | otherwise = ponAlaCola (inserta pc2 c1) rq2
  where pc2 = primero c2
        rq2 = resto c2
```

```
-----
-- Ejercicio 6: Definir la función
--   mezclaColas :: Cola a -> Cola a -> Cola a
-- tal que (mezclaColas c1 c2) es la cola formada por los elementos de
-- c1 y c2 colocados en una cola, de forma alternativa, empezando por
-- los elementos de c1. Por ejemplo,
--   mezclaColas c2 c4 == C [17,4,14,3,11,3,8,0,5,10,2,8,3,7,-1,4]
```

```
-----
mezclaColas :: Cola a -> Cola a -> Cola a
mezclaColas c1 c2 = aux c1 c2 vacia
  where aux c1 c2 c
        | esVacia c1 = ponAlaCola c c2
        | esVacia c2 = ponAlaCola c c1
        | otherwise = aux rc1 rc2 (inserta pc2 (inserta pc1 c))
        where pc1 = primero c1
              rc1 = resto c1
              pc2 = primero c2
              rc2 = resto c2
```

```
-----
-- Ejercicio 7: Definir la función
--   agrupaColas :: [Cola a] -> Cola a
-- tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
-- las colas de la lista como sigue: mezcla c1 con c2, el resultado con
-- c3, el resultado con c4, y así sucesivamente. Por ejemplo,
--   ghci> agrupaColas [c3,c3,c4]
--   C [10,4,10,3,9,3,9,0,8,10,8,8,7,3,7,7,6,-1,6,4,5,5,4,4,3,3]
```

```

-----
agrupaColas :: [Cola a] -> Cola a
agrupaColas []           = vacia
agrupaColas [c]         = c
agrupaColas (c1:c2:colas) = agrupaColas (mezclaColas c1 c2 : colas)

```

```

-----
-- Ejercicio 8: Definir la función
--   perteneceCola :: Eq a => a -> Cola a -> Bool
-- tal que (perteneceCola x c) se verifica si x es un elemento de la
-- cola c. Por ejemplo,
--   perteneceCola 7 c1 == True
--   perteneceCola 70 c1 == False
-----

```

```

perteneceCola :: Eq a => a -> Cola a -> Bool
perteneceCola x c
  | esVacia c = False
  | otherwise = pc == x || perteneceCola x rc
  where pc = primero c
        rc = resto c
-----

```

```

-----
-- Ejercicio 9: Definir la función
--   contenidaCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (contenidaCola c1 c2) se verifica si todos los elementos de
-- c1 son elementos de c2. Por ejemplo,
--   contenidaCola c2 c1 == True
--   contenidaCola c1 c2 == False
-----

```

```

contenidaCola :: Eq a => Cola a -> Cola a -> Bool
contenidaCola c1 c2
  | esVacia c1 = True
  | esVacia c2 = esVacia c1
  | otherwise = perteneceCola pc1 c2 && contenidaCola rc1 c2
  where pc1 = primero c1
        rc1 = resto c1

```

```

-----
-- Ejercicio 10: Definir la función
--   prefijoCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (prefijoCola c1 c2) se verifica si la cola c1 es un prefijo
-- de la cola c2. Por ejemplo,
--   prefijoCola c3 c2 == False
--   prefijoCola c5 c1 == True
-----

```

```

prefijoCola :: Eq a => Cola a -> Cola a -> Bool
prefijoCola c1 c2
  | esVacia c1 = True
  | esVacia c2 = False
  | otherwise  = pc1 == pc2 && prefijoCola rc1 rc2
where pc1 = primero c1
        rc1 = resto c1
        pc2 = primero c2
        rc2 = resto c2

```

```

-----
-- Ejercicio 11: Definir la función
--   subCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (subCola c1 c2) se verifica si c1 es una subcola de c2. Por
-- ejemplo,
--   subCola c2 c1 == False
--   subCola c3 c1 == True
-----

```

```

subCola :: Eq a => Cola a -> Cola a -> Bool
subCola c1 c2
  | esVacia c1 = True
  | esVacia c2 = False
  | pc1 == pc2 = prefijoCola rc1 rc2 || subCola c1 rc2
  | otherwise  = subCola c1 rc2
where pc1 = primero c1
        rc1 = resto c1
        pc2 = primero c2
        rc2 = resto c2

```



```
-- Ejercicio 12: Definir la función
--   ordenadaCola :: Ord a => Cola a -> Bool
-- tal que (ordenadaCola c) se verifica si los elementos de la cola c
-- están ordenados en orden creciente. Por ejemplo,
--   ordenadaCola c6 == True
--   ordenadaCola c4 == False
```

```
ordenadaCola :: Ord a => Cola a -> Bool
ordenadaCola c
  | esVacia c = True
  | esVacia rc = True
  | otherwise = pc <= prc && ordenadaCola rc
  where pc = primero c
        rc = resto c
        prc = primero rc
```

```
-- Ejercicio 13.1: Definir una función
--   lista2Cola :: [a] -> Cola a
-- tal que (lista2Cola xs) es una cola formada por los elementos de xs.
-- Por ejemplo,
--   lista2Cola [1..6] == C [1,2,3,4,5,6]
```

```
lista2Cola :: [a] -> Cola a
lista2Cola xs = foldr inserta vacia (reverse xs)
```

```
-- Ejercicio 13.2: Definir una función
--   cola2Lista :: Cola a -> [a]
-- tal que (cola2Lista c) es la lista formada por los elementos de p.
-- Por ejemplo,
--   cola2Lista c2 == [17,14,11,8,5,2]
```

```
cola2Lista :: Cola a -> [a]
cola2Lista c
  | esVacia c = []
  | otherwise = pc : cola2Lista rc
```

```

where pc = primero c
        rc = resto c

```

```

-----
-- Ejercicio 13.3. Comprobar con QuickCheck que la función cola2Lista es
-- la inversa de lista2Cola, y recíprocamente.
-----

```

```

prop_cola2Lista :: Cola Int -> Bool
prop_cola2Lista c =
  lista2Cola (cola2Lista c) == c

```

```

-- ghci> quickCheck prop_cola2Lista
-- +++ OK, passed 100 tests.

```

```

prop_lista2Cola :: [Int] -> Bool
prop_lista2Cola xs =
  cola2Lista (lista2Cola xs) == xs

```

```

-- ghci> quickCheck prop_lista2Cola
-- +++ OK, passed 100 tests.

```

```

-----
-- Ejercicio 14: Definir la función
--   maxCola :: Ord a => Cola a -> a
-- tal que (maxCola c) es el mayor de los elementos de la cola c. Por
-- ejemplo,
--   maxCola c4 == 10
-----

```

```

maxCola :: Ord a => Cola a -> a
maxCola c
  | esVacia c = error "cola vacia"
  | esVacia rc = pc
  | otherwise = max pc (maxCola rc)
where pc = primero c
        rc = resto c

```

```

prop_maxCola c =
  not (esVacia c) ==>

```

```

maxCola c == maximum (cola2Lista c)

-- ghci> quickCheck prop_maxCola
-- +++ OK, passed 100 tests.

-----
-- Generador de colas                                     --
-----

-- genCola es un generador de colas de enteros. Por ejemplo,
-- ghci> sample genCola
-- C ([],[1])
-- C ([],[1])
-- C ([],[1])
-- C ([],[1])
-- C ([7,8,4,3,7],[5,3,3])
-- C ([],[1])
-- C ([1],[13])
-- C ([18,28],[12,21,28,28,3,18,14])
-- C ([47],[64,45,7])
-- C ([8],[1])
-- C ([42,112,178,175,107],[1])
genCola :: (Num a, Arbitrary a) => Gen (Cola a)
genCola = frequency [(1, return vacia),
                    (30, do n <- choose (10,100)
                          xs <- vectorOf n arbitrary
                          return (creaCola xs))]
    where creaCola = foldr inserta vacia

-- El tipo cola es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
    arbitrary = genCola

```


Relación 15

Vectores y matrices

15.1. Ejercicios propuestos

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es hacer ejercicios sobre vectores y  
-- matrices con el tipo de tablas de las tablas, definido en el módulo  
-- Data.Array y explicado en el tema 18 cuyas transparencias se  
-- encuentran en  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-18t.pdf  
-- Además, en algunos ejemplos de usan matrices con números racionales.  
-- En Haskell, el número racional  $x/y$  se representa por  $x\%y$ . El TAD de  
-- los números racionales está definido en el módulo Data.Ratio.  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
import Data.Array  
import Data.Ratio  
  
-- -----  
-- Tipos de los vectores y de las matrices --  
-- -----  
  
-- Los vectores son tablas cuyos índices son números naturales.
```

```
type Vector a = Array Int a
```

```
-- Las matrices son tablas cuyos índices son pares de números
-- naturales.
```

```
type Matriz a = Array (Int,Int) a
```

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

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

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

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

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

```
-----
-- Ejercicio 3. Definir la función
```

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

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

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

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

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

```
dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas p)
```

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

```
separa :: Int -> [a] -> [[a]]
```

```
separa _ [] = []
separa n xs = take n xs : separa n (drop n xs)
```

```
-----
-- Ejercicio 7. Definir la función
--   matrizLista :: Num a => Matriz a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
--   ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> m
--   array ((1,1),(2,3)) [((1,1),5),((1,2),1),((1,3),0),
--                        ((2,1),3),((2,2),2),((2,3),6)]
--   ghci> matrizLista m
--   [[5,1,0],[3,2,6]]
-----
```

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

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

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

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

```
-----
-- Ejercicio 9. Definir la función
```



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

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

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

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

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

```

--      [1,2,5]
-----

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

-- Producto de matrices
-----

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

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

-- Ejercicio 13. Definir la función
--   prodMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
--   tal que (prodMatrices p q) es el producto de las matrices p y q. Por
--   ejemplo,
--   ghci> let p = listaMatriz [[3,1],[2,4]]
--   ghci> prodMatrices p p
--   array ((1,1),(2,2)) [((1,1),11),((1,2),7),((2,1),14),((2,2),18)]
--   ghci> matrizLista (prodMatrices p p)
--   [[11,7],[14,18]]
--   ghci> let q = listaMatriz [[7],[5]]
--   ghci> prodMatrices p q
--   array ((1,1),(2,1)) [((1,1),26),((2,1),34)]
--   ghci> matrizLista (prodMatrices p q)
--   [[26],[34]]

```

```

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

```

```

-----
-- Traspuestas y simétricas
-----

```

```

-----
-- Ejercicio 14. Definir la función
--   traspuesta :: Num a => Matriz a -> Matriz a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> traspuesta p
--   array ((1,1),(3,2)) [((1,1),5),((1,2),3),
--                         ((2,1),1),((2,2),2),
--                         ((3,1),0),((3,2),6)]
--   ghci> matrizLista (traspuesta p)
--   [[5,3],[1,2],[0,6]]
-----

```

```

traspuesta :: Num a => Matriz a -> Matriz a
traspuesta p =
  array ((1,1),(n,m))
    [((i,j), p!(j,i)) | i <- [1..n], j <- [1..m]]
  where (m,n) = dimension p

```

```

-----
-- Ejercicio 15. Definir la función
--   esCuadrada :: Num a => Matriz a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> esCuadrada p

```

```
-- False
-- ghci> let q = listaMatriz [[5,1],[3,2]]
-- ghci> esCuadrada q
-- True
```

```
-----
esCuadrada :: Num a => Matriz a -> Bool
esCuadrada x = numFilas x == numColumnas x
```

```
-----
-- Ejercicio 16. Definir la función
--   esSimetrica :: (Num a, Eq a) => Matriz a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
-- ghci> esSimetrica p
-- True
-- ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
-- ghci> esSimetrica q
-- False
```

```
-----
esSimetrica :: (Num a, Eq a) => Matriz a -> Bool
esSimetrica x = x == traspuesta x
```

```
-----
-- Diagonales de una matriz
```

```
-----
-- Ejercicio 17. Definir la función
--   diagonalPral :: Num a => Matriz a -> Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> diagonalPral p
-- array (1,2) [(1,5),(2,2)]
-- ghci> vectorLista (diagonalPral p)
-- [5,2]
```

```

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

```

```

-----
-- Ejercicio 18. Definir la función
--   diagonalSec :: Num a => Matriz a -> Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> diagonalSec p
--   array (1,2) [(1,1),(2,3)]
--   ghci> vectorLista (diagonalSec p)
--   [1,3]
--   ghci> let q = traspuesta p
--   ghci> matrizLista q
--   [[5,3],[1,2],[0,6]]
--   ghci> vectorLista (diagonalSec q)
--   [3,1]
-----

```

```

diagonalSec :: Num a => Matriz a -> Vector a
diagonalSec p = array (1,n) [(i,p!(i,n+1-i)) | i <- [1..n]]
  where n = min (numFilas p) (numColumnas p)

```

```

-----
-- Submatrices
-----

```

```

-----
-- Ejercicio 19. Definir la función
--   submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> submatriz 2 3 p
--   array ((1,1),(2,2)) [((1,1),5),((1,2),1),((2,1),4),((2,2),6)]
--   ghci> matrizLista (submatriz 2 3 p)
--   [[5,1],[4,6]]
-----

```

```

-----
submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
  array ((1,1), (m-1,n -1))
    [((k,l), p ! f k l) | k <- [1..m-1], l <- [1.. n-1]]
  where (m,n) = dimension p
        f k l | k < i  && l < j  = (k,l)
              | k >= i && l < j  = (k+1,l)
              | k < i  && l >= j = (k,l+1)
              | otherwise      = (k+1,l+1)

```

```

-----
-- Transformaciones elementales
-----

```

```

-----
-- Ejercicio 20. Definir la función
--   intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> intercambiaFilas 1 3 p
--   array ((1,1),(3,3)) [((1,1),4),((1,2),6),((1,3),9),
--                        ((2,1),3),((2,2),2),((2,3),6),
--                        ((3,1),5),((3,2),1),((3,3),0)]
--   ghci> matrizLista (intercambiaFilas 1 3 p)
--   [[4,6,9],[3,2,6],[5,1,0]]
-----

```

```

intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaFilas k l p =
  array ((1,1), (m,n))
    [((i,j), p ! f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = (l,j)
              | i == l    = (k,j)
              | otherwise = (i,j)

```

```
-- Ejercicio 21. Definir la función
--   intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
--   tal que (intercambiaColumnas k l p) es la matriz obtenida
--   intercambiando las columnas k y l de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> matrizLista (intercambiaColumnas 1 3 p)
--   [[0,1,5],[6,2,3],[9,6,4]]
```

```
intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaColumnas k l p =
  array ((1,1), (m,n))
    [((i,j), p ! f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | j == k    = (i,l)
              | j == l    = (i,k)
              | otherwise = (i,j)
```

```
-- Ejercicio 22. Definir la función
--   multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
--   tal que (multFilaPor k x p) es a matriz obtenida multiplicando la
--   fila k de la matriz p por el número x. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> matrizLista (multFilaPor 2 3 p)
--   [[5,1,0],[9,6,18],[4,6,9]]
```

```
multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
multFilaPor k x p =
  array ((1,1), (m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = x*(p!(i,j))
              | otherwise = p!(i,j)
```

```
-- Ejercicio 23. Definir la función
--   sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
--   tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
```

```
-- a la fila k d la matriz p. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> matrizLista (sumaFilaFila 2 3 p)
-- [[5,1,0],[7,8,15],[4,6,9]]
```

```
-----
sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
sumaFilaFila k l p =
  array ((1,1), (m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = p!(i,j) + p!(l,j)
              | otherwise = p!(i,j)
```

```
-----
-- Ejercicio 24. Definir la función
-- sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> matrizLista (sumaFilaPor 2 3 10 p)
-- [[5,1,0],[43,62,96],[4,6,9]]
```

```
-----
sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
sumaFilaPor k l x p =
  array ((1,1), (m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = p!(i,j) + x*p!(l,j)
              | otherwise = p!(i,j)
```

```
-----
-- Triangularización de matrices
```

```
-----
-- Ejercicio 25. Definir la función
-- buscaIndiceDesde :: (Num a, Eq a) =>
-- Matriz a -> Int -> Int -> Maybe Int
```



```

-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> buscaIndiceDesde p 3 2
-- Just 2
-- ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
-- ghci> buscaIndiceDesde q 3 2
-- Nothing

```

```

-----
buscaIndiceDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
  | null xs    = Nothing
  | otherwise  = Just (head xs)
  where xs = [k | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]

```

```

-----
-- Ejercicio 26. Definir la función
-- buscaPivoteDesde :: (Num a, Eq a) =>
--                   Matriz a -> Int -> Int -> Maybe a
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> buscaPivoteDesde p 3 2
-- Just 6
-- ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
-- ghci> buscaPivoteDesde q 3 2
-- Nothing

```

```

-----
buscaPivoteDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
  | null xs    = Nothing
  | otherwise  = Just (head xs)
  where xs = [y | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]

```

```

-----
-- Ejercicio 27. Definir la función
-- anuladaColumnaDesde :: (Num a, Eq a) =>

```

```

--           Int -> Int -> Matriz a -> Bool
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
--   ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--   ghci> anuladaColumnaDesde q 3 2
--   True
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> anuladaColumnaDesde p 3 2
--   False

```

```

-----
anuladaColumnaDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
  buscaIndiceDesde p j (i+1) == Nothing

```

```

-----
-- Ejercicio 28. Definir la función
--   anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
--                           Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
--   ghci> let p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matriz Double
--   ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
--   [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]

```

```

-----
anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
  Matriz a -> Int -> Int -> Matriz a
anulaEltoColumnaDesde p j i =
  sumaFilaPor l i (-(p!(l,j)/a)) p
  where Just l = buscaIndiceDesde p j (i+1)
        a      = p!(i,j)

```

```

-----
-- Ejercicio 29. Definir la función
--   anulaColumnaDesde :: (Fractional a, Eq a) =>
--                       Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando

```

```
-- todos los elementos de la columna j de la matriz p por debajo de la
-- posición (i,j) (se supone que el elemnto p_(i,j) es no nulo). Por
-- ejemplo,
-- ghci> let p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matriz Double
-- ghci> matrizLista (anulaColumnaDesde p 2 1)
-- [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
-- ghci> let p = listaMatriz [[4,5],[2,7%2],[6,10]]
-- ghci> matrizLista (anulaColumnaDesde p 1 1)
-- [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
```

```
-----
anulaColumnaDesde :: (Fractional a, Eq a) =>
                    Matriz a -> Int -> Int -> Matriz a
```

```
anulaColumnaDesde p j i
  | anuladaColumnaDesde p j i = p
  | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i
```

```
-----
-- Algoritmo de Gauss para triangularizar matrices
-----
```

```
-----
-- Ejercicio 30. Definir la función
```

```
elementosNoNulosColDesde :: (Num a, Eq a) =>
                            Matriz a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
-- ghci> let p = listaMatriz [[3,2],[5,1],[0,4]]
-- ghci> elementosNoNulosColDesde p 1 2
-- [5]
```

```
-----
elementosNoNulosColDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> [a]
```

```
elementosNoNulosColDesde p j i =
  [x | ((k,j'),x) <- assocs p, x /= 0, j' == j, k >= i]
```

```
-----
-- Ejercicio 31. Definir la función
```

```
existeColNoNulaDesde :: (Num a, Eq a) =>
                        Matriz a -> Int -> Int -> Bool
```

```
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la j; es decir, si la submatriz de p obtenida eliminando
-- las i-1 primeras filas y las j-1 primeras columnas es no nula. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
-- ghci> existeColNoNulaDesde p 2 2
-- False
-- ghci> let q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
-- ghci> existeColNoNulaDesde q 2 2
```

```
-----
existeColNoNulaDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Bool
existeColNoNulaDesde p j i =
  or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
  where n = numColumnas p
```

```
-----
-- Ejercicio 32. Definir la función
-- menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
--                               Matriz a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
-- ghci> let p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
-- ghci> menorIndiceColNoNulaDesde p 2 2
-- Just 2
-- ghci> let q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
-- ghci> menorIndiceColNoNulaDesde q 2 2
-- Just 3
-- ghci> let r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
-- ghci> menorIndiceColNoNulaDesde r 2 2
-- Nothing
```

```
-----
menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
                               Matriz a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
  | null js    = Nothing
  | otherwise = Just (head js)
```

```

where n = numColumnas p
        js = [j' | j' <- [j..n],
              not (null (elementosNoNulosColDesde p j' i))]

-----
-- Ejercicio 33. Definir la función
--   gaussAux :: (Fractional a, Eq a) =>
--             Matriz a -> Int -> Int -> Matriz a
-- tal que (gauss p) es la matriz que en el que las i-1 primeras filas y
-- las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
--   1. Si la dimensión de p es (i,j), entonces p.
--   2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--      primeras columnas es nulas, entonces p.
--   3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
--   3.1. j' la primera columna a partir de la j donde p tiene
--        algún elemento no nulo a partir de la fila i,
--   3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--        de p,
--   3.3. i' la primera fila a partir de la i donde la columna j de
--        p1 tiene un elemento no nulo,
--   3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--        la matriz p1 y
--   3.5. p' la matriz obtenida anulando todos los elementos de la
--        columna j de p2 por debajo de la fila i.
-- Por ejemplo,
--   ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
--   ghci> matrizLista (gaussAux p 2 2)
--   [[1.0,2.0,3.0],[1.0,2.0,4.0],[2.0,0.0,1.0]]
-----

gaussAux :: (Fractional a, Eq a) => Matriz a -> Int -> Int -> Matriz a
gaussAux p i j
  | dimension p == (i,j)           = p -- 1
  | not (existeColNoNulaDesde p j i) = p -- 2
  | otherwise                       = gaussAux p' (i+1) (j+1) -- 3
where Just j' = menorIndiceColNoNulaDesde p j i -- 3.1
        p1     = intercambiaColumnas j j' p -- 3.2
        Just i' = buscaIndiceDesde p1 j i -- 3.3
        p2     = intercambiaFilas i i' p1 -- 3.4

```

`p'` = `anulaColumnaDesde p2 j i` -- 3.5

```

-----
-- Ejercicio 34. Definir la función
--   gauss :: (Fractional a, Eq a) => Matriz a -> Matriz a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
--   ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--   ghci> gauss p
--   array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                         ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                         ((3,1),0.0),((3,2),0.0),((3,3),0.0)]
--   ghci> matrizLista (gauss p)
--   [[1.0,3.0,2.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
--   ghci> let p = listaMatriz [[3.0,2,3],[1,2,4],[1,2,5]]
--   ghci> matrizLista (gauss p)
--   [[3.0,2.0,3.0],[0.0,1.3333333333333335,3.0],[0.0,0.0,1.0]]
--   ghci> let p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
--   ghci> matrizLista (gauss p)
--   [[3 % 1,2 % 1,3 % 1],[0 % 1,4 % 3,3 % 1],[0 % 1,0 % 1,1 % 1]]
--   ghci> let p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
--   ghci> matrizLista (gauss p)
--   [[1.0,3.0,0.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
-----

```

```

gauss :: (Fractional a, Eq a) => Matriz a -> Matriz a
gauss p = gaussAux p 1 1

```

```

-----
-- Determinante
-----

```

```

-----
-- Ejercicio 35. Definir la función
--   gaussCAux :: (Fractional a, Eq a) =>
--               Matriz a -> Int -> Int -> Int -> Matriz a
-- tal que (gaussCAux p i j c) es el par (n,q) donde q es la matriz que
-- en el que las i-1 primeras filas y las j-1 primeras columnas son las
-- de p y las restantes están triangularizadas por el método de Gauss;
-- es decir,

```

```

-- 1. Si la dimensión de p es (i,j), entonces p.
-- 2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--    primeras columnas es nulas, entonces p.
-- 3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
-- 3.1. j' la primera columna a partir de la j donde p tiene
--     algún elemento no nulo a partir de la fila i,
-- 3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--     de p,
-- 3.3. i' la primera fila a partir de la i donde la columna j de
--     p1 tiene un elemento no nulo,
-- 3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--     la matriz p1 y
-- 3.5. p' la matriz obtenida anulando todos los elementos de la
--     columna j de p2 por debajo de la fila i.
-- y n es c más el número de intercambios de columnas y filas que se han
-- producido durante el cálculo. Por ejemplo,
-- ghci> gaussCAux (listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]) 1 1 0
-- (1,array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                          ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                          ((3,1),0.0),((3,2),0.0),((3,3),0.0)])
-----

```

```

gaussCAux :: (Fractional a, Eq a) =>
    Matriz a -> Int -> Int -> Int -> (Int,Matriz a)
gaussCAux p i j c
  | dimension p == (i,j)           = (c,p)           -- 1
  | not (existeColNoNulaDesde p j i) = (c,p)         -- 2
  | otherwise                       = gaussCAux p' (i+1) (j+1) c' -- 3
  where Just j' = menorIndiceColNoNulaDesde p j i    -- 3.1
        p1     = intercambiaColumnas j j' p         -- 3.2
        Just i' = buscaIndiceDesde p1 j i           -- 3.3
        p2     = intercambiaFilas i i' p1          -- 3.4
        p'     = anulaColumnaDesde p2 j i          -- 3.5
        c'     = c + signum (abs (j-j')) + signum (abs (i-i'))
-----

```

```

-- Ejercicio 36. Definir la función
-- gaussC :: (Fractional a, Eq a) => Matriz a -> Matriz a
-- tal que (gaussC p) es el par (n,q), donde q es la triangularización
-- de la matriz p por el método de Gauss y n es el número de

```

```
-- intercambios de columnas y filas que se han producido durante el
-- cálculo. Por ejemplo,
-- ghci> gaussC (listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]])
-- (1,array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
-- ((2,1),0.0),((2,2),1.0),((2,3),0.0),
-- ((3,1),0.0),((3,2),0.0),((3,3),0.0)])
-----
```

```
gaussC :: (Fractional a, Eq a) => Matriz a -> (Int,Matriz a)
gaussC p = gaussCAux p 1 1 0
```

```
-- Ejercicio 37. Definir la función
-- determinante :: (Fractional a, Eq a) => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
-- ghci> determinante (listaMatriz [[1.0,2,3],[1,3,4],[1,2,5]])
-- 2.0
-----
```

```
determinante :: (Fractional a, Eq a) => Matriz a -> a
determinante p = (-1)^c * product (elems (diagonalPral p'))
  where (c,p') = gaussC p
```

15.2. Ejercicios de exámenes

```
-- Introducción -----
```

```
-- En esta relación se presenta una recopilación de ejercicios vectores
-- y matrices propuestos en exámenes de la asignatura.
```

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

```
import Data.Array
import Data.List
```



```
-- Nota. En la relación usaremos los tipos de los vectores y las matrices
-- definidos por
```

```
type Vector a = Array Int a
type Matriz a = Array (Int,Int) a
```

```
-----
-- Ejercicio 1. Definir la función
--   esTriangularS :: Num a => Matriz a -> Bool
-- tal que (esTriangularS p) se verifica si p es una matriz triangular
-- superior. Por ejemplo,
--   ghci> esTriangularS (listArray ((1,1),(3,3)) [1,2,1,0,4,7,0,0,5])
--   True
--   ghci> esTriangularS (listArray ((1,1),(3,3)) [1,2,3,1,2,4,1,2,5])
--   False
-----
```

```
esTriangularS:: (Num a, Eq a) => Matriz a -> Bool
esTriangularS p = and [p!(i,j) == 0 | i <- [1..m], j <- [1..n], i > j]
  where (_, (m,n)) = bounds p
```

```
-----
-- Ejercicio 2. Definir la función
--   potencia :: Num a => Matriz a -> Int -> Matriz a
-- tal que (potencia p n) es la potencia n-ésima de la matriz cuadrada
-- p. Por ejemplo, si q es la matriz definida por
--   q :: Matriz Int
--   q = listArray ((1,1),(2,2)) [1,1,1,0]
-- entonces
--   ghci> potencia q 2
--   array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),1),((2,2),1)]
--   ghci> potencia q 3
--   array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),2),((2,2),1)]
--   ghci> potencia q 4
--   array ((1,1),(2,2)) [((1,1),5),((1,2),3),((2,1),3),((2,2),2)]
-- ¿Qué relación hay entre las potencias de la matriz q y la sucesión de
-- Fibonacci?
-----
```

```

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

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

-- (identidad n) es la matriz identidad de orden n. Por ejemplo,
-- ghci> identidad 3
-- array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                       ((2,1),0),((2,2),1),((2,3),0),
--                       ((3,1),0),((3,2),0),((3,3),1)]
identidad :: Num a => Int -> Matriz a
identidad n =
  array ((1,1),(n,n))
    [((i,j),f i j) | i <- [1..n], j <- [1..n]]
  where f i j | i == j    = 1
              | otherwise = 0

-- (prodEscalar v1 v2) es el producto escalar de los vectores v1
-- y v2. Por ejemplo,
-- ghci> prodEscalar (listArray (1,3) [3,2,5]) (listArray (1,3) [4,1,2])
-- 24
prodEscalar :: Num a => Vector a -> Vector a -> a
prodEscalar v1 v2 =
  sum [i*j | (i,j) <- zip (elems v1) (elems v2)]

-- (filaMat i p) es el vector correspondiente a la fila i-ésima
-- de la matriz p. Por ejemplo,
-- filaMat 2 q == array (1,2) [(1,1),(2,0)]
filaMat :: Num a => Int -> Matriz a -> Vector a
filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
  where (_,(n)) = bounds p

-- (columnaMat j p) es el vector correspondiente a la columna
-- j-ésima de la matriz p. Por ejemplo,
-- columnaMat 2 q == array (1,2) [(1,1),(2,0)]
columnaMat :: Num a => Int -> Matriz a -> Vector a
columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]

```

```

where (_, (m, _)) = bounds p

-- (prodMatrices p q) es el producto de las matrices p y q. Por ejemplo,
-- ghci> prodMatrices q q
-- array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),1),((2,2),1)]
prodMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
prodMatrices p q =
  array ((1,1),(m,n))
    [((i,j), prodEscalar (filaMat i p) (columnaMat j q)) |
     i <- [1..m], j <- [1..n]]
where (_, (m,n)) = bounds p

-- Los sucesión de Fibonacci es 0,1,1,2,3,5,8,13,... Se observa que los
-- elementos de (potencia q n) son los términos de la sucesión en los
-- lugares n+1, n, n y n-1.

-----
-- Ejercicio 3. Definir la función
-- indicesMaximo :: (Num a, Ord a) => Matriz a -> [(Int,Int)]
-- tal que (indicesMaximo p) es la lista de los índices del elemento
-- máximo de la matriz p. Por ejemplo,
-- ghci> indicesMaximo (listArray ((1,1),(2,2)) [3,2,3,1])
-- [(1,1),(2,1)]
-----

indicesMaximo :: (Num a, Ord a) => Matriz a -> [(Int,Int)]
indicesMaximo p = [(i,j) | (i,j) <- indices p, p!(i,j) == m]
  where m = maximum (elems p)

-----
-- Ejercicio 4. Definir la función
-- antidiagonal :: (Num a, Eq a) => Matriz a -> Bool
-- tal que (antidiagonal m) se verifica si es cuadrada y todos los
-- elementos de m que no están en su diagonal secundaria son nulos. Por
-- ejemplo,
-- ghci> antidiagonal (listArray ((1,1),(3,3)) [0,0,4, 0,6,0, 0,0,0])
-- True
-- ghci> antidiagonal (listArray ((1,1),(3,3)) [7,0,4, 0,6,0, 0,0,5])
-- False
-----

```

```

-- m1, m2 :: Matriz Int
-- m1 = listArray ((1,1),(3,3)) [0,0,4, 0,6,0, 0,0,0]
-- m2 = listArray ((1,1),(3,3)) [7,0,4, 0,6,0, 0,0,5]

antidiagonal :: (Num a, Eq a) => Matriz a -> Bool
antidiagonal p =
  m == n && nula [p!(i,j) | i <- [1..n], j <- [1..n], j /= n+1-i]
  where (_,(m,n)) = bounds p

nula :: (Num a, Eq a) => [a] -> Bool
nula xs = xs == [0 | x <- xs]

-----
-- Ejercicio 5. Definir la función
--   posiciones :: Int -> Matriz Int -> [(Int,Int)]
-- tal que (posiciones x p) es la lista de las posiciones de la matriz p
-- cuyo valor es x. Por ejemplo,
--   ghci> let p = listArray ((1,1),(2,3)) [1,2,3,2,4,6] :: Matriz Int
--   ghci> p
--   array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
--                        ((2,1),2),((2,2),4),((2,3),6)]
--   ghci> posiciones 2 p
--   [(1,2),(2,1)]
--   ghci> posiciones 6 p
--   [(2,3)]
--   ghci> posiciones 7 p
--   []
-----

posiciones :: Int -> Matriz Int -> [(Int,Int)]
posiciones x p = [(i,j) | (i,j) <- indices p, p!(i,j) == x]

-----
-- Ejercicio 6. Definir la función
--   esEscalar :: Num a => Matriz a -> Bool
-- tal que (esEscalar p) se verifica si p es una matriz es escalar; es
-- decir, diagonal con todos los elementos de la diagonal principal
-- iguales. Por ejemplo,
--   esEscalar (listArray ((1,1),(3,3)) [5,0,0,0,5,0,0,0,5]) == True

```

```

--     esEscalar (listArray ((1,1),(3,3)) [5,0,0,1,5,0,0,0,5]) == False
--     esEscalar (listArray ((1,1),(3,3)) [5,0,0,0,6,0,0,0,5]) == False
-----

esEscalar:: (Num a, Eq a) => Matriz a -> Bool
esEscalar p = esDiagonal p && todosIguales (elems (diagonalPral p))

-- (esDiagonal p) se verifica si la matriz p es diagonal. Por ejemplo.
--     esDiagonal (listArray ((1,1),(3,3)) [5,0,0,0,6,0,0,0,5]) == True
--     esDiagonal (listArray ((1,1),(3,3)) [5,0,0,1,5,0,0,0,5]) == False
esDiagonal:: (Num a, Eq a) => Matriz a -> Bool
esDiagonal p = all (==0) [p!(i,j) | i<-[1..m],j<-[1..n], i/=j]
    where (_,(m,n)) = bounds p

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--     todosIguales [5,5,5] == True
--     todosIguales [5,6,5] == False
todosIguales :: Eq a => [a] -> Bool
todosIguales (x:y:ys) = x == y && todosIguales (y:ys)
todosIguales _ = True

-- (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
--     ghci> diagonalPral (listArray ((1,1),(3,3)) [5,0,0,1,6,0,0,2,4])
--     array (1,3) [(1,5),(2,6),(3,4)]
diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
    where n          = min k l
          (_,(k,l)) = bounds p
-----

-- Ejercicio 7. Definir la función
--     determinante:: Matriz Double -> Double
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--     ghci> determinante (listArray ((1,1),(3,3)) [2,0,0,0,3,0,0,0,1])
--     6.0
--     ghci> determinante (listArray ((1,1),(3,3)) [1..9])
--     0.0

```

```
-- ghci> determinante (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
-- -33.0
```

```
determinante :: Matriz Double -> Double
```

```
determinante p
  | (m,n) == (1,1) = p!(1,1)
  | otherwise =
    sum [((-1)^(i+1))*(p!(i,1))*determinante (submatriz i 1 p)
         | i <- [1..m]]
where (_,(m,n)) = bounds p
```

```
-- (submatriz i j p) es la submatriz de p obtenida eliminado la fila i y
-- la columna j. Está definida en el ejercicio ???.
```

```
-- Ejercicio 8. Definir la función
```

```
-- aplicaT :: (Ix a, Num b) => Array a b -> (b -> c) -> Array a c
-- tal que (aplicaT t f) es la tabla obtenida aplicado la función f a
-- los elementos de la tabla t. Por ejemplo,
-- ghci> aplicaT (array (1,5) [(1,6),(2,3),(3,-1),(4,9),(5,20)]) (+1)
-- array (1,5) [(1,7),(2,4),(3,0),(4,10),(5,21)]
-- ghci> :{
-- *Main| aplicaT (array ((1,1),(2,3)) [((1,1),3),((1,2),-1),((1,3),0),
-- *Main|           ((2,1),0),((2,2),0),((2,3),-1)])
-- *Main|           (*2)
-- *Main| :}
-- array ((1,1),(2,3)) [((1,1),6),((1,2),-2),((1,3),0),
--                       ((2,1),0),((2,2),0),((2,3),-2)]
```

```
aplicaT :: (Ix a, Num b) => Array a b -> (b -> c) -> Array a c
aplicaT t f = listArray (bounds t) [f e | e <- elems t]
```

```
-- Ejercicio 9. Diremos que una matriz es creciente si para toda
-- posición (i,j), el valor de dicha posición es menor o igual que los
-- valores en las posiciones adyacentes de índice superior; es decir,
-- (i+1,j), (i,j+1) e (i+1,j+1) siempre y cuando dichas posiciones
-- existan en la matriz.
```

```
--
-- Definir la función
--   matrizCreciente :: (Num a, Ord a) => Matriz a -> Bool
-- tal que (matrizCreciente p) se verifica si la matriz p es
-- creciente. Por ejemplo,
--   ghci> matrizCreciente (listArray ((1,1),(3,3)) [1,2,3, 2,3,4, 3,4,5])
--   True
--   ghci> matrizCreciente (listArray ((1,1),(3,3)) [1,2,3, 2,1,4, 3,4,5])
--   False
-----
```

```
matrizCreciente :: (Num a, Ord a) => Matriz a -> Bool
matrizCreciente p =
  and ([p!(i,j) <= p!(i,j+1) | i <- [1..m], j <- [1..n-1]] ++
       [p!(i,j) <= p!(i+1,j) | i <- [1..m-1], j <- [1..n]] ++
       [p!(i,j) <= p!(i+1,j+1) | i <- [1..m-1], j <- [1..n-1]])
  where (m,n) = snd (bounds p)
-----
```

```
-- Ejercicio 10. Dada una matriz numérica A de dimensiones (m,n) y una
-- matriz booleana B de las mismas dimensiones, y dos funciones f y g,
-- la transformada de A respecto de B, f y g es la matriz C (de las
-- mismas dimensiones), tal que, para cada celda (i,j):
```

```
--   C(i,j) = f(A(i,j)) si B(i,j) es verdadero
```

```
--   C(i,j) = g(A(i,j)) si B(i,j) es falso
```

```
-- Por ejemplo, si A y B son las matrices
```

```
--   |1 2|   |True False|
```

```
--   |3 4|   |False True |
```

```
-- respectivamente, y f y g son dos funciones tales que f(x) = x+1 y
-- g(x) = 2*x, entonces la transformada de A respecto de B, f y g es
```

```
--   |2 4|
```

```
--   |6 5|
```

```
-- Definir la función
```

```
--   transformada :: Matriz a -> Matriz Bool -> (a -> b) -> (a -> b) -> Matriz b
```

```
-- tal que (transformada a b f g) es la transformada de A respecto de B,
```

```
-- f y g. Por ejemplo,
```

```
--   ghci> let a = listArray ((1,1),(2,2)) [1,2,3,4] :: Matriz Int
```

```
--   ghci> let b = listArray ((1,1),(2,2)) [True,False,False,True] :: Matriz Bool
```

```
--   ghci> transformada a b (+1) (*2)
```

```

-- array ((1,1),(2,2)) [((1,1),2),((1,2),4),((2,1),6),((2,2),5)]
-----

transformada :: Matriz a -> Matriz Bool -> (a -> b) -> (a -> b) -> Matriz b
transformada a b f g =
  array ((1,1),(m,n)) [((i,j),aplica i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = snd (bounds a)
        aplica i j | b!(i,j)   = f (a!(i,j))
                  | otherwise = g (a!(i,j))
-----

-- Ejercicio 11.1. Un vector se denomina estocástico si todos sus
-- elementos son mayores o iguales que 0 y suman 1.
--
-- Definir la función
--   vectorEstocastico :: Vector Float -> Bool
-- tal que (vectorEstocastico v) se verifica si v es estocástico. Por
-- ejemplo,
--   vectorEstocastico (listArray (1,5) [0.1, 0.2, 0, 0, 0.7]) == True
--   vectorEstocastico (listArray (1,5) [0.1, 0.2, 0, 0, 0.9]) == False
-----

vectorEstocastico :: Vector Float -> Bool
vectorEstocastico v = all (>=0) xs && sum xs == 1
  where xs = elems v
-----

-- Ejercicio 11.2. Una matriz se denomina estocástica si sus columnas
-- son vectores estocásticos.
--
-- Definir la función
--   matrizEstocastica :: Matriz Float -> Bool
-- tal que (matrizEstocastico p) se verifica si p es estocástica. Por
-- ejemplo,
--   matrizEstocastica (listArray ((1,1),(2,2)) [0.1,0.2,0.9,0.8]) == True
--   matrizEstocastica (listArray ((1,1),(2,2)) [0.1,0.2,0.3,0.8]) == False
-----

matrizEstocastica :: Matriz Float -> Bool
matrizEstocastica p = all vectorEstocastico (columnas p)

```



```
-- (columnas p) es la lista de las columnas de la matriz p. Por ejemplo,
-- ghci> columnas (listArray ((1,1),(2,3)) [1..6])
-- [array (1,2) [(1,1.0),(2,4.0)],
--   array (1,2) [(1,2.0),(2,5.0)],
--   array (1,2) [(1,3.0),(2,6.0)]]
-- ghci> columnas (listArray ((1,1),(3,2)) [1..6])
-- [array (1,3) [(1,1.0),(2,3.0),(3,5.0)],
--   array (1,3) [(1,2.0),(2,4.0),(3,6.0)]]
```

```
columnas :: Matriz Float -> [Vector Float]
```

```
columnas p =
```

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

```
  where (_, (m,n)) = bounds p
```

```
-----
-- Ejercicio 12. Definir la función
--   maximaSuma :: Matriz Int -> Int
-- tal que (maximaSuma p) es el máximo de las sumas de las listas de
-- elementos de la matriz p tales que cada elemento pertenece sólo a una
-- fila y a una columna. Por ejemplo,
-- ghci> maximaSuma (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
-- 17
-- ya que las selecciones, y sus sumas, de la matriz
--   |1 2 3|
--   |8 4 9|
--   |5 6 7|
-- son
--   [1,4,7] --> 12
--   [1,9,6] --> 16
--   [2,8,7] --> 17
--   [2,9,5] --> 16
--   [3,8,6] --> 17
--   [3,4,5] --> 12
-- Hay dos selecciones con máxima suma: [2,8,7] y [3,8,6].
-----
```

```
maximaSuma :: Matriz Int -> Int
```

```
maximaSuma p = maximum [sum xs | xs <- selecciones p]
```

```
-- (selecciones p) es la lista de las selecciones en las que cada
```

```

-- elemento pertenece a un única fila y a una única columna de la matriz
-- p. Por ejemplo,
-- ghci> selecciones (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
-- [[1,4,7],[2,8,7],[3,4,5],[2,9,5],[3,8,6],[1,9,6]]
selecciones :: Matriz Int -> [[Int]]
selecciones p =
  [p!(i,j) | (i,j) <- ijs] |
  ijs <- [zip [1..n] xs | xs <- permutations [1..n]]]
  where (_,(m,n)) = bounds p

-- Nota: En la anterior definición se ha usado la función permutations
-- de Data.List. También se puede definir mediante
permutaciones :: [a] -> [[a]]
permutaciones [] = [[]]
permutaciones (x:xs) =
  concat [intercala x ys | ys <- permutaciones xs]

-- (intercala x ys) es la lista de las listas obtenidas intercalando x
-- entre los elementos de ys. Por ejemplo,
-- intercala 1 [2,3] == [[1,2,3],[2,1,3],[2,3,1]]
intercala :: a -> [a] -> [[a]]
intercala x [] = [[x]]
intercala x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala x ys]

-- 2ª solución (mediante submatrices):
maximaSuma2 :: Matriz Int -> Int
maximaSuma2 p
  | (m,n) == (1,1) = p!(1,1)
  | otherwise = maximum [p!(1,j)
    + maximaSuma2 (submatriz 1 j p) | j <- [1..n]]
  where (_,(m,n)) = bounds p

-- (submatriz i j p) es la matriz obtenida a partir de la p eliminando
-- la fila i y la columna j. Por ejemplo,
-- ghci> submatriz 2 3 (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
-- array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),5),((2,2),6)]
submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
  array ((1,1), (m-1,n -1))
    [((k,l), p ! f k l) | k <- [1..m-1], l <- [1..n-1]]

```

```

where (_, (m,n)) = bounds p
      f k l | k < i  && l < j = (k,l)
            | k >= i && l < j = (k+1,l)
            | k < i  && l >= j = (k,l+1)
            | otherwise      = (k+1,l+1)

```

-- *Ejercicio 13. Definir la función*

```
--   maximos :: Matriz Int -> [Int]
```

```
-- tal que (maximos p) es la lista de los máximos locales de la matriz
-- p; es decir de los elementos de p que son mayores que todos sus
-- vecinos. Por ejemplo,
```

```
--   ghci> maximos (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,0,2,5,4])
--   [9,7]
```

```
-- ya que los máximos locales de la matriz
```

```
--   |9 4 6 5|
```

```
--   |8 1 7 3|
```

```
--   |0 2 5 4|
```

```
-- son 9 y 7.
```

```
maximos :: Matriz Int -> [Int]
```

```
maximos p =
```

```
  [p!(i,j) | (i,j) <- indices p,
            and [p!(a,b) < p!(i,j) | (a,b) <- vecinos (i,j)]]
```

```
  where (_, (m,n)) = bounds p
```

```
    vecinos (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                             b <- [max 1 (j-1)..min n (j+1)],
                             (a,b) /= (i,j)]
```

```
-- Ejercicio 14. Entre dos matrices de la misma dimensión se puede
-- aplicar distintas operaciones binarias entre los elementos en la
-- misma posición. Por ejemplo, si a y b son las matrices
```

```
--   |3 4 6|   |1 4 2|
```

```
--   |5 6 7|   |2 1 2|
```

```
-- entonces a+b y a-b son, respectivamente
```

```
--   |4 8 8|   |2 0 4|
```

```
--   |7 7 9|   |3 5 5|
```

```

-- Definir la función
--   opMatriz :: (Int -> Int -> Int) ->
--             Matriz Int -> Matriz Int -> Matriz Int
-- tal que (opMatriz f p q) es la matriz obtenida aplicando la operación
-- f entre los elementos de p y q de la misma posición. Por ejemplo,
ghci> let a = listArray ((1,1),(2,3)) [3,4,6,5,6,7] :: Matriz Int
ghci> let b = listArray ((1,1),(2,3)) [1,4,2,2,1,2] :: Matriz Int
ghci> opMatriz (+) a b
array ((1,1),(2,3)) [((1,1),4),((1,2),8),((1,3),8),
                    ((2,1),7),((2,2),7),((2,3),9)]
ghci> opMatriz (-) a b
array ((1,1),(2,3)) [((1,1),2),((1,2),0),((1,3),4),
                    ((2,1),3),((2,2),5),((2,3),5)]
-----

-- 1ª definición
opMatriz :: (Int -> Int -> Int) ->
           Matriz Int -> Matriz Int -> Matriz Int
opMatriz f p q =
  array ((1,1),(m,n)) [((i,j), f (p!(i,j)) (q!(i,j)))
                      | i <- [1..m], j <- [1..n]]
  where (_,(m,n)) = bounds p

-- 2ª definición
opMatriz2 :: (Int -> Int -> Int) ->
            Matriz Int -> Matriz Int -> Matriz Int
opMatriz2 f p q =
  listArray (bounds p) [f x y | (x,y) <- zip (elems p) (elems q)]
-----

-- Ejercicio 15. Definir la función
--   algunMenor :: Matriz Int -> [Int]
-- tal que (algunMenor p) es la lista de los elementos de p que tienen
-- algún vecino menor que él. Por ejemplo,
--   algunMenor (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,4,2,5,4])
--   [9,4,6,5,8,7,4,2,5,4]
-- pues sólo el 1 y el 3 no tienen ningún vecino menor en la matriz
--   |9 4 6 5|
--   |8 1 7 3|
--   |4 2 5 4|

```

```

algunMenor :: Matriz Int -> [Int]
algunMenor p =
  [p!(i,j) | (i,j) <- indices p,
            or [p!(a,b) < p!(i,j) | (a,b) <- vecinos (i,j)]]
  where (_,(m,n)) = bounds p
        vecinos (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                                b <- [max 1 (j-1)..min n (j+1)],
                                (a,b) /= (i,j)]

```

```

-- Ejercicio 16.1. Definir la función
--   esAutovector :: (Fractional a, Eq a) =>
--                 Vector a -> Matriz a -> Bool
-- tal que (esAutovector v p) compruebe si v es un autovector de p
-- (es decir, el producto de v por p es un vector proporcional a
-- v). Por ejemplo,
--   ghci> let p1 = listArray ((1,1),(3,3)) [1,0,0,0,0,1,0,1,0] :: Matriz Float
--   ghci> let v1 = listArray (1,3) [0,-1,1] :: Vector Float
--   ghci> let v2 = listArray (1,3) [1,2,1] :: Vector Float
--   ghci> esAutovector v1 p1
--   True
--   ghci> esAutovector v2 p1
--   False

```

```

esAutovector :: (Fractional a, Eq a) => Vector a -> Matriz a -> Bool
esAutovector v p = proporcional (producto p v) v

```

```

-- (producto p v) es el producto de la matriz p por el vector v. Por
-- ejemplo,
--   producto p1 v1 = array (1,3) [(1,0.0),(2,1.0),(3,-1.0)]
--   producto p1 v2 = array (1,3) [(1,1.0),(2,1.0),(3,2.0)]
producto :: (Fractional a, Eq a) => Matriz a -> Vector a -> Vector a
producto p v =
  array (1,n) [(i, sum [p!(i,j)*v!j | j <- [1..n]]) | i <- [1..m]]
  where (_,n)      = bounds v
        (_,(m,_)) = bounds p

```

```

-- (proporcional v1 v2) se verifica si los vectores v1 y v2 son
-- proporcionales. Por ejemplo,
--   proporcional v1 v1           = True
--   proporcional v1 v2           = False
--   proporcional v1 (listArray (1,3) [0,-5,5]) = True
--   proporcional v1 (listArray (1,3) [0,-5,4]) = False
--   proporcional (listArray (1,3) [0,-5,5]) v1 = True
--   proporcional v1 (listArray (1,3) [0,0,0]) = True
--   proporcional (listArray (1,3) [0,0,0]) v1 = False
proporcional :: (Fractional a, Eq a) => Vector a -> Vector a -> Bool
proporcional v1 v2
  | esCero v1 = esCero v2
  | otherwise = and [v2!i == k*(v1!i) | i <- [1..n]]
  where (_,n) = bounds v1
        j     = minimum [i | i <- [1..n], v1!i /= 0]
        k     = (v2!j) / (v1!j)

-- (esCero v) se verifica si v es el vector 0.
esCero :: (Fractional a, Eq a) => Vector a -> Bool
esCero v = null [x | x <- elems v, x /= 0]

```

```

-----
-- Ejercicio 16.2. Definir la función
--   autovalorAsociado :: (Fractional a, Eq a) =>
--                       Matriz a -> Vector a -> Maybe a
-- tal que si v es un autovector de p, calcule el autovalor asociado.
-- Por ejemplo,
--   ghci> let p1 = listArray ((1,1),(3,3)) [1,0,0,0,0,1,0,1,0] :: Matriz Float
--   ghci> let v1 = listArray (1,3) [0,-1,1] :: Vector Float
--   ghci> let v2 = listArray (1,3) [1,2,1] :: Vector Float
--   autovalorAsociado p1 v1 == Just (-1.0)
--   autovalorAsociado p1 v2 == Nothing
-----

```

```

autovalorAsociado :: (Fractional a, Eq a) =>
                    Matriz a -> Vector a -> Maybe a
autovalorAsociado p v
  | esAutovector v p = Just (producto p v ! j / v ! j)
  | otherwise        = Nothing
  where (_,n) = bounds v

```

```

j      = minimum [i | i <- [1..n], v!i /= 0]

-----
-- Ejercicio 17. Definir la función
--   borraCols :: Int -> Int -> Matriz Int -> Matriz Int
-- tal que (borraCols j1 j2 p) es la matriz obtenida borrando las
-- columnas j1 y j2 (con j1 < j2) de la matriz p. Por ejemplo,
-- ghci> let p = listArray ((1,1),(2,4)) [1..8] :: Matriz Int
-- ghci> p
-- array ((1,1),(2,4)) [((1,1),1),((1,2),2),((1,3),3),((1,4),4),
--                       ((2,1),5),((2,2),6),((2,3),7),((2,4),8)]
-- ghci> borraCols 1 3 p
-- array ((1,1),(2,2)) [((1,1),2),((1,2),4),((2,1),6),((2,2),8)]
-- ghci> borraCols 2 3 p
-- array ((1,1),(2,2)) [((1,1),1),((1,2),4),((2,1),5),((2,2),8)]
-----

-- 1ª definición:
borraCols :: Int -> Int -> Matriz Int -> Matriz Int
borraCols j1 j2 p =
  borraCol (j2-1) (borraCol j1 p)

-- (borraCol j1 p) es la matriz obtenida borrando la columna j1 de la
-- matriz p. Por ejemplo,
-- ghci> let p = listArray ((1,1),(2,4)) [1..8]
-- ghci> borraCol 2 p
-- array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),4),
--                       ((2,1),5),((2,2),7),((2,3),8)]
-- ghci> borraCol 3 p
-- array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),4),
--                       ((2,1),5),((2,2),6),((2,3),8)]
borraCol :: Int -> Matriz Int -> Matriz Int
borraCol j1 p =
  array ((1,1),(m,n-1))
    [((i,j), f i j) | i <- [1..m], j <- [1..n-1]]
  where (_,(m,n)) = bounds p
        f i j | j < j1 = p!(i,j)
              | otherwise = p!(i,j+1)

-- 2ª definición:

```

```
borraCols2 :: Int -> Int -> Matriz Int -> Matriz Int
```

```
borraCols2 j1 j2 p =
  array ((1,1),(m,n-2))
    [((i,j), f i j) | i <- [1..m], j <- [1..n-2]]
  where (_,(m,n)) = bounds p
        f i j | j < j1      = p!(i,j)
              | j < j2-1   = p!(i,j+1)
              | otherwise   = p!(i,j+2)
```

```
-- 3ª definición:
```

```
borraCols3 :: Int -> Int -> Matriz Int -> Matriz Int
```

```
borraCols3 j1 j2 p =
  listArray ((1,1),(n,m-2)) [p!(i,j) | i <- [1..n], j <- [1..m], j/=j1 && j/=j2]
  where (_,(n,m)) = bounds p
```

```
-----
-- Ejercicio 18.1. Definir la función
```

```
-- cambiaM :: (Int, Int) -> Matriz Int -> Matriz Int
-- tal que (cambiaM i p) es la matriz obtenida cambiando en p los
-- elementos de la fila y la columna en i transformando los 0 en 1 y
-- viceversa. El valor en i cambia solo una vez. Por ejemplo,
-- ghci> cambiaM (2,3) (listArray ((1,1),(3,3)) [1,0,1, 0,7,1, 1,1,1])
-- array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                       ((2,1),1),((2,2),7),((2,3),0),
--                       ((3,1),1),((3,2),1),((3,3),0)]
```

```
-----
cambiaM :: (Int, Int) -> Matriz Int -> Matriz Int
```

```
cambiaM (a,b) p = array (bounds p) [((i,j),f i j) | (i,j) <- indices p]
  where f i j | i == a || j == b = cambia (p!(i,j))
              | otherwise       = p!(i,j)
        cambia x | x == 0      = 1
                  | x == 1      = 0
                  | otherwise   = x
```

```
-----
-- Ejercicio 18.2. Definir la función
```

```
-- quitaRepetidosFila :: Int -> Matriz Int -> Matriz Int
-- tal que (quitaRepetidosFila i p) es la matriz obtenida a partir de p
-- eliminando los elementos repetidos de la fila i y rellenando con
```



```

-- ceros al final hasta completar la fila. Por ejemplo,
-- ghci> let m1 = listArray ((1,1),(3,3)) [1,0,1, 0,7,1, 1,1,1] :: Matriz Int
-- ghci> quitaRepetidosFila 1 m1
-- array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                       ((2,1),0),((2,2),7),((2,3),1),
--                       ((3,1),1),((3,2),1),((3,3),1)]
-- ghci> quitaRepetidosFila 2 m1
-- array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                       ((2,1),0),((2,2),7),((2,3),1),
--                       ((3,1),1),((3,2),1),((3,3),1)]
-- ghci> quitaRepetidosFila 3 m1
-- array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                       ((2,1),0),((2,2),7),((2,3),1),
--                       ((3,1),1),((3,2),0),((3,3),0)]
-----

```

```

quitaRepetidosFila :: Int -> Matriz Int -> Matriz Int

```

```

quitaRepetidosFila x p =
  array (bounds p) [((i,j),f i j) | (i,j) <- indices p]
    where f i j | i == x    = (cambia (fila i p)) !! (j-1)
              | otherwise = p!(i,j)

```

```

-- (fila i p) es la fila i-ésima de la matriz p. Por ejemplo,

```

```

-- ghci> m1
-- array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                       ((2,1),0),((2,2),7),((2,3),1),
--                       ((3,1),1),((3,2),1),((3,3),1)]
-- ghci> fila 2 m1
-- [0,7,1]

```

```

fila :: Int -> Matriz Int -> [Int]

```

```

fila i p = [p!(i,j) | j <- [1..n]]
  where (_,(_,n)) = bounds p

```

```

-- (cambia xs) es la lista obtenida eliminando los elementos repetidos
-- de xs y completando con ceros al final para que tenga la misma
-- longitud que xs. Por ejemplo,

```

```

-- cambia [2,3,2,5,3,2] == [2,3,5,0,0,0]

```

```

cambia :: [Int] -> [Int]

```

```

cambia xs = ys ++ replicate (n-m) 0
  where ys = nub xs

```

```

n = length xs
m = length ys

```

```

-----
-- Ejercicio 19. Definir la función
-- sumaVecinos :: Matriz Int -> Matriz Int
-- tal que (sumaVecinos p) es la matriz obtenida al escribir en la
-- posición (i,j) la suma de los todos vecinos del elemento que ocupa
-- el lugar (i,j) en la matriz p. Por ejemplo,
-- ghci> sumaVecinos (listArray ((1,1),(3,3)) [0,1,3, 1,2,0, 0,5,7])
-- array ((1,1),(3,3)) [((1,1),4),((1,2), 6),((1,3), 3),
--                       ((2,1),8),((2,2),17),((2,3),18),
--                       ((3,1),8),((3,2),10),((3,3), 7)]
-----

```

```

sumaVecinos :: Matriz Int -> Matriz Int
sumaVecinos p =
  array ((1,1),(m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (_,(m,n)) = bounds p
        f i j = sum [p!(i+a,j+b) | a <- [-1..1], b <- [-1..1],
                                   a /= 0 || b /= 0,
                                   inRange (bounds p) (i+a,j+b)]

```

```

-----
-- Ejercicio 20. Definir la función
-- ampliaColumnas :: Matriz -> Matriz -> Matriz
-- tal que (ampliaColumnas p q) es la matriz construida añadiendo las
-- columnas de la matriz q a continuación de las de p (se supone que
-- tienen el mismo número de filas). Por ejemplo, si p y q representa
-- las dos primeras matrices, entonces (ampliaColumnas p q) es la
-- tercera
--   |0 1|   |4 5 6|   |0 1 4 5 6|
--   |2 3|   |7 8 9|   |2 3 7 8 9|
-- En Haskell,
-- ghci> let p = listArray ((1,1),(2,2)) [0..3] :: Matriz Int
-- ghci> let q = listArray ((1,1),(2,3)) [4..9] :: Matriz Int
-- ghci> ampliaColumnas p q
-- array ((1,1),(2,5))
--       [((1,1),0),((1,2),1),((1,3),4),((1,4),5),((1,5),6),

```

```

--          ((2,1),2),((2,2),3),((2,3),7),((2,4),8),((2,5),9)]
-----

ampliaColumnas :: Matriz a -> Matriz a -> Matriz a
ampliaColumnas p1 p2 =
  array ((1,1),(m,n1+n2)) [((i,j), f i j) | i <- [1..m], j <- [1..n1+n2]]
    where ((_,_), (m,n1)) = bounds p1
          ((_,_), (n1,n2)) = bounds p2
          f i j | j <= n1   = p1!(i,j)
                | otherwise = p2!(i,j-n1)

-----

-- Ejercicio 21. Una matriz cuadrada es bisimétrica si es simétrica
-- respecto de su diagonal principal y de su diagonal secundaria.
--
-- Definir la función
--   esBisimetrica :: Eq a => Matriz a -> Bool
-- tal que (esBisimetrica p) se verifica si p es bisimétrica. Por
-- ejemplo,
--   esBisimetrica ejM1 == True
--   esBisimetrica ejM2 == False
-- donde las matrices ejM1 y ejM2 están definidas por
--   ejM1, ejM2 :: Matriz Int
--   ejM1 = listArray ((1,1),(5,5)) [1,2,3,4,5,
--                                     2,6,8,9,4,
--                                     3,8,0,8,3,
--                                     4,9,8,6,2,
--                                     5,4,3,2,1]
--
--   ejM2 = listArray ((1,1),(3,3)) [1,2,3,
--                                     2,6,8,
--                                     3,8,0]
-----

ejM1, ejM2 :: Matriz Int
ejM1 = listArray ((1,1),(5,5)) [1,2,3,4,5,
                                  2,6,8,9,4,
                                  3,8,0,8,3,
                                  4,9,8,6,2,
                                  5,4,3,2,1]

```

```

ejM2 = listArray ((1,1),(3,3)) [1,2,3,
                               2,6,8,
                               3,8,0]

-- 1ª definición:
esBisimetrica :: Eq a => Matriz a -> Bool
esBisimetrica p =
  and [p!(i,j) == p!(j,i) | i <- [1..n], j <- [1..n]] &&
  and [p!(i,j) == p!(n+1-j,n+1-i) | i <- [1..n], j <- [1..n]]
  where ((_,_),(n,_)) = bounds p

-- 2ª definición:
esBisimetrica2 :: Eq a => Matriz a -> Bool
esBisimetrica2 p = p == simetrica p && p == simetricaS p

-- (simetrica p) es la simétrica de la matriz p respecto de la diagonal
-- principal. Por ejemplo,
--   ghci> simetrica (listArray ((1,1),(4,4)) [1..16])
--   array ((1,1),(4,4)) [((1,1),1),((1,2),5),((1,3), 9),((1,4),13),
--                        ((2,1),2),((2,2),6),((2,3),10),((2,4),14),
--                        ((3,1),3),((3,2),7),((3,3),11),((3,4),15),
--                        ((4,1),4),((4,2),8),((4,3),12),((4,4),16)]
simetrica :: Eq a => Matriz a -> Matriz a
simetrica p =
  array ((1,1),(n,n)) [((i,j),p!(j,i)) | i <- [1..n], j <- [1..n]]
  where ((_,_),(n,_)) = bounds p

-- (simetricaS p) es la simétrica de la matriz p respecto de la diagonal
-- secundaria. Por ejemplo,
--   ghci> simetricaS (listArray ((1,1),(4,4)) [1..16])
--   array ((1,1),(4,4)) [((1,1),16),((1,2),12),((1,3),8),((1,4),4),
--                        ((2,1),15),((2,2),11),((2,3),7),((2,4),3),
--                        ((3,1),14),((3,2),10),((3,3),6),((3,4),2),
--                        ((4,1),13),((4,2), 9),((4,3),5),((4,4),1)]
simetricaS :: Matriz a -> Matriz a
simetricaS p =
  array ((1,1),(n,n)) [((i,j),p!(n+1-j,n+1-i)) | i <- [1..n], j <- [1..n]]
  where ((_,_),(n,_)) = bounds p

```



```
-- es decir, el resultado es la matriz
```

```
-- | 7 5 4|
```

```
-- | 9 3 10|
```

```
-- |13 11 12|
```

```
-----
```

```
sumaColumnas :: Matriz Int -> Matriz Int
```

```
sumaColumnas p =
```

```
  array ((1,1),(m,n))
```

```
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
```

```
  where (_,(m,n)) = bounds p
```

```
    f i 1 = p!(i,1) + p!(i,m)
```

```
    f i j = p!(i,j) + p!(i,j-1)
```

```
-----
```

```
-- Ejercicio 24. Las matrices piramidales son las formadas por unos y
```

```
-- ceros de forma que los unos forman una pirámide. Por ejemplo,
```

```
-- |1|   |0 1 0|   |0 0 1 0 0|   |0 0 0 1 0 0 0|
```

```
--      |1 1 1|   |0 1 1 1 0|   |0 0 1 1 1 0 0|
```

```
--                |1 1 1 1 1|   |0 1 1 1 1 1 0|
```

```
--                                     |1 1 1 1 1 1 1|
```

```
-- En Haskell, las matrices anteriores se definen por
```

```
-- p1, p2, p3 :: Matriz Int
```

```
-- p1 = listArray ((1,1),(1,1)) [1]
```

```
-- p2 = listArray ((1,1),(2,3)) [0,1,0,
```

```
--                               1,1,1]
```

```
-- p3 = listArray ((1,1),(3,5)) [0,0,1,0,0,
```

```
--                               0,1,1,1,0,
```

```
--                               1,1,1,1,1]
```

```
-- Definir la función
```

```
-- esPiramidal :: (Eq a, Num a) => Matriz a -> Bool
```

```
-- tal que (esPiramidal p) se verifica si la matriz p es piramidal. Por
```

```
-- ejemplo,
```

```
-- esPiramidal p3 == True
```

```
-- esPiramidal (listArray ((1,1),(2,3)) [0,1,0, 1,5,1]) == False
```

```
-- esPiramidal (listArray ((1,1),(2,3)) [0,1,1, 1,1,1]) == False
```

```
-- esPiramidal (listArray ((1,1),(2,3)) [0,1,0, 1,0,1]) == False
```

```
-----
```

```

p1, p2, p3 :: Matriz Int
p1 = listArray ((1,1),(1,1)) [1]
p2 = listArray ((1,1),(2,3)) [0,1,0,
                              1,1,1]
p3 = listArray ((1,1),(3,5)) [0,0,1,0,0,
                              0,1,1,1,0,
                              1,1,1,1,1]

esPiramidal :: (Eq a, Num a) => Matriz a -> Bool
esPiramidal p =
  p == listArray ((1,1),(n,m)) (concat (filasPiramidal n))
  where (_,(n,m)) = bounds p

-- (filasPiramidal n) es la lista de las filas de la matriz piramidal de n
-- filas. Por ejemplo,
--   filasPiramidal 1 == [[1]]
--   filasPiramidal 2 == [[0,1,0],[1,1,1]]
--   filasPiramidal 3 == [[0,0,1,0,0],[0,1,1,1,0],[1,1,1,1,1]]
filasPiramidal 1 = [[1]]
filasPiramidal n = [0:xs++[0] | xs <- filasPiramidal (n-1)] ++
  [replicate (2*n-1) 1]

-- 2ª definición
-- =====

esPiramidal2 :: (Eq a, Num a) => Matriz a -> Bool
esPiramidal2 p =
  p == piramidal n
  where (_,(n,_)) = bounds p

-- (piramidal n) es la matriz piramidal con n filas. Por ejemplo,
--   ghci> piramidal 3
--   array ((1,1),(3,5)) [((1,1),0),((1,2),0),((1,3),1),((1,4),0),((1,5),0),
--                        ((2,1),0),((2,2),1),((2,3),1),((2,4),1),((2,5),0),
--                        ((3,1),1),((3,2),1),((3,3),1),((3,4),1),((3,5),1)]
piramidal :: (Eq a, Num a) => Int -> Matriz a
piramidal n =
  array ((1,1),(n,2*n-1)) [(i,j),f i j | i <- [1..n], j <- [1..2*n-1]]
  where f i j | j <= n-i = 0

```



```

-- Definir la función
--   iteracion_jacobi :: Matriz Float -> Matriz Float
-- tal que (iteracion_jacobi p) es la matriz obtenida aplicándole una
-- transformación de Jacobi a la matriz p. Por ejemplo,
--   iteracion_jacobi matriz1 == matriz2
-----

matriz1 :: Matriz Float
matriz1 = listArray ((1,1),(5,5)) ([2, 2, 2, 2, 2,
                                   2, 0, 0, 0, 2,
                                   2, 0, 0, 0, 2,
                                   2, 0, 0, 0, 2,
                                   2, 2, 2, 2, 2])

matriz2 :: Matriz Float
matriz2 = listArray ((1,1),(5,5)) ([2.0, 2.0, 2.0, 2.0, 2.0,
                                   2.0, 0.8, 0.4, 0.8, 2.0,
                                   2.0, 0.4, 0.0, 0.4, 2.0,
                                   2.0, 0.8, 0.4, 0.8, 2.0,
                                   2.0, 2.0, 2.0, 2.0, 2.0])

-- 1ª definición:
iteracion_jacobi :: Matriz Float -> Matriz Float
iteracion_jacobi p = array ((1,1),(n,m)) [((i,j), f i j) | i <- [1..n], j <- [1..m]]
  where (_,(n,m)) = bounds p
        f i j | frontera (i,j) = p!(i,j)
              | otherwise      = 0.2*(p!(i,j)+p!(i+1,j)+p!(i-1,j)+p!(i,j+1)+p!(i,j-1))
        frontera (i,j) = i == 1 || i == n || j == 1 || j == m

-- 2ª definición:
iteracion_jacobi2 :: Matriz Float -> Matriz Float
iteracion_jacobi2 p =
  array ((1,1),(n,m))
    ([((i,j), 0.2*(p!(i,j)+p!(i+1,j)+p!(i-1,j)+p!(i,j+1)+p!(i,j-1))) |
     i <- [2..n-1], j <- [2..m-1]] ++
     [((i,j),p!(i,j)) | i <- [1,n], j <- [1..m]] ++
     [((i,j),p!(i,j)) | i <- [1..n], j <- [1,m]])
  where (_,(n,m)) = bounds p

```

```

-- Ejercicio 26.1. Una matriz tridiagonal es aquella en la que sólo hay
-- elementos distintos de 0 en la diagonal principal o en las diagonales
-- por encima y por debajo de la diagonal principal. Por ejemplo,
--   ( 1 2 0 0 0 0 )
--   ( 3 4 5 0 0 0 )
--   ( 0 6 7 8 0 0 )
--   ( 0 0 9 1 2 0 )
--   ( 0 0 0 3 4 5 )
--   ( 0 0 0 0 6 7 )
--
-- Definir la función
--   creaTridiagonal :: Int -> Matriz Int
-- tal que (creaTridiagonal n) es la siguiente matriz tridiagonal
-- cuadrada con n filas y n columnas:
--   ( 1 1 0 0 0 0 ... 0 0 )
--   ( 1 2 2 0 0 0 ... 0 0 )
--   ( 0 2 3 3 0 0 ... 0 0 )
--   ( 0 0 3 4 4 0 ... 0 0 )
--   ( 0 0 0 4 5 5 ... 0 0 )
--   ( 0 0 0 0 5 6 ... 0 0 )
--   ( ..... )
--   ( 0 0 0 0 0 0 ... n n )
--   ( 0 0 0 0 0 0 ... n n+1 )
-- Por ejemplo,
--   ghci> creaTridiagonal 4
--   array ((1,1),(4,4)) [((1,1),1),((1,2),1),((1,3),0),((1,4),0),
--                        ((2,1),1),((2,2),2),((2,3),2),((2,4),0),
--                        ((3,1),0),((3,2),2),((3,3),3),((3,4),3),
--                        ((4,1),0),((4,2),0),((4,3),3),((4,4),4)]
--
-----

```

```

creaTridiagonal :: Int -> Matriz Int
creaTridiagonal n =
  array ((1,1),(n,n))
    [((i,j),valores i j) | i <- [1..n], j <- [1..n]]
  where valores i j | i == j      = i
                   | i == j+1    = j
                   | i+1 == j    = i
                   | otherwise   = 0

```

```

-----
-- Ejercicio 26.2. Definir la función
--   esTridiagonal :: Matriz Int -> Bool
-- tal que (esTridiagonal p) se verifica si la matriz p es tridiagonal. Por
-- ejemplo,
--   esTridiagonal (creaTridiagonal 5)           == True
--   esTridiagonal (listArray ((1,1),(3,3)) [1..9]) == False
-----

```

```

esTridiagonal :: Matriz Int -> Bool
esTridiagonal p =
  and [p!(i,j) == 0 | i <- [1..m], j <- [1..n], (j < i-1 || j > i+1)]
  where (_, (m,n)) = bounds p

```

```

-----
-- Ejercicio 27. La matriz de Vandermonde generada por
-- [a(1),a(2),a(3),...,a(n)] es la siguiente
--   |1 a(1) a(1)^2 ... a(1)^{n-1}|
--   |1 a(2) a(2)^2 ... a(2)^{n-1}|
--   |1 a(3) a(3)^2 ... a(3)^{n-1}|
--   |. . . . .|
--   |. . . . .|
--   |. . . . .|
--   |1 a(n) a(n)^2 ... a(n)^{n-1}|
--
-- Definir la función
--   vandermonde :: [Integer] -> Matriz Integer
-- tal que (vandermonde xs) es la matriz de Vandermonde cuyos
-- generadores son los elementos de xs. Por ejemplo,
--   ghci> vandermonde [5,2,3,4]
--   array ((1,1),(4,4)) [((1,1),1),((1,2),5),((1,3),25),((1,4),125),
--                         ((2,1),1),((2,2),2),((2,3), 4),((2,4), 8),
--                         ((3,1),1),((3,2),3),((3,3), 9),((3,4), 27),
--                         ((4,1),1),((4,2),4),((4,3),16),((4,4), 64)]
-----

```

```

-- 1ª solución
-- =====

```

```

vandermonde1 :: [Integer] -> Matriz Integer

```

```

vandermonde1 xs = array ((1,1), (n,n))
                  [((i,j), f i j) | i <- [1..n], j <- [1..n]]
  where n         = length xs
        f i j    = (xs!!(i-1))^(j-1)

-- 2ª solución
-- =====

vandermonde2 :: [Integer] -> Matriz Integer
vandermonde2 xs = listArray ((1,1),(n,n)) (concat (listaVandermonde xs))
  where n = length xs

-- (listaVandermonde xs) es la lista correspondiente a la matriz de
-- Vandermonde generada por xs. Por ejemplo,
-- ghci> listaVandermonde [5,2,3,4]
-- [[1,5,25,125],[1,2,4,8],[1,3,9,27],[1,4,16,64]]
listaVandermonde :: [Integer] -> [[Integer]]
listaVandermonde xs = [[x^i | i <- [0..n-1]] | x <- xs]
  where n = length xs

-----
-- Ejercicio 28. Una matriz es monomial si en cada una de sus filas y
-- columnas todos los elementos son nulos excepto 1. Por ejemplo, de las
-- matrices
--   |0 0 3 0|   |0 0 3 0|
--   |0 -2 0 0|   |0 -2 0 0|
--   |1 0 0 0|   |1 0 0 0|
--   |0 0 0 1|   |0 1 0 1|
-- la primera es monomial y la segunda no lo es.
--
-- En Haskell, las matrices anteriores se definen por
--   ej1, ej2 :: Matriz Int
--   ej1 = listArray ((1,1),(4,4)) [0, 0, 3, 0,
--                                   0, -2, 0, 0,
--                                   1, 0, 0, 0,
--                                   0, 0, 0, 1]
--   ej2 = listArray ((1,1),(4,4)) [0, 0, 3, 0,
--                                   0, -2, 0, 0,
--                                   1, 0, 0, 0,
--                                   0, 1, 0, 1]

```

```

-- Definir la función
--   esMonomial :: Matriz Int -> Bool
-- tal que (esMonomial p) se verifica si la matriz p es monomial. Por
-- ejemplo,
--   esMonomial ej1 == True
--   esMonomial ej2 == False
-----

ej1, ej2 :: Matriz Int
ej1 = listArray ((1,1),(4,4)) [0, 0, 3, 0,
                               0, -2, 0, 0,
                               1, 0, 0, 0,
                               0, 0, 0, 1]
ej2 = listArray ((1,1),(4,4)) [0, 0, 3, 0,
                               0, -2, 0, 0,
                               1, 0, 0, 0,
                               0, 1, 0, 1]

esMonomial :: Matriz Int -> Bool
esMonomial p = all esListaMonomial (filas ++ columnas)
  where filas      = [[p!(i,j) | j <- [1..n]] | i <- [1..m]]
        columnas  = [[p!(i,j) | i <- [1..m]] | j <- [1..n]]
        (_, (m,n)) = bounds p

-- (esListaMonomial xs) se verifica si todos los elementos de xs excepto
-- uno son nulos. Por ejemplo,
--   esListaMonomial [0,3,0,0] == True
--   esListaMonomial [0,3,0,2] == False
--   esListaMonomial [0,0,0,0] == False
esListaMonomial :: [Int] -> Bool
esListaMonomial xs = length (filter (/=0) xs) == 1
-----

-- Ejercicio 29. 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.
--
-- La matriz de Pascal es la matriz cuyas filas son los elementos de la
-- correspondiente fila del triángulo de Pascal completadas con
-- ceros. Por ejemplo, la matriz de Pascal de orden 6 es
--   |1 0 0 0 0 0|
--   |1 1 0 0 0 0|
--   |1 2 1 0 0 0|
--   |1 3 3 1 0 0|
--   |1 4 6 4 1 0|
--   |1 5 10 10 5 1|
--
-- Definir la función
--   matrizPascal :: Int -> Matriz Int
-- tal que (matrizPascal n) es la matriz de Pascal de orden n. Por
-- ejemplo,
--   ghci> matrizPascal 5
--   array ((1,1),(5,5))
--         [((1,1),1),((1,2),0),((1,3),0),((1,4),0),((1,5),0),
--          ((2,1),1),((2,2),1),((2,3),0),((2,4),0),((2,5),0),
--          ((3,1),1),((3,2),2),((3,3),1),((3,4),0),((3,5),0),
--          ((4,1),1),((4,2),3),((4,3),3),((4,4),1),((4,5),0),
--          ((5,1),1),((5,2),4),((5,3),6),((5,4),4),((5,5),1)]
-- -----
-- 1ª solución
-- =====
matrizPascal1 :: Int -> Matriz Int
matrizPascal1 1 = array ((1,1),(1,1)) [((1,1),1)]
matrizPascal1 n =
  array ((1,1),(n,n)) [((i,j), f i j) | i <- [1..n], j <- [1..n]]
  where f i j | i < n && j < n = p!(i,j)
              | i < n && j == n = 0
              | j == 1 || j == n = 1

```

```

        | otherwise          = p!(i-1,j-1) + p!(i-1,j)
    p = matrizPascal2 (n-1)

-- 2ª solución
-- =====

matrizPascal2 :: Int -> Matriz Int
matrizPascal2 n = listArray ((1,1),(n,n)) (concat xss)
    where yss = take n pascal
          xss = map (take n) (map (++ (repeat 0)) yss)

pascal :: [[Int]]
pascal = [1] : map f pascal
    where f xs = zipWith (+) (0:xs) (xs++[0])

-- 3ª solución
-- =====

matrizPascal3 :: Int -> Matriz Int
matrizPascal3 n =
    array ((1,1),(n,n)) [((i,j), f i j) | i <- [1..n], j <- [1..n]]
    where f i j | i >= j    = comb (i-1) (j-1)
                | otherwise = 0

-- (comb n k) es el número de combinaciones (o coeficiente binomial) de
-- n sobre k. Por ejemplo,
comb :: Int -> Int -> Int
comb n k = product [n,n-1..n-k+1] 'div' product [1..k]

-----
-- Ejercicio 30. Para cada número n la matriz completa de orden n es la
-- matriz cuadrada de orden n formada por los números enteros
-- consecutivos. Por ejemplo, la matriz completa de orden 3 es
--   | 1 2 3 |
--   | 4 5 6 |
--   | 7 8 9 |
-- las ternas primas de orden n son los listas formadas por un
-- elemento de la matriz junto con dos de sus vecinos de manera que los
-- tres son primos. Por ejemplo, en la matriz anterior una terna prima
-- es [2,3,5] (formada por el elemento 2, su vecino derecho 3 y su

```

```

-- vecino inferior 5), otra es [5,2,7] (formada por el elemento 5, su
-- vecino superior 2 y su vecino inferior-izquierda 7) y otra es [5,3,7]
-- (formada por el elemento 5, su vecino superior-derecha 3 y su
-- vecino inferior-izquierda 7).
--
-- Definir la función
--   ternasPrimasOrden :: Int -> [[Int]]
-- tal que (ternasPrimasOrden n) es el conjunto de las ternas primas de
-- la matriz completa de orden n. Por ejemplo,
--   ghci> ternasPrimasOrden 3
--   [[2,3,5],[3,2,5],[5,2,3],[5,2,7],[5,3,7]]
--   ghci> ternasPrimasOrden 4
--   [[2,3,5],[2,3,7],[2,5,7],[3,2,7],[7,2,3],[7,2,11],[7,3,11]]
-----
ternasPrimasOrden :: Int -> [[Int]]
ternasPrimasOrden = ternasPrimas . matrizCompleta

-- (ternasPrimas p) es la lista de las ternas primas de p. Por ejemplo,
--   ghci> ternasPrimas (listArray ((1,1),(3,3)) [2,3,7,5,4,1,6,8,9])
--   [[2,3,5],[3,2,7],[3,2,5],[3,7,5],[5,2,3]]
ternasPrimas :: Matriz Int -> [[Int]]
ternasPrimas p =
  [xs | xs <- ternas p, all esPrimo xs]

-- (ternas p) es la lista de las ternas de p formadas por un elemento de
-- p junto con dos vecinos. Por ejemplo,
--   ghci> ternas (listArray ((1,1),(3,3)) [2,3,7,5,4,0,6,8,9])
--   [[2,3,5],[2,3,4],[2,5,4],[3,2,7],[3,2,5],[3,2,4],[3,2,0],[3,7,5],
--    [3,7,4],[3,7,0],[3,5,4],[3,5,0],[3,4,0],[7,3,4],[7,3,0],[7,4,0],
--    [5,2,3],[5,2,4],[5,2,6],[5,2,8],[5,3,4],[5,3,6],[5,3,8],[5,4,6],
--    [5,4,8],[5,6,8],[4,2,3],[4,2,7],[4,2,5],[4,2,0],[4,2,6],[4,2,8],
--    [4,2,9],[4,3,7],[4,3,5],[4,3,0],[4,3,6],[4,3,8],[4,3,9],[4,7,5],
--    [4,7,0],[4,7,6],[4,7,8],[4,7,9],[4,5,0],[4,5,6],[4,5,8],[4,5,9],
--    [4,0,6],[4,0,8],[4,0,9],[4,6,8],[4,6,9],[4,8,9],[0,3,7],[0,3,4],
--    [0,3,8],[0,3,9],[0,7,4],[0,7,8],[0,7,9],[0,4,8],[0,4,9],[0,8,9],
--    [6,5,4],[6,5,8],[6,4,8],[8,5,4],[8,5,0],[8,5,6],[8,5,9],[8,4,0],
--    [8,4,6],[8,4,9],[8,0,6],[8,0,9],[8,6,9],[9,4,0],[9,4,8],[9,0,8]]
ternas :: Matriz Int -> [[Int]]
ternas p =

```



```

[[p!(i1,j1),p!(i2,j2),p!(i3,j3)] |
 (i1,j1) <- indices p,
 ((i2,j2):ps) <- tails (vecinos (i1,j1) n),
 (i3,j3) <- ps]
where (_,(n,_)) = bounds p

-- (vecinos (i,j) n) es la lista de las posiciones vecinas de la (i,j)
-- en una matriz cuadrada de orden n. Por ejemplo,
--   vecinos (2,3) 4 == [(1,2),(1,3),(1,4),(2,2),(2,4),(3,2),(3,3),(3,4)]
--   vecinos (2,4) 4 == [(1,3),(1,4),(2,3),(3,3),(3,4)]
--   vecinos (1,4) 4 == [(1,3),(2,3),(2,4)]
vecinos :: (Int,Int) -> Int -> [(Int,Int)]
vecinos (i,j) n = [(a,b) | a <- [max 1 (i-1)..min n (i+1)],
                          b <- [max 1 (j-1)..min n (j+1)],
                          (a,b) /= (i,j)]

-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--   esPrimo 7 == True
--   esPrimo 15 == False
esPrimo :: Int -> Bool
esPrimo n = [x | x <- [1..n], n `rem` x == 0] == [1,n]

-- (matrizCompleta n) es la matriz completa de orden n. Por ejemplo,
--   ghci> matrizCompleta 3
--   array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
--                        ((2,1),4),((2,2),5),((2,3),6),
--                        ((3,1),7),((3,2),8),((3,3),9)]
matrizCompleta :: Int -> Matriz Int
matrizCompleta n =
  listArray ((1,1),(n,n)) [1..n*n]

-- 2ª definición
-- =====

ternasPrimasOrden2 :: Int -> [[Int]]
ternasPrimasOrden2 = ternasPrimas2 . matrizCompleta

ternasPrimas2 :: Matriz Int -> [[Int]]
ternasPrimas2 p =
  [[p!(i1,j1),p!(i2,j2),p!(i3,j3)] |

```

```
(i1,j1) <- indices p,  
esPrimo (p!(i1,j1)),  
((i2,j2):ps) <- tails (vecinos (i1,j1) n),  
esPrimo (p!(i2,j2)),  
(i3,j3) <- ps,  
esPrimo (p!(i3,j3))]  
where (_,(n,_)) = bounds p  
  
-- Comparación:  
-- ghci> length (ternasPrimasOrden 30)  
-- 51  
-- (5.52 secs, 211095116 bytes)  
-- ghci> length (ternasPrimasOrden2 30)  
-- 51  
-- (0.46 secs, 18091148 bytes)
```

Relación 16

Cálculo numérico

```
-----  
-- Introducción --  
-----  
  
-- En esta relación se definen funciones para resolver los siguientes  
-- problemas de cálculo numérico:  
-- * diferenciación numérica,  
-- * cálculo de la raíz cuadrada mediante el método de Herón,  
-- * cálculo de los ceros de una función por el método de Newton y  
-- * cálculo de funciones inversas.  
  
-----  
-- Importación de librerías --  
-----  
  
import Test.QuickCheck  
  
-----  
-- Diferenciación numérica --  
-----  
  
-----  
-- Ejercicio 1.1. Definir la función  
-- derivada :: Double -> (Double -> Double) -> Double -> Double  
-- tal que (derivada a f x) es el valor de la derivada de la función f  
-- en el punto x con aproximación a. Por ejemplo,  
-- derivada 0.001 sin pi == -0.9999998333332315
```

```

--   derivada 0.001 cos pi == 4.999999583255033e-4
-----

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

-----

-- Ejercicio 1.2. Definir las funciones
--   derivadaBurda :: (Double -> Double) -> Double -> Double
--   derivadaFina  :: (Double -> Double) -> Double -> Double
--   derivadaSuper :: (Double -> Double) -> Double -> Double
-- tales que
--   * (derivadaBurda f x) es el valor de la derivada de la función f
--     en el punto x con aproximación 0.01,
--   * (derivadaFina f x) es el valor de la derivada de la función f
--     en el punto x con aproximación 0.0001.
--   * (derivadaSuper f x) es el valor de la derivada de la función f
--     en el punto x con aproximación 0.000001.
-- Por ejemplo,
--   derivadaBurda cos pi == 4.999958333473664e-3
--   derivadaFina  cos pi == 4.999999969612645e-5
--   derivadaSuper cos pi == 5.000444502911705e-7
-----

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

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

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

-----

-- Ejercicio 1.3. Definir la función
--   derivadaFinaDelSeno :: Double -> Double
-- tal que (derivadaFinaDelSeno x) es el valor de la derivada fina del
-- seno en x. Por ejemplo,
--   derivadaFinaDelSeno pi == -0.9999999983354436
-----

```

```
derivadaFinaDelSeno :: Double -> Double
```

```
derivadaFinaDelSeno = derivadaFina sin
```

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

```
-----
-- Ejercicio 2.1. En los siguientes apartados de este ejercicio se va a
-- calcular la raíz cuadrada de un número basándose en las siguientes
-- propiedades:
```

```
-- * Si  $y$  es una aproximación de la raíz cuadrada de  $x$ , entonces
```

```
--  $(y+x/y)/2$  es una aproximación mejor.
```

```
-- * El límite de la sucesión definida por
```

```
--  $x_0 = 1$ 
```

```
--  $x_{n+1} = (x_n + x/x_n)/2$ 
```

```
-- es la raíz cuadrada de  $x$ .
```

```
--
```

```
-- Definir, por recursión, la función
```

```
-- raiz :: Double -> Double
```

```
-- tal que (raiz x) es la raíz cuadrada de  $x$  calculada usando la
```

```
-- propiedad anterior con una aproximación de  $0.00001$  y tomando como
```

```
--  $v$ . Por ejemplo,
```

```
-- raiz 9 == 3.000000001396984
```

```
-----
```

```
raiz :: Double -> Double
```

```
raiz x = raiz' 1
```

```
  where raiz' y | acceptable y = y
```

```
          | otherwise = raiz' (mejora y)
```

```
    mejora y = 0.5*(y+x/y)
```

```
    acceptable y = abs(y*y-x) < 0.00001
```

```
-----
```

```
-- Ejercicio 3.2. Definir el operador
```

```
-- (~=) :: Double -> Double -> Bool
```

```
-- tal que (x ~= y) si  $|x-y| < 0.001$ . Por ejemplo,
```

```
-- 3.05 ~= 3.07 == False
```

```
-- 3.00005 ~= 3.00007 == True
```

```

-----
infix 5 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs(x-y) < 0.001

```

```

-----
-- Ejercicio 3.3. Comprobar con QuickCheck que si x es positivo,
-- entonces
--   (raiz x)^2 ~= x
-----

```

```

-- La propiedad es
prop_raiz :: Double -> Bool
prop_raiz x =
  (raiz x')^2 ~= x'
  where x' = abs x

```

```

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

```

```

-----
-- Ejercicio 3.4. Definir por recursión la función
--   until' :: (a -> Bool) -> (a -> a) -> a -> a
-- tal que (until' p f x) es el resultado de aplicar la función f a x el
-- menor número posible de veces, hasta alcanzar un valor que satisface
-- el predicado p. Por ejemplo,
--   until' (>1000) (2*) 1 == 1024
-- Nota: until' es equivalente a la predefinida until.
-----

```

```

until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f x | p x      = x
              | otherwise = until' p f (f x)

```

```

-----
-- Ejercicio 3.5. Definir, por iteración con until, la función
--   raizI :: Double -> Double
-- tal que (raizI x) es la raíz cuadrada de x calculada usando la

```

```

-- propiedad anterior. Por ejemplo,
--   raizI 9 == 3.000000001396984
-----

raizI :: Double -> Double
raizI x = until aceptable mejora 1
  where mejora y      = 0.5*(y+x/y)
        aceptable y = abs(y*y-x) < 0.00001
-----

-- Ejercicio 3.6. Comprobar con QuickCheck que si x es positivo,
-- entonces
--   (raizI x)^2 ~= x
-----

-- La propiedad es
prop_raizI :: Double -> Bool
prop_raizI x =
  (raizI x')^2 ~= x'
  where x' = abs x

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

-- Ceros de una función
-----

-- Ejercicio 4. Los ceros de una función pueden calcularse mediante el
-- método de Newton basándose en las siguientes propiedades:
-- * Si b es una aproximación para el punto cero de f, entonces
--    $b - f(b)/f'(b)$  es una mejor aproximación.
-- * el límite de la sucesión  $x_n$  definida por
--    $x_0 = 1$ 
--    $x_{n+1} = x_n - f(x_n)/f'(x_n)$ 
--   es un cero de f.
-----

```

```

-----
-- Ejercicio 4.1. Definir, por recursión, la función
-- puntoCero :: (Double -> Double) -> Double
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
-- puntoCero cos == 1.5707963267949576
-----

puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCero' f 1
  where puntoCero' f x | acceptable x = x
                    | otherwise     = puntoCero' f (mejora x)
        mejora b      = b - f b / derivadaFina f b
        acceptable b  = abs (f b) < 0.00001

-----
-- Ejercicio 4.2. Definir, por iteración con until, la función
-- puntoCeroI :: (Double -> Double) -> Double
-- tal que (puntoCeroI f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
-- puntoCeroI cos == 1.5707963267949576
-----

puntoCeroI :: (Double -> Double) -> Double
puntoCeroI f = until acceptable mejora 1
  where mejora b      = b - f b / derivadaFina f b
        acceptable b  = abs (f b) < 0.00001

-----
-- Funciones inversas
-----

-----
-- Ejercicio 5. En este ejercicio se usará la función puntoCero para
-- definir la inversa de distintas funciones.
-----

-----
-- Ejercicio 5.1. Definir, usando puntoCero, la función
-- raizCuadrada :: Double -> Double

```



```
-- tal que (raizCuadrada x) es la raíz cuadrada de x. Por ejemplo,  
--   raizCuadrada 9 == 3.000000002941184  
-----
```

```
raizCuadrada :: Double -> Double  
raizCuadrada a = puntoCero f  
  where f x = x*x-a  
-----
```

```
-- Ejercicio 5.2. Comprobar con QuickCheck que si x es positivo,  
-- entonces  
--   (raizCuadrada x)^2 ~= x  
-----
```

```
-- La propiedad es  
prop_raizCuadrada :: Double -> Bool  
prop_raizCuadrada x =  
  (raizCuadrada x')^2 ~= x'  
  where x' = abs x  
-----
```

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

```
-- Ejercicio 5.3. Definir, usando puntoCero, la función  
--   raizCubica :: Double -> Double  
-- tal que (raizCubica x) es la raíz cuadrada de x. Por ejemplo,  
--   raizCubica 27 == 3.0000000000196048  
-----
```

```
raizCubica :: Double -> Double  
raizCubica a = puntoCero f  
  where f x = x*x*x-a  
-----
```

```
-- Ejercicio 5.4. Comprobar con QuickCheck que si x es positivo,  
-- entonces  
--   (raizCubica x)^3 ~= x  
-----
```

```
-- La propiedad es
prop_raizCubica :: Double -> Bool
prop_raizCubica x =
  (raizCubica x)^3 == x
  where x' = abs x
```

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

```
-----
-- Ejercicio 5.5. Definir, usando puntoCero, la función
--   arcoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--   arcoseno 1 == 1.5665489428306574
-----
```

```
arcoseno :: Double -> Double
arcoseno a = puntoCero f
  where f x = sin x - a
```

```
-----
-- Ejercicio 5.6. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--   sin (arcoseno x) == x
-----
```

```
-- La propiedad es
prop_arcoseno :: Double -> Bool
prop_arcoseno x =
  sin (arcoseno x) == x
  where x' = abs (x - fromIntegral (truncate x))
```

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

```
-- Otra forma de expresar la propiedad es
prop_arcoseno2 :: Property
```

```
prop_arcoseno2 = forAll (choose (0,1)) $ \x -> sin (arcoseno x) ~= x
```

```
-- La comprobación es
-- ghci> quickCheck prop_arcoseno2
-- OK, passed 100 tests.
```

```
-----
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--   arcocoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--   arcocoseno 0 == 1.5707963267949576
-----
```

```
arcocoseno :: Double -> Double
arcocoseno a = puntoCero f
  where f x = cos x - a
```

```
-----
-- Ejercicio 5.8. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--   cos (arcocoseno x) ~= x
-----
```

```
-- La propiedad es
prop_arcocoseno :: Double -> Bool
prop_arcocoseno x =
  cos (arcocoseno x') ~= x'
  where x' = abs (x - fromIntegral (truncate x))
```

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

```
-- Otra forma de expresar la propiedad es
```

```
prop_arcocoseno2 :: Property
prop_arcocoseno2 = forAll (choose (0,1)) $ \x -> cos (arcocoseno x) ~= x
```

```
-- La comprobación es
-- ghci> quickCheck prop_arcocoseno2
-- OK, passed 100 tests.
```

```
-----  
-- Ejercicio 5.7. Definir, usando puntoCero, la función  
-- inversa :: (Double -> Double) -> Double -> Double  
-- tal que (inversa g x) es el valor de la inversa de g en x. Por  
-- ejemplo,  
-- inversa (^2) 9 == 3.0000000002941184  
-----
```

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

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

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

Relación 17

Operaciones con el TAD de polinomios

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación es ampliar el conjunto de operaciones  
-- sobre polinomios definidas utilizando las implementaciones del TAD de  
-- polinomio estudiadas en el tema 21 que se pueden descargar desde  
-- http://www.cs.us.es/~jalonso/cursos/ilm/codigos/I1M2014.zip  
-- Además, en algunos ejemplos de usan polinomios con coeficientes  
-- racionales. En Haskell, el número racional  $x/y$  se representa por  
--  $x\%y$ . El TAD de los números racionales está definido en el módulo  
-- Data.Ratio.  
--  
-- Las transparencias del tema 21 se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-21.pdf  
  
-----  
-- Importación de librerías --  
-----  
  
import I1M.PolOperaciones -- Se instala con http://bit.ly/1AKmUQB  
import Test.QuickCheck  
import Data.Ratio  
  
-----
```

```
-- Ejercicio 1. Definir la función
--   creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
--   tal que (creaPolDispersa xs) es el polinomio cuya representación
--   dispersa es xs. Por ejemplo,
--   creaPolDispersa [7,0,0,4,0,3] == 7*x^5 + 4*x^2 + 3
```

```
creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
creaPolDispersa [] = polCero
creaPolDispersa (x:xs) = consPol (length xs) x (creaPolDispersa xs)
```

```
-- Ejercicio 2. Definir la función
--   creaPolDensa :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
--   tal que (creaPolDensa xs) es el polinomio cuya representación
--   densa es xs. Por ejemplo,
--   creaPolDensa [(5,7),(4,2),(3,0)] == 7*x^5 + 2*x^4
```

```
creaPolDensa :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
creaPolDensa [] = polCero
creaPolDensa ((n,a):ps) = consPol n a (creaPolDensa ps)
```

```
-- Nota. En el resto de la sucesión se usará en los ejemplos los
-- los polinomios que se definen a continuación.
```

```
pol1, pol2, pol3 :: (Num a, Eq a) => Polinomio a
pol1 = creaPolDensa [(5,1),(2,5),(1,4)]
pol2 = creaPolDispersa [2,3]
pol3 = creaPolDensa [(7,2),(4,5),(2,5)]
```

```
pol4, pol5, pol6 :: Polinomio Rational
pol4 = creaPolDensa [(4,3),(2,5),(0,3)]
pol5 = creaPolDensa [(2,6),(1,2)]
pol6 = creaPolDensa [(2,8),(1,14),(0,3)]
```

```
-- Ejercicio 3. Definir la función
```

```
-- densa :: (Num a, Eq a) => Polinomio a -> [(Int,a)]
-- tal que (densa p) es la representación densa del polinomio p. Por
-- ejemplo,
-- pol1      == x^5 + 5*x^2 + 4*x
-- densa pol1 == [(5,1),(2,5),(1,4)]
```

```
densa :: (Num a, Eq a) => Polinomio a -> [(Int,a)]
densa p | esPolCero p = []
        | otherwise   = (grado p, coefLider p) : densa (restoPol p)
```

```
-- Ejercicio 4. Definir la función
-- densaAdispersa :: Num a => [(Int,a)] -> [a]
-- tal que (densaAdispersa ps) es la representación dispersa del
-- polinomio cuya representación densa es ps. Por ejemplo,
-- densaAdispersa [(5,1),(2,5),(1,4)] == [1,0,0,5,4,0]
```

```
densaAdispersa :: Num a => [(Int,a)] -> [a]
densaAdispersa [] = []
densaAdispersa [(n,a)] = a : replicate n 0
densaAdispersa ((n,a):(m,b):ps) =
  a : (replicate (n-m-1) 0) ++ densaAdispersa ((m,b):ps)
```

```
-- Ejercicio 5. Definir la función
-- dispersa :: (Num a, Eq a) => Polinomio a -> [a]
-- tal que (dispersa p) es la representación dispersa del polinomio
-- p. Por ejemplo,
-- pol1      == x^5 + 5*x^2 + 4*x
-- dispersa pol1 == [1,0,0,5,4,0]
```

```
dispersa :: (Num a, Eq a) => Polinomio a -> [a]
dispersa = densaAdispersa . densa
```

```
-- Ejercicio 6. Definir la función
-- coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
```

```
-- tal que (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--   poll                == x^5 + 5*x^2 + 4*x
--   coeficiente 2 poll == 5
--   coeficiente 3 poll == 0
```

```
-----
coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
coeficiente k p | k == n           = coefLider p
                | k > grado (restoPol p) = 0
                | otherwise        = coeficiente k (restoPol p)
                where n = grado p
```

```
-- Otra definición equivalente es
```

```
coeficiente' :: (Num a, Eq a) => Int -> Polinomio a -> a
coeficiente' k p = busca k (densa p)
  where busca k ps = head ([a | (n,a) <- ps, n == k] ++ [0])
```

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

```
--   coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de los coeficientes del
-- polinomio p. Por ejemplo,
--   poll                == x^5 + 5*x^2 + 4*x
--   coeficientes poll == [1,0,0,5,4,0]
```

```
-----
coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <- [n,n-1..0]]
  where n = grado p
```

```
-- Una definición equivalente es
```

```
coeficientes' :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes' = dispersa
```

```
-----
-- Ejercicio 8. Definir la función
```

```
--   potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potencia p n) es la potencia n-ésima del polinomio p. Por
-- ejemplo,
```



```
--      pol2                == 2*x + 3
--      potencia pol2 2    == 4*x^2 + 12*x + 9
--      potencia pol2 3    == 8*x^3 + 36*x^2 + 54*x + 27
```

```
-----
potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potencia p 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))
```

```
-----
-- Ejercicio 9. Mejorar la definición de potencia definiendo la función
-- potenciaM :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potenciaM p n) es la potencia n-ésima del polinomio p,
-- utilizando las siguientes propiedades:
```

```
-- * Si n es par, entonces  $x^n = (x^2)^{(n/2)}$ 
-- * Si n es impar, entonces  $x^n = x * (x^2)^{((n-1)/2)}$ 
```

```
-- Por ejemplo,
```

```
--      pol2                == 2*x + 3
--      potenciaM pol2 2    == 4*x^2 + 12*x + 9
--      potenciaM pol2 3    == 8*x^3 + 36*x^2 + 54*x + 27
```

```
-----
potenciaM :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potenciaM p 0 = polUnidad
potenciaM p n
  | even n    = potenciaM (multPol p p) (n `div` 2)
  | otherwise = multPol p (potenciaM (multPol p p) ((n-1) `div` 2))
```

```
-----
-- Ejercicio 10. Definir la función
```

```
-- integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
-- tal que (integral p) es la integral del polinomio p cuyos coeficientes
-- son números racionales. Por ejemplo,
```

```
-- ghci> pol3
--      2*x^7 + 5*x^4 + 5*x^2
-- ghci> integral pol3
--      0.25*x^8 + x^5 + 1.6666666666666667*x^3
-- ghci> integral pol3 :: Polinomio Rational
--      1 % 4*x^8 + x^5 + 5 % 3*x^3
```

```

integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
integral p
  | esPolCero p = polCero
  | otherwise   = consPol (n+1) (b / (fromIntegral (n+1))) (integral r)
  where n = grado p
        b = coefLider p
        r = restoPol p

```

```

-----
-- Ejercicio 11. Definir la función
--   integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
-- tal que (integralDef p a b) es la integral definida del polinomio p
-- cuyos coeficientes son números racionales. Por ejemplo,
--   ghci> integralDef pol3 0 1
--   2.916666666666667
--   ghci> integralDef pol3 0 1 :: Rational
--   35 % 12
-----

```

```

integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
integralDef p a b = (valor q b) - (valor q a)
  where q = integral p

```

```

-----
-- Ejercicio 12. Definir la función
--   multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
-- tal que (multEscalar c p) es el polinomio obtenido multiplicando el
-- número c por el polinomio p. Por ejemplo,
--   pol2 == 2*x + 3
--   multEscalar 4 pol2 == 8*x + 12
--   multEscalar (1%4) pol2 == 1 % 2*x + 3 % 4
-----

```

```

multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
multEscalar c p
  | esPolCero p = polCero
  | otherwise   = consPol n (c*b) (multEscalar c r)
  where n = grado p
        b = coefLider p

```

```
r = restoPol p
```

```
-----
-- Ejercicio 13. Definir la función
--   cociente:: (Fractional a, Eq a) =>
--               Polinomio a -> Polinomio a -> Polinomio a
-- tal que (cociente p q) es el cociente de la división de p entre
-- q. Por ejemplo,
--   pol4 == 3 % 1*x^4 + 5 % 1*x^2 + 3 % 1
--   pol5 == 6 % 1*x^2 + 2 % 1*x
--   cociente pol4 pol5 == 1 % 2*x^2 + (-1) % 6*x + 8 % 9
-----
```

```
cociente:: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
cociente p q
  | n2 == 0   = multEscalar (1/a2) p
  | n1 < n2   = polCero
  | otherwise = consPol n' a' (cociente p' q)
where n1 = grado p
      a1 = coefLider p
      n2 = grado q
      a2 = coefLider q
      n'  = n1-n2
      a'  = a1/a2
      p'  = restaPol p (multPorTerm (creaTermino n' a') q)
```

```
-----
-- Ejercicio 14. Definir la función
--   resto:: (Fractional a, Eq a) =>
--           Polinomio a -> Polinomio a -> Polinomio a
-- tal que (resto p q) es el resto de la división de p entre q. Por
-- ejemplo,
--   pol4 == 3 % 1*x^4 + 5 % 1*x^2 + 3 % 1
--   pol5 == 6 % 1*x^2 + 2 % 1*x
--   resto pol4 pol5 == (-16) % 9*x + 3 % 1
-----
```

```
resto :: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
resto p q = restaPol p (multPol (cociente p q) q)
```

```

-----
-- Ejercicio 15. Definir la función
--   divisiblePol :: (Fractional a, Eq a) =>
--                 Polinomio a -> Polinomio a -> Bool
-- tal que (divisiblePol p q) se verifica si el polinomio p es divisible
-- por el polinomio q. Por ejemplo,
--   pol6 == 8 % 1*x^2 + 14 % 1*x + 3 % 1
--   pol2 == 2*x + 3
--   pol5 == 6 % 1*x^2 + 2 % 1*x
--   divisiblePol pol6 pol2 == True
--   divisiblePol pol6 pol5 == False
-----

```

```

divisiblePol :: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Bool
divisiblePol p q = esPolCero (resto p q)

```

```

-----
-- Ejercicio 16. El método de Horner para calcular el valor de un
-- polinomio se basa en representarlo de una forma forma alernativa. Por
-- ejemplo, para calcular el valor de
--   a*x^5 + b*x^4 + c*x^3 + d*x^2 + e*x + f
-- se representa como
--   (((((0 * x + a) * x + b) * x + c) * x + d) * x + e) * x + f
-- y se evalúa de dentro hacia afuera; es decir,
--   v(0) = 0
--   v(1) = v(0)*x+a = 0*x+a = a
--   v(2) = v(1)*x+b = a*x+b
--   v(3) = v(2)*x+c = (a*x+b)*x+c = a*x^2+b*x+c
--   v(4) = v(3)*x+d = (a*x^2+b*x+c)*x+d = a*x^3+b*x^2+c*x+d
--   v(5) = v(4)*x+e = (a*x^3+b*x^2+c*x+d)*x+e = a*x^4+b*x^3+c*x^2+d*x+e
--   v(6) = v(5)*x+f = (a*x^4+b*x^3+c*x^2+d*x+e)*x+f = a*x^5+b*x^4+c*x^3+d*x^2+e*x+f
--
-- Definir la función
--   horner :: (Num a, Eq a) => Polinomio a -> a -> a
-- tal que (horner p x) es el valor del polinomio p al sustituir su
-- variable por el número x. Por ejemplo,
--   horner pol1 0 == 0
--   horner pol1 1 == 10
--   horner pol1 1.5 == 24.84375
--   horner pol1 (3%2) == 795 % 32
-----

```

```

horner :: (Num a, Eq a) => Polinomio a -> a -> a
horner p x = hornerAux (coeficientes p) 0
  where hornerAux [] v      = v
        hornerAux (a:as) v = hornerAux as (v*x+a)

-- El cálculo de (horner pol1 2) es el siguiente
-- horner pol1 2
-- = hornerAux [1,0,0,5,4,0] 0
-- = hornerAux [0,0,5,4,0] (0*2+1) = hornerAux [0,0,5,4,0] 1
-- = hornerAux [0,5,4,0] (1*2+0) = hornerAux [0,5,4,0] 2
-- = hornerAux [5,4,0] (2*2+0) = hornerAux [5,4,0] 4
-- = hornerAux [4,0] (4*2+5) = hornerAux [4,0] 13
-- = hornerAux [0] (13*2+4) = hornerAux [0] 30
-- = hornerAux [] (30*2+0) = hornerAux [] 60

-- Una definición equivalente por plegado es
horner' :: (Num a, Eq a) => Polinomio a -> a -> a
horner' p x = (foldr (\a b -> a + b*x) 0) (coeficientes p)

```


Relación 18

División y factorización de polinomios mediante la regla de Ruffini

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es implementar la regla de  
-- Ruffini y sus aplicaciones utilizando las implementaciones del TAD de  
-- polinomio estudiadas en el tema 21 que se pueden descargar desde  
-- http://www.cs.us.es/~jalonso/cursos/ilm/codigos/I1M2014.zip  
--  
-- Las transparencias del tema 21 se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-13/temas/tema-21.pdf  
  
-----  
-- Importación de librerías --  
-----  
  
import I1M.PolOperaciones -- Se encuentra en http://bit.ly/1AKmUQB  
import Test.QuickCheck  
  
-----  
-- Ejemplos --  
-----
```

-- Además de los ejemplos de polinomios (ejPol1, ejPol2 y ejPol3) que se encuentran en PolOperaciones, usaremos el siguiente ejemplo.

```
ejPol4 :: Polinomio Int
```

```
ejPol4 = consPol 3 1
          (consPol 2 2
            (consPol 1 (-1)
              (consPol 0 (-2) polCero)))
```

-- Ejercicio 1. Definir la función

```
-- divisores :: Int -> [Int]
```

-- tal que (divisores n) es la lista de todos los divisores enteros de n. Por ejemplo,

```
-- divisores 4 == [1,-1,2,-2,4,-4]
```

```
-- divisores (-6) == [1,-1,2,-2,3,-3,6,-6]
```

```
-----  
divisores :: Int -> [Int]
```

```
divisores n = concat [[x,-x] | x <- [1..abs n], rem n x == 0]
```

-- Ejercicio 2. Definir la función

```
-- coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
```

-- tal que (coeficiente k p) es el coeficiente del término de grado k en p. Por ejemplo:

```
-- coeficiente 4 ejPol1 == 3
```

```
-- coeficiente 3 ejPol1 == 0
```

```
-- coeficiente 2 ejPol1 == -5
```

```
-- coeficiente 5 ejPol1 == 0
```

```
-----  
coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
```

```
coeficiente k p | k == gp = coefLider p
```

```
                | k > grado rp = 0
```

```
                | otherwise = coeficiente k rp
```

```
  where gp = grado p
```

```
        rp = restoPol p
```

-- Ejercicio 3. Definir la función


```
-- terminoIndep :: (Num a, Eq a) => Polinomio a -> a
-- tal que (terminoIndep p) es el término independiente del polinomio
-- p. Por ejemplo,
-- terminoIndep ejPol1 == 3
-- terminoIndep ejPol2 == 0
-- terminoIndep ejPol4 == -2
```

```
terminoIndep :: (Num a, Eq a) => Polinomio a -> a
terminoIndep p = coeficiente 0 p
```

```
-- Ejercicio 4. Definir la función
-- coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de coeficientes de p, ordenada
-- según el grado. Por ejemplo,
-- coeficientes ejPol1 == [3,0,-5,0,3]
-- coeficientes ejPol4 == [1,2,-1,-2]
-- coeficientes ejPol2 == [1,0,0,5,4,0]
```

```
coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <- [n,n-1..0]]
  where n = grado p
```

```
-- Ejercicio 5. Definir la función
-- creaPol :: (Num a, Eq a) => [a] -> Polinomio a
-- tal que (creaPol cs) es el polinomio cuya lista de coeficientes es
-- cs. Por ejemplo,
-- creaPol [1,0,0,5,4,0] == x^5 + 5*x^2 + 4*x
-- creaPol [1,2,0,3,0] == x^4 + 2*x^3 + 3*x
```

```
creaPol :: (Num a, Eq a) => [a] -> Polinomio a
creaPol [] = polCero
creaPol (a:as) = consPol n a (creaPol as)
  where n = length as
```

```
-- Ejercicio 6. Comprobar con QuickCheck que, dado un polinomio p, el
-- polinomio obtenido mediante creaPol a partir de la lista de
-- coeficientes de p coincide con p.
```

```
-----
-- La propiedad es
prop_coef :: Polinomio Int -> Bool
prop_coef p =
    creaPol (coeficientes p) == p
```

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

```
-----
-- Ejercicio 7. Definir una función
-- pRuffini :: Int -> [Int] -> [Int]
-- tal que (pRuffini r cs) es la lista que resulta de aplicar un paso
-- del regla de Ruffini al número entero r y a la lista de coeficientes
-- cs. Por ejemplo,
-- pRuffini 2 [1,2,-1,-2] == [1,4,7,12]
-- pRuffini 1 [1,2,-1,-2] == [1,3,2,0]
-- ya que
```

```
--   | 1  2  -1  -2           | 1  2  -1  -2
-- 2 |   2  8  14           1 |   1  3  2
-- --+-----             --+-----
--   | 1  4  7  12           | 1  3  2  0
```

```
-----
pRuffini :: Int -> [Int] -> [Int]
pRuffini r p@(c:cs) =
    c : [x+r*y | (x,y) <- zip cs (pRuffini r p)]
```

```
-- Otra forma:
pRuffini' :: Int -> [Int] -> [Int]
pRuffini' r = scanl1 (\s x -> s * r + x)
```

```
-----
-- Ejercicio 8. Definir la función
-- cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
```

```
-- tal que (cocienteRuffini r p) es el cociente de dividir el polinomio
-- p por el polinomio x-r. Por ejemplo:
--   cocienteRuffini 2 ejPol4    == x^2 + 4*x + 7
--   cocienteRuffini (-2) ejPol4 == x^2 + -1
--   cocienteRuffini 3 ejPol4    == x^2 + 5*x + 14
-----
```

```
cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = creaPol (init (pRuffini r (coeficientes p)))
```

```
-- Ejercicio 9. Definir la función
--   restoRuffini:: Int -> Polinomio Int -> Int
-- tal que (restoRuffini r p) es el resto de dividir el polinomio p por
-- el polinomio x-r. Por ejemplo,
--   restoRuffini 2 ejPol4    == 12
--   restoRuffini (-2) ejPol4 == 0
--   restoRuffini 3 ejPol4    == 40
-----
```

```
restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (pRuffini r (coeficientes p))
```

```
-- Ejercicio 10. Comprobar con QuickCheck que, dado un polinomio p y un
-- número entero r, las funciones anteriores verifican la propiedad de
-- la división euclídea.
-----
```

```
-- La propiedad es
prop_diviEuclidea:: Int -> Polinomio Int -> Bool
prop_diviEuclidea r p =
  p == sumaPol (multPol coc div) res
  where coc = cocienteRuffini r p
        div = creaPol [1,-r]
        res = creaTermino 0 (restoRuffini r p)
```

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

```

-----
-- Ejercicio 11. Definir la función
--   esRaizRuffini :: Int -> Polinomio Int -> Bool
-- tal que (esRaizRuffini r p) se verifica si r es una raíz de p, usando
-- para ello el regla de Ruffini. Por ejemplo,
--   esRaizRuffini 0 ejPol3 == True
--   esRaizRuffini 1 ejPol3 == False
-----

```

```

esRaizRuffini :: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0

```

```

-----
-- Ejercicio 12. Definir la función
--   raicesRuffini :: Polinomio Int -> [Int]
-- tal que (raicesRuffini p) es la lista de las raíces enteras de p,
-- calculadas usando el regla de Ruffini. Por ejemplo,
--   raicesRuffini ejPol1 == []
--   raicesRuffini ejPol2 == [0, -1]
--   raicesRuffini ejPol3 == [0]
--   raicesRuffini ejPol4 == [1, -1, -2]
--   raicesRuffini polCero == []
-----

```

```

raicesRuffini :: Polinomio Int -> [Int]
raicesRuffini p
  | esPolCero p = []
  | otherwise  = aux (0 : divisores (terminoIndep p))
  where
    aux [] = []
    aux (r:rs)
      | esRaizRuffini r p = r : raicesRuffini (cocienteRuffini r p)
      | otherwise        = aux rs

```

```

-----
-- Ejercicio 13. Definir la función
--   factorizacion :: Polinomio Int -> [Polinomio Int]
-- tal que (factorizacion p) es la lista de la descomposición del
-- polinomio p en factores obtenida mediante el regla de Ruffini. Por

```

```

-- ejemplo,
-- ejPol2 == x^5 + 5*x^2 + 4*x
-- factorizacion ejPol2 == [1*x,1*x+1,x^3+-1*x^2+1*x+4]
-- ejPol4 == x^3 + 2*x^2 + -1*x + -2
-- factorizacion ejPol4 == [1*x + -1,1*x + 1,1*x + 2,1]
-- factorizacion (creaPol [1,0,0,0,-1]) == [1*x + -1,1*x + 1,x^2 + 1]
-- -----

```

```

factorizacion :: Polinomio Int -> [Polinomio Int]

```

```

factorizacion p

```

```

| esPolCero p = [p]

```

```

| otherwise = aux (0 : divisores (terminoIndep p))

```

```

where

```

```

aux [] = [p]

```

```

aux (r:rs)

```

```

| esRaizRuffini r p =

```

```

    (creaPol [1,-r]) : factorizacion (cocienteRuffini r p)

```

```

| otherwise = aux rs

```


Relación 19

Operaciones con conjuntos

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir operaciones  
-- entre conjuntos, representados mediante listas ordenadas sin  
-- repeticiones, explicado en el tema 17 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-17t.pdf  
-----  
  
{-# LANGUAGE FlexibleInstances #-}  
  
-----  
-- § Librerías auxiliares --  
-----  
  
import Test.QuickCheck  
  
-----  
-- § Representación de conjuntos y operaciones básicas --  
-----  
  
-- Los conjuntos como listas ordenadas sin repeticiones.  
newtype Conj a = Cj [a]  
  deriving Eq
```

```

-- Procedimiento de escritura de los conjuntos.
instance (Show a) => Show (Conj a) where
  showsPrec _ (Cj s) cad = showConj s cad

showConj []      cad = showString "{}" cad
showConj (x:xs) cad = showChar '{' (shows x (showl xs cad))
  where showl []      cad = showChar '}' cad
        showl (x:xs) cad = showChar ',' (shows x (showl xs cad))

-- Ejemplo de conjunto:
-- ghci> c1
-- {0,1,2,3,5,7,9}
c1, c2, c3, c4 :: Conj Int
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]
c2 = foldr inserta vacio [2,6,8,6,1,2,1,9,6]
c3 = Cj [2..100000]
c4 = Cj [1..100000]

-- vacio es el conjunto vacío. Por ejemplo,
-- ghci> vacio
-- {}
vacio :: Conj a
vacio = Cj []

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
-- esVacio c1      == False
-- esVacio vacio  == True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
-- c1              == {0,1,2,3,5,7,9}
-- pertenece 3 c1 == True
-- pertenece 4 c1 == False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = x `elem` takeWhile (<= x) s

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
-- c1              == {0,1,2,3,5,7,9}

```



```

-- inserta 5 c1 == {0,1,2,3,5,7,9}
-- inserta 4 c1 == {0,1,2,3,4,5,7,9}
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (agrega x s)
  where agrega x [] = [x]
        agrega x s@(y:ys) | x > y = y : agrega x ys
                          | x < y = x : s
                          | otherwise = s

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
-- c1 == {0,1,2,3,5,7,9}
-- elimina 3 c1 == {0,1,2,5,7,9}
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (elimina x s)
  where elimina x [] = []
        elimina x s@(y:ys) | x > y = y : elimina x ys
                          | x < y = s
                          | otherwise = ys

-----
-- § Ejercicios
-----

-----
-- Ejercicio 1. Definir la función
-- subconjunto :: Ord a => Conj a -> Conj a -> Bool
-- tal que (subconjunto c1 c2) se verifica si todos los elementos de c1
-- pertenecen a c2. Por ejemplo,
-- subconjunto (Cj [2..100000]) (Cj [1..100000]) == True
-- subconjunto (Cj [1..100000]) (Cj [2..100000]) == False
-----

-- 1ª definición
subconjunto1 :: Ord a => Conj a -> Conj a -> Bool
subconjunto1 (Cj xs) (Cj ys) = sublista xs ys
  where sublista [] _ = True
        sublista (x:xs) ys = elem x ys && sublista xs ys

-- 2ª definición

```

```
subconjunto2 :: Ord a => Conj a -> Conj a -> Bool
```

```
subconjunto2 (Cj xs) c =
  and [pertenece x c | x <- xs]
```

```
-- 3ª definición
```

```
subconjunto3 :: Ord a => Conj a -> Conj a -> Bool
```

```
subconjunto3 (Cj xs) (Cj ys) = sublista' xs ys
  where sublista' [] _      = True
        sublista' _ []     = False
        sublista' (x:xs) ys@(y:zs) = x >= y && elem x ys &&
                                     sublista' xs zs
```

```
-- 4ª definición
```

```
subconjunto4 :: Ord a => Conj a -> Conj a -> Bool
```

```
subconjunto4 (Cj xs) (Cj ys) = sublista' xs ys
  where sublista' [] _      = True
        sublista' _ []     = False
        sublista' (x:xs) ys@(y:zs)
          | x < y  = False
          | x == y = sublista' xs zs
          | x > y  = elem x zs && sublista' xs zs
```

```
-- Comparación de la eficiencia:
```

```
-- ghci> subconjunto1 (Cj [2..100000]) (Cj [1..100000])
-- C-c C-cInterrupted.
-- ghci> subconjunto2 (Cj [2..100000]) (Cj [1..100000])
-- C-c C-cInterrupted.
-- ghci> subconjunto3 (Cj [2..100000]) (Cj [1..100000])
-- True
-- (0.52 secs, 26097076 bytes)
-- ghci> subconjunto4 (Cj [2..100000]) (Cj [1..100000])
-- True
-- (0.66 secs, 32236700 bytes)
-- ghci> subconjunto1 (Cj [2..100000]) (Cj [1..10000])
-- False
-- (0.54 secs, 3679024 bytes)
-- ghci> subconjunto2 (Cj [2..100000]) (Cj [1..10000])
-- False
-- (38.19 secs, 1415562032 bytes)
-- ghci> subconjunto3 (Cj [2..100000]) (Cj [1..10000])
```

```
-- False
-- (0.08 secs, 3201112 bytes)
-- ghci> subconjunto4 (Cj [2..100000]) (Cj [1..10000])
-- False
-- (0.09 secs, 3708988 bytes)
```

```
-- En lo que sigue, se usará la 3ª definición:
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto = subconjunto3
```

```
-----
-- Ejercicio 2. Definir la función
--   subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
-- tal (subconjuntoPropio c1 c2) se verifica si c1 es un subconjunto
-- propio de c2. Por ejemplo,
--   subconjuntoPropio (Cj [2..5]) (Cj [1..7]) == True
--   subconjuntoPropio (Cj [2..5]) (Cj [1..4]) == False
--   subconjuntoPropio (Cj [2..5]) (Cj [2..5]) == False
-----
```

```
subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
subconjuntoPropio c1 c2 =
  subconjunto c1 c2 && c1 /= c2
```

```
-----
-- Ejercicio 3. Definir la función
--   unitario :: Ord a => a -> Conj a
-- tal que (unitario x) es el conjunto {x}. Por ejemplo,
--   unitario 5 == {5}
-----
```

```
unitario :: Ord a => a -> Conj a
unitario x = inserta x vacio
```

```
-----
-- Ejercicio 4. Definir la función
--   cardinal :: Conj a -> Int
-- tal que (cardinal c) es el número de elementos del conjunto c. Por
-- ejemplo,
--   cardinal c1 == 7
-----
```

```

--     cardinal c2 == 5
-----

cardinal :: Conj a -> Int
cardinal (Cj xs) = length xs

-----

-- Ejercicio 5. Definir la función
--     union :: Ord a => Conj a -> Conj a -> Conj a
-- tal (union c1 c2) es la unión de ambos conjuntos. Por ejemplo,
--     union c1 c2           == {0,1,2,3,5,6,7,8,9}
--     cardinal (union2 c3 c4) == 100000
-----

-- 1ª definición:
union1 :: Ord a => Conj a -> Conj a -> Conj a
union1 (Cj xs) (Cj ys) = foldr inserta (Cj ys) xs

-- Otra definición es
union2 :: Ord a => Conj a -> Conj a -> Conj a
union2 (Cj xs) (Cj ys) = Cj (unionL xs ys)
  where unionL [] ys = ys
        unionL xs [] = xs
        unionL l1@(x:xs) l2@(y:ys)
          | x < y  = x : unionL xs l2
          | x == y = x : unionL xs ys
          | x > y  = y : unionL l1 ys

-- Comparación de eficiencia
--     ghci> :set +s
--     ghci> let c = Cj [1..1000]
--     ghci> cardinal (union1 c c)
--     1000
--     (1.04 secs, 56914332 bytes)
--     ghci> cardinal (union2 c c)
--     1000
--     (0.01 secs, 549596 bytes)

-- En lo que sigue se usará la 2ª definición
union :: Ord a => Conj a -> Conj a -> Conj a

```

```
union = union2
```

```
-----
-- Ejercicio 6. Definir la función
--   unionG :: Ord a => [Conj a] -> Conj a
-- tal (unionG cs) calcule la unión de la lista de conjuntos cd. Por
-- ejemplo,
--   unionG [c1, c2] == {0,1,2,3,5,6,7,8,9}
-----
```

```
unionG :: Ord a => [Conj a] -> Conj a
unionG [] = vacio
unionG (Cj xs:css) = Cj xs 'union' unionG css
```

```
-- Se puede definir por plegados
unionG2 :: Ord a => [Conj a] -> Conj a
unionG2 = foldr union vacio
```

```
-----
-- Ejercicio 7. Definir la función
--   interseccion :: Eq a => Conj a -> Conj a -> Conj a
-- tal que (interseccion c1 c2) es la intersección de los conjuntos c1 y
-- c2. Por ejemplo,
--   interseccion (Cj [1..7]) (Cj [4..9]) == {4,5,6,7}
--   interseccion (Cj [2..1000000]) (Cj [1]) == {}
-----
```

```
-- 1ª definición
interseccion1 :: Eq a => Conj a -> Conj a -> Conj a
interseccion1 (Cj xs) (Cj ys) = Cj [x | x <- xs, x 'elem' ys]
```

```
-- 2ª definición
interseccion2 :: Ord a => Conj a -> Conj a -> Conj a
interseccion2 (Cj xs) (Cj ys) = Cj (interseccionL xs ys)
  where interseccionL l1@(x:xs) l2@(y:ys)
        | x > y = interseccionL l1 ys
        | x == y = x : interseccionL xs ys
        | x < y = interseccionL xs l2
interseccionL _ _ = []
```

```
-- La comparación de eficiencia es
-- ghci> interseccion1 (Cj [2..1000000]) (Cj [1])
-- {}
-- (0.32 secs, 80396188 bytes)
-- ghci> interseccion2 (Cj [2..1000000]) (Cj [1])
-- {}
-- (0.00 secs, 2108848 bytes)

-- En lo que sigue se usa la 2ª definición:
interseccion :: Ord a => Conj a -> Conj a -> Conj a
interseccion = interseccion2
```

```
-----
-- Ejercicio 8. Definir la función
--   interseccionG :: Ord a => [Conj a] -> Conj a
-- tal que (interseccionG cs) es la intersección de la lista de
-- conjuntos cs. Por ejemplo,
--   interseccionG [c1, c2] == {1,2,9}
-----
```

```
interseccionG :: Ord a => [Conj a] -> Conj a
interseccionG [c] = c
interseccionG (cs:css) = interseccion cs (interseccionG css)
```

```
-- Se puede definir por plegado
interseccionG2 :: Ord a => [Conj a] -> Conj a
interseccionG2 = foldr1 interseccion
```

```
-----
-- Ejercicio 9. Definir la función
--   disjuntos :: Ord a => Conj a -> Conj a -> Bool
-- tal que (disjuntos c1 c2) se verifica si los conjuntos c1 y c2 son
-- disjuntos. Por ejemplo,
--   disjuntos (Cj [2..5]) (Cj [6..9]) == True
--   disjuntos (Cj [2..5]) (Cj [1..9]) == False
-----
```

```
disjuntos :: Ord a => Conj a -> Conj a -> Bool
disjuntos c1 c2 = esVacio (interseccion c1 c2)
```

```

-----
-- Ejercicio 10. Definir la función
--   diferencia :: Eq a => Conj a -> Conj a -> Conj a
-- tal que (diferencia c1 c2) es el conjunto de los elementos de c1 que
-- no son elementos de c2. Por ejemplo,
--   diferencia c1 c2 == {0,3,5,7}
--   diferencia c2 c1 == {6,8}
-----

```

```

diferencia :: Eq a => Conj a -> Conj a -> Conj a
diferencia (Cj xs) (Cj ys) = Cj zs
  where zs = [x | x <- xs, x 'notElem' ys]

```

```

-----
-- Ejercicio 11. Definir la función
--   diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
-- tal que (diferenciaSimetrica c1 c2) es la diferencia simétrica de los
-- conjuntos c1 y c2. Por ejemplo,
--   diferenciaSimetrica c1 c2 == {0,3,5,6,7,8}
--   diferenciaSimetrica c2 c1 == {0,3,5,6,7,8}
-----

```

```

diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
diferenciaSimetrica c1 c2 =
  diferencia (union c1 c2) (interseccion c1 c2)

```

```

-----
-- Ejercicio 12. Definir la función
--   filtra :: (a -> Bool) -> Conj a -> Conj a
-- tal (filtra p c) es el conjunto de elementos de c que verifican el
-- predicado p. Por ejemplo,
--   filtra even c1 == {0,2}
--   filtra odd  c1 == {1,3,5,7,9}
-----

```

```

filtra :: (a -> Bool) -> Conj a -> Conj a
filtra p (Cj xs) = Cj (filter p xs)

```

```

-----
-- Ejercicio 13. Definir la función

```

```
--   particion :: (a -> Bool) -> Conj a -> (Conj a, Conj a)
--   tal que (particion c) es el par formado por dos conjuntos: el de sus
--   elementos que verifican p y el de los elementos que no lo
--   verifica. Por ejemplo,
--   particion even c1 == ({0,2},{1,3,5,7,9})
-----
```

```
particion :: (a -> Bool) -> Conj a -> (Conj a, Conj a)
particion p c = (filtra p c, filtra (not . p) c)
```

```
-- Ejercicio 14. Definir la función
--   divide :: (Ord a) => a -> Conj a -> (Conj a, Conj a)
--   tal que (divide x c) es el par formado por dos subconjuntos de c: el
--   de los elementos menores o iguales que x y el de los mayores que x.
--   Por ejemplo,
--   divide 5 c1 == ({0,1,2,3,5},{7,9})
-----
```

```
divide :: Ord a => a -> Conj a -> (Conj a, Conj a)
divide x = particion (<= x)
```

```
-- Ejercicio 15. Definir la función
--   mapC :: (a -> b) -> Conj a -> Conj b
--   tal que (map f c) es el conjunto formado por las imágenes de los
--   elementos de c, mediante f. Por ejemplo,
--   mapC (*2) (Cj [1..4]) == {2,4,6,8}
-----
```

```
mapC :: (a -> b) -> Conj a -> Conj b
mapC f (Cj xs) = Cj (map f xs)
```

```
-- Ejercicio 16. Definir la función
--   everyC :: (a -> Bool) -> Conj a -> Bool
--   tal que (everyC p c) se verifica si todos los elemntos de c
--   verifican el predicado p. Por ejemplo,
--   everyC even (Cj [2,4..10]) == True
--   everyC even (Cj [2..10])   == False
```



```

-----
everyC :: (a -> Bool) -> Conj a -> Bool
everyC p (Cj xs) = all p xs

```

```

-----
-- Ejercicio 17. Definir la función
--   someC :: (a -> Bool) -> Conj a -> Bool
-- tal que (someC p c) se verifica si algún elemento de c verifica el
-- predicado p. Por ejemplo,
--   someC even (Cj [1,4,7]) == True
--   someC even (Cj [1,3,7]) == False
-----

```

```

someC :: (a -> Bool) -> Conj a -> Bool
someC p (Cj xs) = any p xs

```

```

-----
-- Ejercicio 18. Definir la función
--   productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
-- tal que (productoC c1 c2) es el producto cartesiano de los
-- conjuntos c1 y c2. Por ejemplo,
--   productoC (Cj [1,3]) (Cj [2,4]) == {(1,2),(1,4),(3,2),(3,4)}
-----

```

```

productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
productoC (Cj xs) (Cj ys) =
  foldr inserta vacio [(x,y) | x <- xs, y <- ys]

```

```

-----
-- Ejercicio. Especificar que, dado un tipo ordenado a, el orden entre
-- los conjuntos con elementos en a es el orden inducido por el orden
-- existente entre las listas con elementos en a.
-----

```

```

instance Ord a => Ord (Conj a) where
  (Cj xs) <= (Cj ys) = xs <= ys

```

```

-----
-- Ejercicio 19. Definir la función

```

```

-- potencia :: Ord a => Conj a -> Conj (Conj a)
-- tal que (potencia c) es el conjunto potencia de c; es decir, el
-- conjunto de todos los subconjuntos de c. Por ejemplo,
-- potencia (Cj [1,2]) == {{},{1},{1,2},{2}}
-- potencia (Cj [1..3]) == {{},{1},{1,2},{1,2,3},{1,3},{2},{2,3},{3}}
-----

```

```

potencia :: Ord a => Conj a -> Conj (Conj a)
potencia (Cj []) = unitario vacio
potencia (Cj (x:xs)) = mapC (inserta x) pr 'union' pr
  where pr = potencia (Cj xs)

```

```

-----
-- Ejercicio 20. Comprobar con QuickCheck que la relación de subconjunto
-- es un orden parcial. Es decir, es una relación reflexiva,
-- antisimétrica y transitiva.
-----

```

```

propSubconjuntoReflexiva :: Conj Int -> Bool
propSubconjuntoReflexiva c = subconjunto c c

```

```

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

```

```

propSubconjuntoAntisimetrica :: Conj Int -> Conj Int -> Property
propSubconjuntoAntisimetrica c1 c2 =
  subconjunto c1 c2 && subconjunto c2 c1 ==> c1 == c2

```

```

-- La comprobación es
-- ghci> quickCheck propSubconjuntoAntisimetrica
-- *** Gave up! Passed only 13 tests.

```

```

propSubconjuntoTransitiva :: Conj Int -> Conj Int -> Conj Int -> Property
propSubconjuntoTransitiva c1 c2 c3 =
  subconjunto c1 c2 && subconjunto c2 c3 ==> subconjunto c1 c3

```

```

-- La comprobación es
-- ghci> quickCheck propSubconjuntoTransitiva
-- *** Gave up! Passed only 7 tests.

```

```

-----
-- Ejercicio 21. Comprobar con QuickCheck que el conjunto vacío está
-- contenido en cualquier conjunto.
-----

```

```

propSubconjuntoVacio :: Conj Int -> Bool
propSubconjuntoVacio c = subconjunto vacio c

```

```

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

```

```

-----
-- Ejercicio 22. Comprobar con QuickCheck las siguientes propiedades de
-- la unión de conjuntos:
--   Idempotente:      A U A = A
--   Neutro:           A U {} = A
--   Conmutativa:      A U B = B U A
--   Asociativa:       A U (B U C) = (A U B) U C
--   UnionSubconjunto: A y B son subconjuntos de (A U B)
--   UnionDiferencia:  A U B = A U (B \ A)
-----

```

```

propUnionIdempotente :: Conj Int -> Bool
propUnionIdempotente c =
  union c c == c

```

```

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

```

```

propVacioNeutroUnion :: Conj Int -> Bool
propVacioNeutroUnion c =
  union c vacio == c

```

```

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

```

```
propUnionCommutativa :: Conj Int -> Conj Int -> Bool
```

```
propUnionCommutativa c1 c2 =
  union c1 c2 == union c2 c1
```

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

```
propUnionAsociativa :: Conj Int -> Conj Int -> Conj Int -> Bool
```

```
propUnionAsociativa c1 c2 c3 =
  union c1 (union c2 c3) == union (union c1 c2) c3
```

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

```
propUnionSubconjunto :: Conj Int -> Conj Int -> Bool
```

```
propUnionSubconjunto c1 c2 =
  subconjunto c1 c3 && subconjunto c2 c3
  where c3 = union c1 c2
```

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

```
propUnionDiferencia :: Conj Int -> Conj Int -> Bool
```

```
propUnionDiferencia c1 c2 =
  union c1 c2 == union c1 (diferencia c2 c1)
```

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

```
-----
-- Ejercicio 23. Comprobar con QuickCheck las siguientes propiedades de
-- la intersección de conjuntos:
-- Idempotente:           A n A = A
-- VacioInterseccion:    A n {} = {}
-- Commutativa:           A n B = B n A
-- Asociativa:            A n (B n C) = (A n B) n C
```

```

--   InterseccionSubconjunto: (A n B) es subconjunto de A y B
--   DistributivaIU:          A n (B U C) = (A n B) U (A n C)
--   DistributivaUI:          A U (B n C) = (A U B) n (A U C)
-----

propInterseccionIdempotente :: Conj Int -> Bool
propInterseccionIdempotente c =
  interseccion c c == c

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

propVacioInterseccion :: Conj Int -> Bool
propVacioInterseccion c =
  interseccion c vacio == vacio

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

propInterseccionCommutativa :: Conj Int -> Conj Int -> Bool
propInterseccionCommutativa c1 c2 =
  interseccion c1 c2 == interseccion c2 c1

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

propInterseccionAsociativa :: Conj Int -> Conj Int -> Conj Int -> Bool
propInterseccionAsociativa c1 c2 c3 =
  interseccion c1 (interseccion c2 c3) == interseccion (interseccion c1 c2) c3

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

propInterseccionSubconjunto :: Conj Int -> Conj Int -> Bool
propInterseccionSubconjunto c1 c2 =

```

```

subconjunto c3 c1 && subconjunto c3 c2
  where c3 = interseccion c1 c2

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

propDistributivaIU :: Conj Int -> Conj Int -> Conj Int -> Bool
propDistributivaIU c1 c2 c3 =
  interseccion c1 (union c2 c3) == union (interseccion c1 c2)
                                     (interseccion c1 c3)

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

propDistributivaUI :: Conj Int -> Conj Int -> Conj Int -> Bool
propDistributivaUI c1 c2 c3 =
  union c1 (interseccion c2 c3) == interseccion (union c1 c2)
                                     (union c1 c3)

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

-----
-- Ejercicio 24. Comprobar con QuickCheck las siguientes propiedades de
-- la diferencia de conjuntos:
-- DiferenciaVacio1:  $A \setminus \{\} = A$ 
-- DiferenciaVacio2:  $\{\} \setminus A = \{\}$ 
-- DiferenciaDif1:  $(A \setminus B) \setminus C = A \setminus (B \cup C)$ 
-- DiferenciaDif2:  $A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$ 
-- DiferenciaSubc:  $(A \setminus B)$  es subconjunto de  $A$ 
-- DiferenciaDisj:  $A$  y  $(B \setminus A)$  son disjuntos
-- DiferenciaUI:  $(A \cup B) \setminus A = B \setminus (A \cap B)$ 
-----

propDiferenciaVacio1 :: Conj Int -> Bool
propDiferenciaVacio1 c = diferencia c vacio == c

-- La comprobación es

```

```
-- ghci> quickCheck propDiferenciaVacio2
-- +++ OK, passed 100 tests.

propDiferenciaVacio2 :: Conj Int -> Bool
propDiferenciaVacio2 c = diferencia vacio c == vacio

-- La comprobación es
-- ghci> quickCheck propDiferenciaVacio2
-- +++ OK, passed 100 tests.

propDiferenciaDif1 :: Conj Int -> Conj Int -> Conj Int -> Bool
propDiferenciaDif1 c1 c2 c3 =
  diferencia (diferencia c1 c2) c3 == diferencia c1 (union c2 c3)

-- La comprobación es
-- ghci> quickCheck propDiferenciaDif1
-- +++ OK, passed 100 tests.

propDiferenciaDif2 :: Conj Int -> Conj Int -> Conj Int -> Bool
propDiferenciaDif2 c1 c2 c3 =
  diferencia c1 (diferencia c2 c3) == union (diferencia c1 c2)
                                           (interseccion c1 c3)

-- La comprobación es
-- ghci> quickCheck propDiferenciaDif2
-- +++ OK, passed 100 tests.

propDiferenciaSubc :: Conj Int -> Conj Int -> Bool
propDiferenciaSubc c1 c2 =
  subconjunto (diferencia c1 c2) c1

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

propDiferenciaDisj :: Conj Int -> Conj Int -> Bool
propDiferenciaDisj c1 c2 =
  disjuntos c1 (diferencia c2 c1)

-- La comprobación es
```

```

-- ghci> quickCheck propDiferenciaDisj
-- +++ OK, passed 100 tests.

propDiferenciaUI :: Conj Int -> Conj Int -> Bool
propDiferenciaUI c1 c2 =
  diferencia (union c1 c2) c1 == diferencia c2 (interseccion c1 c2)

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

-----
-- Generador de conjuntos --
-----

-- genConjunto es un generador de conjuntos. Por ejemplo,
-- ghci> sample genConjunto
-- {}
-- {}
-- {}
-- {3,-2,-2,-3,-2,4}
-- {-8,0,4,6,-5,-2}
-- {12,-2,-1,-10,-2,2,15,15}
-- {2}
-- {}
-- {-42,55,55,-11,23,23,-11,27,-17,-48,16,-15,-7,5,41,43}
-- {-124,-66,-5,-47,58,-88,-32,-125}
-- {49,-38,-231,-117,-32,-3,45,227,-41,54,169,-160,19}
genConjunto :: Gen (Conj Int)
genConjunto = do xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance Arbitrary (Conj Int) where
  arbitrary = genConjunto

```


Relación 20

Relaciones binarias homogéneas

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir propiedades y  
-- operaciones sobre las relaciones binarias (homogéneas).  
--  
-- Como referencia se puede usar el artículo de la wikipedia  
-- http://bit.ly/HVHOPS  
  
-----  
-- § Librerías auxiliares --  
-----  
  
import Test.QuickCheck  
import Data.List  
  
-----  
-- Ejercicio 1. Una relación binaria R sobre un conjunto A puede  
-- representar mediante un par (xs,ps) donde xs es la lista de los  
-- elementos de A (el universo de R) y ps es la lista de pares de R (el  
-- grafo de R). Definir el tipo de dato (Rel a) para representar las  
-- relaciones binarias sobre a.  
-----  
  
type Rel a = ([a],[(a,a)])
```

```

-----
-- Nota. En los ejemplos usaremos las siguientes relaciones binarias:
--   r1, r2, r3 :: Rel Int
--   r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
--   r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
--   r3 = ([1..9],[(1,3), (2,6), (8,9), (3,6)])
-----

```

```

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

```

```

-----
-- Ejercicio 2. Definir la función
--   universo :: Eq a => Rel a -> [a]
-- tal que (universo r) es el universo de la relación r. Por ejemplo,
--   r1           == ([1,2,3,4,5,6,7,8,9],[(1,3), (2,6), (8,9), (2,7)])
--   universo r1 == [1,2,3,4,5,6,7,8,9]
-----

```

```

universo :: Eq a => Rel a -> [a]
universo (us,_) = us

```

```

-----
-- Ejercicio 3. Definir la función
--   grafo :: Eq a => ([a],[(a,a)]) -> [(a,a)]
-- tal que (grafo r) es el grafo de la relación r. Por ejemplo,
--   r1           == ([1,2,3,4,5,6,7,8,9],[(1,3), (2,6), (8,9), (2,7)])
--   grafo r1    == [(1,3), (2,6), (8,9), (2,7)]
-----

```

```

grafo :: Eq a => Rel a -> [(a,a)]
grafo (_,ps) = ps

```

```

-----
-- Ejercicio 4. Definir la función
--   reflexiva :: Eq a => Rel a -> Bool
-- tal que (reflexiva r) se verifica si la relación r es reflexiva. Por
-- ejemplo,

```

```
-- reflexiva ([1,3],[(1,1),(1,3),(3,3)]) == True
-- reflexiva ([1,2,3],[(1,1),(1,3),(3,3)]) == False
```

```
-----
reflexiva :: Eq a => Rel a -> Bool
reflexiva (us,ps) = and [elem (x,x) ps | x <- us]
```

```
-----
-- Ejercicio 5. Definir la función
-- simetrica :: Eq a => Rel a -> Bool
-- tal que (simetrica r) se verifica si la relación r es simétrica. Por
-- ejemplo,
-- simetrica ([1,3],[(1,1),(1,3),(3,1)]) == True
-- simetrica ([1,3],[(1,1),(1,3),(3,2)]) == False
-- simetrica ([1,3],[]) == True
```

```
-----
simetrica :: Eq a => Rel a -> Bool
simetrica (us,ps) = and [(y,x) 'elem' ps | (x,y) <- ps]
```

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

```
-----
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]
```

```
-----
-- Ejercicio 7. Definir la función
-- composicion :: Eq a => Rel a -> Rel a -> Rel a
-- tal que (composicion r s) es la composición de las relaciones r y
-- s. Por ejemplo,
-- ghci> composicion ([1,2],[(1,2),(2,2)]) ([1,2],[(2,1)])
-- ([1,2],[(1,1),(2,1)])
```

```

composicion :: Eq a => Rel a -> Rel a -> Rel a
composicion (xs,ps) (_,qs) =
  (xs,[(x,z) | (x,y) <- ps, (y',z) <- qs, y == y'])

```

```

-----
-- Ejercicio 8. Definir la función
--   transitiva :: Eq a => Rel a -> Bool
-- tal que (transitiva r) se verifica si la relación r es transitiva.
-- Por ejemplo,
--   transitiva ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)]) == True
--   transitiva ([1,3,5],[(1,1),(1,3),(3,1),(5,5)])       == False
-----

```

```

transitiva :: Eq a => Rel a -> Bool
transitiva r@(xs,ps) =
  subconjunto (grafo (composicion r r)) ps

```

```

-----
-- Ejercicio 9. Definir la función
--   esEquivalencia :: Eq a => Rel a -> Bool
-- tal que (esEquivalencia r) se verifica si la relación r es de
-- equivalencia. Por ejemplo,
--   ghci> esEquivalencia ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])
--   True
--   ghci> esEquivalencia ([1,2,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])
--   False
--   ghci> esEquivalencia ([1,3,5],[(1,1),(1,3),(3,3),(5,5)])
--   False
-----

```

```

esEquivalencia :: Eq a => Rel a -> Bool
esEquivalencia r = reflexiva r && simetrica r && transitiva r

```

```

-----
-- Ejercicio 10. Definir la función
--   irreflexiva :: Eq a => Rel a -> Bool
-- tal que (irreflexiva r) se verifica si la relación r es irreflexiva;
-- es decir, si ningún elemento de su universo está relacionado con
-- él mismo. Por ejemplo,

```

```
-- irreflexiva ([1,2,3],[1,2],[2,1],[2,3]) == True
-- irreflexiva ([1,2,3],[1,2],[2,1],[3,3]) == False
```

```
-----
irreflexiva :: Eq a => Rel a -> Bool
irreflexiva (xs,ps) = and [(x,x) 'notElem' ps | x <- xs]
```

```
-----
-- Ejercicio 11. Definir la función
-- antisimetrica :: Eq a => Rel a -> Bool
-- tal que (antisimetrica r) se verifica si la relación r es
-- antisimétrica; es decir, si (x,y) e (y,x) están relacionado, entonces
-- x=y. Por ejemplo,
-- antisimetrica ([1,2],[1,2]) == True
-- antisimetrica ([1,2],[1,2],[2,1]) == False
-- antisimetrica ([1,2],[1,1],[2,1]) == True
```

```
-----
antisimetrica :: Eq a => Rel a -> Bool
antisimetrica (_,ps) =
  null [(x,y) | (x,y) <- ps, x /= y, (y,x) 'elem' ps]
```

```
-- 2ª definición
antisimetrica2 :: Eq a => Rel a -> Bool
antisimetrica2 (_,ps) = and [notElem (y,x) ps | (x,y) <- ps, x /= y]
```

```
-- 3ª definición
antisimetrica3 :: Eq a => Rel a -> Bool
antisimetrica3 (xs,ps) =
  and [(x,y) 'elem' ps && (y,x) 'elem' ps --> (x == y)
       | x <- xs, y <- xs]
  where p --> q = not p || q
```

```
-----
-- Ejercicio 12. Definir la función
-- total :: Eq a => Rel a -> Bool
-- tal que (total r) se verifica si la relación r es total; es decir, si
-- para cualquier par x, y de elementos del universo de r, se tiene que
-- x está relacionado con y ó y está relacionado con x. Por ejemplo,
```

```
-- total ([1,3],[(1,1),(3,1),(3,3)]) == True
-- total ([1,3],[(1,1),(3,1)])       == False
-- total ([1,3],[(1,1),(3,3)])       == False
```

```
total :: Eq a => Rel a -> Bool
```

```
total (xs,ps) =
  and [(x,y) 'elem' ps || (y,x) 'elem' ps | x <- xs, y <- xs]
```

```
-- Ejercicio 13. Comprobar con QuickCheck que las relaciones totales son
-- reflexivas.
```

```
prop_total_reflexiva :: Rel Int -> Property
```

```
prop_total_reflexiva r =
  total r ==> reflexiva r
```

```
-- La comprobación es
-- ghci> quickCheck prop_total_reflexiva
-- *** Gave up! Passed only 19 tests.
```

```
-- § Clausuras
```

```
-- Ejercicio 14. Definir la función
```

```
-- clausuraReflexiva :: Eq a => Rel a -> Rel a
-- tal que (clausuraReflexiva r) es la clausura reflexiva de r; es
-- decir, la menor relación reflexiva que contiene a r. Por ejemplo,
-- ghci> clausuraReflexiva ([1,3],[(1,1),(3,1)])
-- ([1,3],[(1,1),(3,1),(3,3)])
```

```
clausuraReflexiva :: Eq a => Rel a -> Rel a
```

```
clausuraReflexiva (xs,ps) =
  (xs, ps 'union' [(x,x) | x <- xs])
```

```
-- Ejercicio 15. Comprobar con QuickCheck que clausuraReflexiva es
-- reflexiva.
```

```
-----
prop_ClausuraReflexiva :: Rel Int -> Bool
prop_ClausuraReflexiva r =
  reflexiva (clausuraReflexiva r)
```

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

```
-----
-- Ejercicio 16. Definir la función
--   clausuraSimetrica :: Eq a => Rel a -> Rel a
-- tal que (clausuraSimetrica r) es la clausura simétrica de r; es
-- decir, la menor relación simétrica que contiene a r. Por ejemplo,
-- ghci> clausuraSimetrica ([1,3,5],[(1,1),(3,1),(1,5)])
--   [(1,3,5],[(1,1),(3,1),(1,5),(1,3),(5,1)])
```

```
-----
clausuraSimetrica :: Eq a => Rel a -> Rel a
clausuraSimetrica (xs,ps) =
  (xs, ps 'union' [(y,x) | (x,y) <- ps])
```

```
-----
-- Ejercicio 17. Comprobar con QuickCheck que clausuraSimetrica es
-- simétrica.
```

```
-----
prop_ClausuraSimetrica :: Rel Int -> Bool
prop_ClausuraSimetrica r =
  simetrica (clausuraSimetrica r)
```

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

```
-----
-- Ejercicio 18. Definir la función
```

```
--   clausuraTransitiva :: Eq a => Rel a -> Rel a
--   tal que (clausuraTransitiva r) es la clausura transitiva de r; es
--   decir, la menor relación transitiva que contiene a r. Por ejemplo,
--   ghci> clausuraTransitiva ([1..6],[(1,2),(2,5),(5,6)])
--   ([1,2,3,4,5,6],[(1,2),(2,5),(5,6),(1,5),(2,6),(1,6)])
```

```
-----
clausuraTransitiva :: Eq a => Rel a -> Rel a
clausuraTransitiva (xs,ps) = (xs, aux ps)
  where aux xs | cerradoTr xs = xs
              | otherwise    = aux (xs 'union' (comp xs xs))
        cerradoTr r = subconjunto (comp r r) r
        comp r s    = [(x,z) | (x,y) <- r, (y',z) <- s, y == y']
```

```
-----
--   Ejercicio 19. Comprobar con QuickCheck que clausuraTransitiva es
--   transitiva.
```

```
-----
prop_ClausuraTransitiva :: Rel Int -> Bool
prop_ClausuraTransitiva r =
  transitiva (clausuraTransitiva r)

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


Relación 21

Ecuaciones con factoriales

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

```
-- El objetivo de esta relación de ejercicios es resolver la ecuación  
--  $a! * b! = a! + b! + c!$   
-- donde  $a$ ,  $b$  y  $c$  son números naturales.
```

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

```
import Test.QuickCheck
```

```
-----  
-- Ejercicio 1. Definir la función  
-- factorial :: Integer -> Integer  
-- tal que (factorial n) es el factorial de  $n$ . Por ejemplo,  
-- factorial 5 == 120  
-----
```

```
factorial :: Integer -> Integer  
factorial n = product [1..n]
```

```
-----  
-- Ejercicio 2. Definir la constante  
-- factoriales :: [Integer]
```

```
-- tal que factoriales es la lista de los factoriales de los números
-- naturales. Por ejemplo,
--   take 7 factoriales == [1,1,2,6,24,120,720]
```

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

```
-----
-- Ejercicio 3. Definir, usando factoriales, la función
--   esFactorial :: Integer -> Bool
-- tal que (esFactorial n) se verifica si existe un número natural m
-- tal que n es m!. Por ejemplo,
--   esFactorial 120 == True
--   esFactorial 20  == False
```

```
-----
esFactorial :: Integer -> Bool
esFactorial n = n == head (dropWhile (<n) factoriales)
```

```
-----
-- Ejercicio 4. Definir la constante
--   posicionesFactoriales :: [(Integer,Integer)]
-- tal que posicionesFactoriales es la lista de los factoriales con su
-- posición. Por ejemplo,
--   ghci> take 7 posicionesFactoriales
--   [(0,1),(1,1),(2,2),(3,6),(4,24),(5,120),(6,720)]
```

```
-----
posicionesFactoriales :: [(Integer,Integer)]
posicionesFactoriales = zip [0..] factoriales
```

```
-----
-- Ejercicio 5. Definir la función
--   invFactorial :: Integer -> Maybe Integer
-- tal que (invFactorial x) es (Just n) si el factorial de n es x y es
-- Nothing, en caso contrario. Por ejemplo,
--   invFactorial 120 == Just 5
--   invFactorial 20  == Nothing
```

```

invFactorial :: Integer -> Maybe Integer
invFactorial x
  | esFactorial x = Just (head [n | (n,y) <- posicionesFactoriales, y==x])
  | otherwise     = Nothing

```

```

-----
-- Ejercicio 6. Definir la constante
--   pares :: [(Integer,Integer)]
--   tal que pares es la lista de todos los pares de números naturales. Por
--   ejemplo,
--   ghci> take 11 pares
--   [(0,0),(0,1),(1,1),(0,2),(1,2),(2,2),(0,3),(1,3),(2,3),(3,3),(0,4)]
-----

```

```

pares :: [(Integer,Integer)]
pares = [(x,y) | y <- [0..], x <- [0..y]]

```

```

-----
-- Ejercicio 7. Definir la constante
--   solucionFactoriales :: (Integer,Integer,Integer)
--   tal que solucionFactoriales es una terna (a,b,c) que es una solución
--   de la ecuación
--    $a! * b! = a! + b! + c!$ 
--   Calcular el valor de solucionFactoriales.
-----

```

```

solucionFactoriales :: (Integer,Integer,Integer)
solucionFactoriales = (a,b,c)
  where (a,b) = head [(x,y) | (x,y) <- pares,
                             esFactorial (f x * f y - f x - f y)]
        f     = factorial
        Just c = invFactorial (f a * f b - f a - f b)

```

```

-- El cálculo es
--   ghci> solucionFactoriales
--   (3,3,4)

```

```

-----
-- Ejercicio 8. Comprobar con QuickCheck que solucionFactoriales es la

```

```
-- única solución de la ecuación
--  $a! * b! = a! + b! + c!$ 
-- con a, b y c números naturales
```

```
-----
prop_solucionFactoriales :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales x y z =
  x >= 0 && y >= 0 && z >= 0 && (x,y,z) /= solucionFactoriales
  ==> not (f x * f y == f x + f y + f z)
  where f = factorial
```

```
-- La comprobación es
-- ghci> quickCheck prop_solucionFactoriales
-- *** Gave up! Passed only 86 tests.
```

```
-- También se puede expresar como
```

```
prop_solucionFactoriales' :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales' x y z =
  x >= 0 && y >= 0 && z >= 0 &&
  f x * f y == f x + f y + f z
  ==> (x,y,z) == solucionFactoriales
  where f = factorial
```

```
-- La comprobación es
-- ghci> quickCheck prop_solucionFactoriales
-- *** Gave up! Passed only 0 tests.
```

```
-----
-- Nota: El ejercicio se basa en el artículo "Ecuación con factoriales"
-- del blog Gaussianos publicado en
-- http://gaussianos.com/ecuacion-con-factoriales
-----
```

Relación 22

Vectores y matrices con las librerías

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es adaptar los ejercicios de la relación  
-- 26 (sobre vectores y matrices) usando las librerías Data.Vector y  
-- Data.Matrix.  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
import qualified Data.Vector as V  
import Data.Matrix  
import Data.Ratio  
  
-- -----  
-- Tipos de los vectores y de las matrices --  
-- -----  
  
-- Los vectores (con elementos de tipo a son del tipo (V.Vector a).  
-- Los matrices (con elementos de tipo a son del tipo (Matrix a).  
  
-- -----  
-- Operaciones básicas con matrices --  
-- -----
```

```

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

```

```

listaVector :: Num a => [a] -> V.Vector a
listaVector = V.fromList

```

```

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

```

```

listaMatriz :: Num a => [[a]] -> Matrix a
listaMatriz = fromLists

```

```

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

```

```

numFilas :: Num a => Matrix a -> Int
numFilas = nRows

```

```

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

```

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

```
numColumns :: Num a => Matrix a -> Int
numColumns = nCols
```

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

```
dimension :: Num a => Matrix a -> (Int,Int)
dimension p = (numFilas p, numColumns p)
```

```
-- Ejercicio 7. Definir la función
-- matrizLista :: Num a => Matrix a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
-- ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> m
-- ( 5 1 0 )
-- ( 3 2 6 )
-- ghci> matrizLista m
-- [[5,1,0],[3,2,6]]
```

```
matrizLista :: Num a => Matrix a -> [[a]]
matrizLista p =
  [[getElem i j p | j <- [1..nCols p]] | i <- [1..nRows p]]
```

```
-- Ejercicio 8. Definir la función
-- vectorLista :: Num a => V.Vector a -> [a]
-- tal que (vectorLista x) es la lista de los elementos del vector
```

```
-- v. Por ejemplo,
-- ghci> let v = listaVector [3,2,5]
-- ghci> v
-- fromList [3,2,5]
-- ghci> vectorLista v
-- [3,2,5]
```

```
vectorLista :: Num a => V.Vector a -> [a]
vectorLista = V.toList
```

```
-- Suma de matrices
```

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

```
sumaMatrices :: Num a => Matrix a -> Matrix a -> Matrix a
sumaMatrices p q = p + q
```

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



```
-----  
filaMat :: Num a => Int -> Matrix a -> V.Vector a  
filaMat = getRow
```

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

```
columnaMat :: Num a => Int -> Matrix a -> V.Vector a  
columnaMat = getCol
```

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

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

```
prodEscalar :: Num a => V.Vector a -> V.Vector a -> a  
prodEscalar v1 v2 = V.sum (V.zipWith (*) v1 v2)
```

```
-----  
-- Ejercicio 13. Definir la función  
-- prodMatrices :: Num a => Matrix a -> Matrix a -> Matrix a
```

```
-- tal que (prodMatrices p q) es el producto de las matrices p y q. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[3,1],[2,4]]
-- ghci> prodMatrices p p
-- ( 11 7 )
-- ( 14 18 )
-- ghci> let q = listaMatriz [[7],[5]]
-- ghci> prodMatrices p q
-- ( 26 )
-- ( 34 )
```

```
prodMatrices :: Num a => Matrix a -> Matrix a -> Matrix a
prodMatrices p q = p * q
```

```
-- -----
-- Traspuestas y simétricas
```

```
-- -----
-- Ejercicio 14. Definir la función
-- traspuesta :: Num a => Matrix a -> Matrix a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> traspuesta p
-- ( 5 3 )
-- ( 1 2 )
-- ( 0 6 )
```

```
traspuesta :: Num a => Matrix a -> Matrix a
traspuesta = transpose
```

```
-- -----
-- Ejercicio 15. Definir la función
-- esCuadrada :: Num a => Matrix a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> esCuadrada p
```

```
-- False
-- ghci> let q = listaMatriz [[5,1],[3,2]]
-- ghci> esCuadrada q
-- True
```

```
-----
esCuadrada :: Num a => Matrix a -> Bool
esCuadrada p = nrows p == ncols p
```

```
-----
-- Ejercicio 16. Definir la función
-- esSimetrica :: (Num a, Eq a) => Matrix a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
-- ghci> esSimetrica p
-- True
-- ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
-- ghci> esSimetrica q
-- False
```

```
-----
esSimetrica :: (Num a, Eq a) => Matrix a -> Bool
esSimetrica x = x == transpose x
```

```
-----
-- Diagonales de una matriz
```

```
-----
-- Ejercicio 17. Definir la función
-- diagonalPral :: Num a => Matrix a -> V.Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> diagonalPral p
-- fromList [5,2]
```

```
-----
diagonalPral :: Num a => Matrix a -> V.Vector a
```

```
diagonalPral = getDiag
```

```
-----
-- Ejercicio 18. Definir la función
--   diagonalSec :: Num a => Matrix a -> V.Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> diagonalSec p
--   fromList [1,3]
--   ghci> let q = traspuesta p
--   ghci> matrizLista q
--   [[5,3],[1,2],[0,6]]
--   ghci> diagonalSec q
--   fromList [1,2]
-----
```

```
diagonalSec :: Num a => Matrix a -> V.Vector a
diagonalSec p = V.fromList [p!(i,m+1-i) | i <- [1..n]]
  where m = nrows p
        n = min m (ncols p)
```

```
-----
-- Submatrices
-----
```

```
-----
-- Ejercicio 19. Definir la función
--   submatriz :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> submatriz 2 3 p
--   ( 5 1 )
--   ( 4 6 )
-----
```

```
submatriz :: Num a => Int -> Int -> Matrix a -> Matrix a
submatriz = minorMatrix
```

```

-----
-- Transformaciones elementales
-----

-- Ejercicio 20. Definir la función
--   intercambiaFilas :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> intercambiaFilas 1 3 p
--   ( 4 6 9 )
--   ( 3 2 6 )
--   ( 5 1 0 )
-----

```

```

intercambiaFilas :: Num a => Int -> Int -> Matrix a -> Matrix a
intercambiaFilas = switchRows

```

```

-----
-- Ejercicio 21. Definir la función
--   intercambiaColumnas :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (intercambiaColumnas k l p) es la matriz obtenida
-- intercambiando las columnas k y l de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> intercambiaColumnas 1 3 p
--   ( 0 1 5 )
--   ( 6 2 3 )
--   ( 9 6 4 )
-----

```

```

intercambiaColumnas :: Num a => Int -> Int -> Matrix a -> Matrix a
intercambiaColumnas = switchCols

```

```

-----
-- Ejercicio 22. Definir la función
--   multFilaPor :: Num a => Int -> a -> Matrix a -> Matrix a
-- tal que (multFilaPor k x p) es la matriz obtenida multiplicando la
-- fila k de la matriz p por el número x. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-----

```

```
-- ghci> multFilaPor 2 3 p
-- ( 5 1 0 )
-- ( 9 6 18 )
-- ( 4 6 9 )
```

```
-----
multFilaPor :: Num a => Int -> a -> Matrix a -> Matrix a
multFilaPor k x p = scaleRow x k p
```

```
-----
-- Ejercicio 23. Definir la función
```

```
-- sumaFilaFila :: Num a => Int -> Int -> Matrix a -> Matrix a
-- tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
-- a la fila k d la matriz p. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> sumaFilaFila 2 3 p
-- ( 5 1 0 )
-- ( 7 8 15 )
-- ( 4 6 9 )
```

```
-----
sumaFilaFila :: Num a => Int -> Int -> Matrix a -> Matrix a
sumaFilaFila k l p = combineRows k 1 l p
```

```
-----
-- Ejercicio 24. Definir la función
```

```
-- sumaFilaPor :: Num a => Int -> Int -> a -> Matrix a -> Matrix a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> sumaFilaPor 2 3 10 p
-- ( 5 1 0 )
-- ( 43 62 96 )
-- ( 4 6 9 )
```

```
-----
sumaFilaPor :: Num a => Int -> Int -> a -> Matrix a -> Matrix a
sumaFilaPor k l x p = combineRows k x l p
```

```
-- Triangularización de matrices
-- -----
--
-- Ejercicio 25. Definir la función
--   buscaIndiceDesde :: (Num a, Eq a) =>
--                       Matrix a -> Int -> Int -> Maybe Int
-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> buscaIndiceDesde p 3 2
--   Just 2
--   ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--   ghci> buscaIndiceDesde q 3 2
--   Nothing
```

```
buscaIndiceDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
  | null xs    = Nothing
  | otherwise  = Just (head xs)
  where xs = [k | k <- [i..nrows p], getElem k j p /= 0]
```

```
-- Ejercicio 26. Definir la función
--   buscaPivoteDesde :: (Num a, Eq a) =>
--                       Matrix a -> Int -> Int -> Maybe a
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> buscaPivoteDesde p 3 2
--   Just 6
--   ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--   ghci> buscaPivoteDesde q 3 2
--   Nothing
```

```
buscaPivoteDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
```

```

| null xs    = Nothing
| otherwise = Just (head xs)
where xs = [y | k <- [i..nrows p], let y = getElem k j p, y /= 0]

```

```

-----
-- Ejercicio 27. Definir la función
--   anuladaColumnaDesde :: (Num a, Eq a) =>
--                           Int -> Int -> Matrix a -> Bool
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
--   ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--   ghci> anuladaColumnaDesde q 3 2
--   True
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> anuladaColumnaDesde p 3 2
--   False

```

```

anuladaColumnaDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
  buscaIndiceDesde p j (i+1) == Nothing

```

```

-----
-- Ejercicio 28. Definir la función
--   anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
--                           Matrix a -> Int -> Int -> Matrix a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
--   ghci> let p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matrix Double
--   ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
--   [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]

```

```

anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
  Matrix a -> Int -> Int -> Matrix a
anulaEltoColumnaDesde p j i =
  sumaFilaPor l i (-(p!(l,j)/a)) p
  where Just l = buscaIndiceDesde p j (i+1)

```


$$a = p!(i, j)$$

```

-----
-- Ejercicio 29. Definir la función
--   anulaColumnaDesde :: (Fractional a, Eq a) =>
--                       Matrix a -> Int -> Int -> Matrix a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando
-- todos los elementos de la columna j de la matriz p por debajo de la
-- posición (i,j) (se supone que el elemento p_(i,j) es no nulo). Por
-- ejemplo,
--   ghci> let p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matrix Double
--   ghci> matrizLista (anulaColumnaDesde p 2 1)
--   [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
--   ghci> let p = listaMatriz [[4,5],[2,7%2],[6,10]]
--   ghci> matrizLista (anulaColumnaDesde p 1 1)
--   [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
-----

```

```

anulaColumnaDesde :: (Fractional a, Eq a) =>
                    Matrix a -> Int -> Int -> Matrix a
anulaColumnaDesde p j i
  | anuladaColumnaDesde p j i = p
  | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i

```

```

-----
-- Algoritmo de Gauss para triangularizar matrices
-----

```

```

-----
-- Ejercicio 30. Definir la función
--   elementosNoNulosColDesde :: (Num a, Eq a) =>
--                               Matrix a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
--   ghci> let p = listaMatriz [[3,2],[5,1],[0,4]]
--   ghci> elementosNoNulosColDesde p 1 2
--   [5]
-----

```

```

elementosNoNulosColDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> [a]

```

```

elementosNoNulosColDesde p j i =
  [y | k <- [i..nrows p], let y = getElem k j p, y /= 0]

-----
-- Ejercicio 31. Definir la función
--   existeColNoNulaDesde :: (Num a, Eq a) =>
--                           Matrix a -> Int -> Int -> Bool
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la j; es decir, si la submatriz de p obtenida eliminando
-- las i-1 primeras filas y las j-1 primeras columnas es no nula. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--   ghci> existeColNoNulaDesde p 2 2
--   False
--   ghci> let q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--   ghci> existeColNoNulaDesde q 2 2
--   True
-----

existeColNoNulaDesde :: (Num a, Eq a) => Matrix a -> Int -> Int -> Bool
existeColNoNulaDesde p j i =
  or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
  where n = numColumnas p

-- 2ª solución
existeColNoNulaDesde2 :: (Num a, Eq a) => Matrix a -> Int -> Int -> Bool
existeColNoNulaDesde2 p j i =
  submatrix i m j n p /= zero (m-i+1) (n-j+1)
  where (m,n) = dimension p

-----
-- Ejercicio 32. Definir la función
--   menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
--                           Matrix a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
--   ghci> let p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--   ghci> menorIndiceColNoNulaDesde p 2 2

```

```
-- Just 2
-- ghci> let q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
-- ghci> menorIndiceColNoNulaDesde q 2 2
-- Just 3
-- ghci> let r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
-- ghci> menorIndiceColNoNulaDesde r 2 2
-- Nothing
```

```
menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
    Matrix a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
  | null js    = Nothing
  | otherwise  = Just (head js)
  where n     = numColumnas p
        js   = [j' | j' <- [j..n],
                    not (null (elementosNoNulosColDesde p j' i))]
```

```
-- -----
-- Ejercicio 33. Definir la función
-- gaussAux :: (Fractional a, Eq a) =>
--           Matrix a -> Int -> Int -> Matrix a
-- tal que (gaussAux p i j) es la matriz que en el que las i-1 primeras
-- filas y las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
-- 1. Si la dimensión de p es (i,j), entonces p.
-- 2. Si la submatriz de p sin las i-1 primeras filas y las j-1
-- primeras columnas es nulas, entonces p.
-- 3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
-- 3.1. j' la primera columna a partir de la j donde p tiene
-- algún elemento no nulo a partir de la fila i,
-- 3.2. p1 la matriz obtenida intercambiando las columnas j y j'
-- de p,
-- 3.3. i' la primera fila a partir de la i donde la columna j de
-- p1 tiene un elemento no nulo,
-- 3.4. p2 la matriz obtenida intercambiando las filas i e i' de
-- la matriz p1 y
-- 3.5. p' la matriz obtenida anulando todos los elementos de la
-- columna j de p2 por debajo de la fila i.
-- Por ejemplo,
```

```
-- ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
-- ghci> gaussAux p 2 2
-- ( 1.0 2.0 3.0 )
-- ( 1.0 2.0 4.0 )
-- ( 2.0 0.0 1.0 )
```

```
-----
gaussAux :: (Fractional a, Eq a) => Matrix a -> Int -> Int -> Matrix a
gaussAux p i j
  | dimension p == (i,j)           = p                -- 1
  | not (existeColNoNulaDesde p j i) = p                -- 2
  | otherwise                       = gaussAux p' (i+1) (j+1) -- 3
  where Just j' = menorIndiceColNoNulaDesde p j i      -- 3.1
        p1     = intercambiaColumnas j j' p           -- 3.2
        Just i' = buscaIndiceDesde p1 j i             -- 3.3
        p2     = intercambiaFilas i i' p1             -- 3.4
        p'     = anulaColumnaDesde p2 j i             -- 3.5
```

```
-----
-- Ejercicio 34. Definir la función
-- gauss :: (Fractional a, Eq a) => Matrix a -> Matrix a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
-- ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
-- ghci> gauss p
-- ( 1.0 3.0 2.0 )
-- ( 0.0 1.0 0.0 )
-- ( 0.0 0.0 0.0 )
-- ghci> let p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
-- ghci> gauss p
-- ( 3 % 1 2 % 1 3 % 1 )
-- ( 0 % 1 4 % 3 3 % 1 )
-- ( 0 % 1 0 % 1 1 % 1 )
-- ghci> let p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
-- ghci> gauss p
-- ( 1.0 3.0 0.0 )
-- ( 0.0 1.0 0.0 )
-- ( 0.0 0.0 0.0 )
-----
```

```
gauss :: (Fractional a, Eq a) => Matrix a -> Matrix a
gauss p = gaussAux p 1 1
```

```
-----
-- Determinante                                     --
-----
```

```
-----
-- Ejercicio 35. Definir la función
```

```
gaussCAux :: (Fractional a, Eq a) =>
  Matrix a -> Int -> Int -> Int -> Matrix a
-- tal que (gaussCAux p i j c) es el par (n,q) donde q es la matriz que
-- en el que las i-1 primeras filas y las j-1 primeras columnas son las
-- de p y las restantes están triangularizadas por el método de Gauss;
-- es decir,
-- 1. Si la dimensión de p es (i,j), entonces p.
-- 2. Si la submatriz de p sin las i-1 primeras filas y las j-1
-- primeras columnas es nulas, entonces p.
-- 3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
-- 3.1. j' la primera columna a partir de la j donde p tiene
-- algún elemento no nulo a partir de la fila i,
-- 3.2. p1 la matriz obtenida intercambiando las columnas j y j'
-- de p,
-- 3.3. i' la primera fila a partir de la i donde la columna j de
-- p1 tiene un elemento no nulo,
-- 3.4. p2 la matriz obtenida intercambiando las filas i e i' de
-- la matriz p1 y
-- 3.5. p' la matriz obtenida anulando todos los elementos de la
-- columna j de p2 por debajo de la fila i.
-- y n es c más el número de intercambios de columnas y filas que se han
-- producido durante el cálculo. Por ejemplo,
ghci> gaussCAux (fromLists [[1.0,2,3],[1,2,4],[1,2,5]]) 1 1 0
(1,( 1.0 3.0 2.0 )
 ( 0.0 1.0 0.0 )
 ( 0.0 0.0 0.0 ))
-----
```

```
gaussCAux :: (Fractional a, Eq a) =>
  Matrix a -> Int -> Int -> Int -> (Int,Matrix a)
gaussCAux p i j c
```

```

| dimension p == (i,j)           = (c,p)           -- 1
| not (existeColNoNulaDesde p j i) = (c,p)         -- 2
| otherwise                       = gaussCAux p' (i+1) (j+1) c' -- 3
where Just j' = menorIndiceColNoNulaDesde p j i    -- 3.1
      p1      = switchCols j j' p                 -- 3.2
      Just i' = buscaIndiceDesde p1 j i           -- 3.3
      p2      = switchRows i i' p1               -- 3.4
      p'      = anulaColumnaDesde p2 j i         -- 3.5
      c'      = c + signum (abs (j-j')) + signum (abs (i-i'))

-----
-- Ejercicio 36. Definir la función
--   gaussC :: (Fractional a, Eq a) => Matriz a -> Matriz a
-- tal que (gaussC p) es el par (n,q), donde q es la triangularización
-- de la matriz p por el método de Gauss y n es el número de
-- intercambios de columnas y filas que se han producido durante el
-- cálculo. Por ejemplo,
--   ghci> gaussC (fromLists [[1.0,2,3],[1,2,4],[1,2,5]])
--   (1, ( 1.0 3.0 2.0 )
--        ( 0.0 1.0 0.0 )
--        ( 0.0 0.0 0.0 )
-----

gaussC :: (Fractional a, Eq a) => Matriz a -> (Int,Matriz a)
gaussC p = gaussCAux p 1 1 0

-----
-- Ejercicio 37. Definir la función
--   determinante :: (Fractional a, Eq a) => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--   ghci> determinante (fromLists [[1.0,2,3],[1,3,4],[1,2,5]])
--   2.0
-----

determinante :: (Fractional a, Eq a) => Matriz a -> a
determinante p = (-1)^c * V.product (getDiag p')
  where (c,p') = gaussC p

```

Relación 23

Problemas básicos con el TAD de los grafos

23.1. Ejercicios propuestos

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir funciones sobre  
-- el TAD de los grafos estudiado en el tema 22 que se pueden descargar desde  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/codigos/I1M2014.zip  
--  
-- Las transparencias del tema se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-22.pdf  
  
-----  
-- Importación de librerías --  
-----  
  
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}  
  
import Data.Array  
import Data.List (nub)  
import Test.QuickCheck  
import I1M.Grafo -- Se instala con http://bit.ly/1AKmUQB  
  
-- En lugar de I1M.Grafos, se puede seleccionar una implementación del
```

```

-- TAD de los grafos
--   import GrafoConVectorDeAdyacencia
--   import GrafoConMatrizDeAdyacencia

-----
-- Ejemplos
-----

-- Para los ejemplos se usarán los siguientes grafos.
g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 :: Grafo Int Int
g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                        (2,4,55),(2,5,32),
                        (3,4,61),(3,5,44),
                        (4,5,93)]
g2 = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                        (2,4,55),(2,5,32),
                        (4,3,61),(4,5,93)]
g3 = creaGrafo D (1,3) [(1,2,0),(2,2,0),(3,1,0),(3,2,0)]
g4 = creaGrafo D (1,4) [(1,2,3),(2,1,5)]
g5 = creaGrafo D (1,1) [(1,1,0)]
g6 = creaGrafo D (1,4) [(1,3,0),(3,1,0),(3,3,0),(4,2,0)]
g7 = creaGrafo ND (1,4) [(1,3,0)]
g8 = creaGrafo D (1,5) [(1,1,0),(1,2,0),(1,3,0),(2,4,0),(3,1,0),
                        (4,1,0),(4,2,0),(4,4,0),(4,5,0)]
g9 = creaGrafo D (1,5) [(4,1,1),(4,3,2),(5,1,0)]
g10 = creaGrafo ND (1,3) [(1,2,1),(1,3,1),(2,3,1),(3,3,1)]
g11 = creaGrafo D (1,3) [(1,2,1),(1,3,1),(2,3,1),(3,3,1)]
g12 = creaGrafo ND (1,4) [(1,1,0),(1,2,0),(3,3,0)]

-----
-- Ejercicio 1. El grafo completo de orden  $n$ ,  $K(n)$ , es un grafo no
-- dirigido cuyo conjunto de vértices es  $\{1,..n\}$  y tiene una arista
-- entre cada par de vértices distintos. Definir la función,
--   completo :: Int -> Grafo Int Int
-- tal que (completo  $n$ ) es el grafo completo de orden  $n$ . Por ejemplo,
--   ghci> completo 4
--   G ND (array (1,4) [(1,[(2,0),(3,0),(4,0)]),
--                     (2,[(1,0),(3,0),(4,0)]),
--                     (3,[(1,0),(2,0),(4,0)]),
--                     (4,[(1,0),(2,0),(3,0)])])

```



```

-----

-- 1ª definición
completo1 :: Int -> Grafo Int Int
completo1 n = creaGrafo ND (1,n) xs
  where xs = [(x,y,0) | x <- [1..n], y <- [1..n], x < y]

-- 2ª definición
completo2 :: Int -> Grafo Int Int
completo2 n = creaGrafo ND (1,n) [(a,b,0) | a <- [1..n], b <- [1..a-1]]

-- La 2ª es más eficiente
-- ghci> length (nodos (completo 2000))
-- 2000
-- (7.21 secs, 1137479656 bytes)
-- ghci> length (nodos (completo' 2000))
-- 2000
-- (4.81 secs, 785507504 bytes)

-- Usaremos la 2ª
completo :: Int -> Grafo Int Int
completo = completo2

-----

-- Ejercicio 2. El ciclo de orden n, C(n), es un grafo no dirigido
-- cuyo conjunto de vértices es {1,...,n} y las aristas son
-- (1,2), (2,3), ..., (n-1,n), (n,1)
-- Definir la función
-- grafoCiclo :: Int -> Grafo Int Int
-- tal que (grafoCiclo n) es el grafo ciclo de orden n. Por ejemplo,
-- ghci> grafoCiclo 3
-- G ND (array (1,3) [(1,[(3,0),(2,0)]),(2,[(1,0),(3,0)]),(3,[(2,0),(1,0)])])
-----

grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n = creaGrafo ND (1,n) xs
  where xs = [(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)]

-----

-- Ejercicio 3. Definir la función

```

```

--      nVertices :: (Ix v,Num p) => Grafo v p -> Int
-- tal que (nVertices g) es el número de vértices del grafo g. Por
-- ejemplo,
--      nVertices (completo 4) == 4
--      nVertices (completo 5) == 5

```

```

nVertices :: (Ix v,Num p) => Grafo v p -> Int
nVertices = length . nodos

```

```

-----
-- Ejercicio 4. Definir la función
--      noDirigido :: (Ix v,Num p) => Grafo v p -> Bool
-- tal que (noDirigido g) se verifica si el grafo g es no dirigido. Por
-- ejemplo,
--      noDirigido g1           == True
--      noDirigido g2           == False
--      noDirigido (completo 4) == True

```

```

noDirigido :: (Ix v,Num p) => Grafo v p -> Bool
noDirigido = not . dirigido

```

```

-----
-- Ejercicio 5. En un un grafo g, los incidentes de un vértice v es el
-- conjuntos de vértices x de g para los que hay un arco (o una arista)
-- de x a v; es decir, que v es adyacente a x. Definir la función
--      incidentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
-- tal que (incidentes g v) es la lista de los vértices incidentes en el
-- vértice v. Por ejemplo,
--      incidentes g2 5 == [1,2,4]
--      adyacentes g2 5 == []
--      incidentes g1 5 == [1,2,3,4]
--      adyacentes g1 5 == [1,2,3,4]

```

```

incidentes :: (Ix v,Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

```

```
-- Ejercicio 6. En un un grafo g, los contiguos de un vértice v es el
-- conjuntos de vértices x de g tales que x es adyacente o incidente con
-- v. Definir la función
--   contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
-- tal que (contiguos g v) es el conjunto de los vértices de g contiguos
-- con el vértice v. Por ejemplo,
--   contiguos g2 5 == [1,2,4]
--   contiguos g1 5 == [1,2,3,4]
```

```
contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
contiguos g v = nub (adyacentes g v ++ incidentes g v)
```

```
-- Ejercicio 7. Definir la función
--   lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
-- tal que (lazos g) es el conjunto de los lazos (es decir, aristas
-- cuyos extremos son iguales) del grafo g. Por ejemplo,
--   ghci> lazos g3
--   [(2,2)]
--   ghci> lazos g2
--   []
```

```
lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
lazos g = [(x,x) | x <- nodos g, aristaEn g (x,x)]
```

```
-- Ejercicio 8. Definir la función
--   nLazos :: (Ix v,Num p) => Grafo v p -> Int
-- tal que (nLazos g) es el número de lazos del grafo g. Por
-- ejemplo,
--   nLazos g3 == 1
--   nLazos g2 == 0
```

```
nLazos :: (Ix v,Num p) => Grafo v p -> Int
nLazos = length . lazos
```

```

-- Ejercicio 9. Definir la función
--   nAristas :: (Ix v, Num p) => Grafo v p -> Int
-- tal que (nAristas g) es el número de aristas del grafo g. Si g es no
-- dirigido, las aristas de v1 a v2 y de v2 a v1 sólo se cuentan una
-- vez. Por ejemplo,
--   nAristas g1           == 8
--   nAristas g2           == 7
--   nAristas g10          == 4
--   nAristas g12          == 3
--   nAristas (completo 4) == 6
--   nAristas (completo 5) == 10

```

```

nAristas :: (Ix v, Num p) => Grafo v p -> Int
nAristas g | dirigido g = length $ aristas g
           | otherwise  = (length (aristas g) + nLazos g) 'div' 2

```

-- 2ª definición

```

nAristas2 :: (Ix v, Num p) => Grafo v p -> Int
nAristas2 g | dirigido g = length (aristas g)
           | otherwise  = length [(x,y) | (x,y,_) <- aristas g, x <= y]

```

```

-- Ejercicio 10. Definir la función
--   prop_nAristasCompleto :: Int -> Bool
-- tal que (prop_nAristasCompleto n) se verifica si el número de aristas
-- del grafo completo de orden n es  $n*(n-1)/2$  y, usando la función,
-- comprobar que la propiedad se cumple para n de 1 a 20.

```

```

prop_nAristasCompleto :: Int -> Bool
prop_nAristasCompleto n =
  nAristas (completo n) == n*(n-1) 'div' 2

```

-- La comprobación es

```

--   ghci> and [prop_nAristasCompleto n | n <- [1..20]]
--   True

```

```

-- Ejercicio 11. El grado positivo de un vértice v de un grafo dirigido

```

```

-- g, es el número de vértices de g adyacentes con v. Definir la función
-- gradoPos :: (Ix v, Num p) => Grafo v p -> v -> Int
-- tal que (gradoPos g v) es el grado positivo del vértice v en el grafo
-- g. Por ejemplo,
-- gradoPos g1 5 == 4
-- gradoPos g2 5 == 0
-- gradoPos g2 1 == 3

```

```

gradoPos :: (Ix v, Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)

```

```

-- Ejercicio 12. El grado negativo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g incidentes con v. Definir la función
-- gradoNeg :: (Ix v, Num p) => Grafo v p -> v -> Int
-- tal que (gradoNeg g v) es el grado negativo del vértice v en el grafo
-- g. Por ejemplo,
-- gradoNeg g1 5 == 4
-- gradoNeg g2 5 == 3
-- gradoNeg g2 1 == 0

```

```

gradoNeg :: (Ix v, Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)

```

```

-- Ejercicio 13. El grado de un vértice v de un grafo dirigido g, es el
-- número de aristas de g que contiene a v. Si g es no dirigido, el
-- grado de un vértice v es el número de aristas incidentes en v, teniendo
-- en cuenta que los lazos se cuentan dos veces. Definir la función
-- grado :: (Ix v, Num p) => Grafo v p -> v -> Int
-- tal que (grado g v) es el grado del vértice v en el grafo g. Por
-- ejemplo,
-- grado g1 5 == 4
-- grado g2 5 == 3
-- grado g2 1 == 3
-- grado g3 2 == 4
-- grado g3 1 == 2
-- grado g3 3 == 2

```

```
-- grado g5 1 == 2
-- grado g10 3 == 4
-- grado g11 3 == 4
```

```
-----
grado :: (Ix v, Num p) => Grafo v p -> v -> Int
grado g v | dirigido g           = gradoNeg g v + gradoPos g v
          | (v,v) 'elem' lazos g = length (incidentes g v) + 1
          | otherwise            = length (incidentes g v)
```

```
-----
-- Ejercicio 14. Comprobar con QuickCheck que para cualquier grafo g, la
-- suma de los grados positivos de los vértices de g es igual que la
-- suma de los grados negativos de los vértices de g.
-----
```

```
-- La propiedad es
prop_sumaGrados :: Grafo Int Int -> Bool
prop_sumaGrados g =
  sum [gradoPos g v | v <- vs] == sum [gradoNeg g v | v <- vs]
  where vs = nodos g
```

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

```
-----
-- Ejercicio 15. En la teoría de grafos, se conoce como "Lema del
-- apretón de manos" la siguiente propiedad: la suma de los grados de
-- los vértices de g es el doble del número de aristas de g.
-- Comprobar con QuickCheck que para cualquier grafo g, se verifica
-- dicha propiedad.
-----
```

```
prop_apretonManos :: Grafo Int Int -> Bool
prop_apretonManos g =
  sum [grado g v | v <- nodos g] == 2 * nAristas g
```

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

```

--      +++ OK, passed 100 tests.

-----

-- Ejercicio 16. Comprobar con QuickCheck que en todo grafo, el número
-- de nodos de grado impar es par.
-----

prop_numNodosGradoImpar :: Grafo Int Int -> Bool
prop_numNodosGradoImpar g = even m
  where vs = nodos g
        m = length [v | v <- vs, odd(grado g v)]

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

-----

-- Ejercicio 17. Definir la propiedad
-- prop_GradoCompleto :: Int -> Bool
-- tal que (prop_GradoCompleto n) se verifica si todos los vértices del
-- grafo completo  $K(n)$  tienen grado  $n-1$ . Usarla para comprobar que dicha
-- propiedad se verifica para los grafos completos de grados 1 hasta 30.
-----

prop_GradoCompleto :: Int -> Bool
prop_GradoCompleto n =
  and [grado g v == (n-1) | v <- nodos g]
  where g = completo n

-- La comprobación es
-- ghci> and [prop_GradoCompleto n | n <- [1..30]]
--      True

-----

-- Ejercicio 18. Un grafo es regular si todos sus vértices tienen el
-- mismo grado. Definir la función
-- regular :: (Ix v, Num p) => Grafo v p -> Bool
-- tal que (regular g) se verifica si todos los nodos de g tienen el
-- mismo grado.
-- regular g1 == False

```

```

--      regular g2                == False
--      regular (completo 4)     == True
-----

regular :: (Ix v, Num p) => Grafo v p -> Bool
regular g = and [grado g v == k | v <- vs]
  where vs = nodos g
        k  = grado g (head vs)
-----

-- Ejercicio 19. Definir la propiedad
--   prop_CompletoRegular :: Int -> Int -> Bool
-- tal que (prop_CompletoRegular m n) se verifica si todos los grafos
-- completos desde el de orden m hasta el de orden n son regulares y
-- usarla para comprobar que todos los grafos completo desde el de orden
-- 1 hasta el de orden 30 son regulares.
-----

prop_CompletoRegular :: Int -> Int -> Bool
prop_CompletoRegular m n = and [regular (completo x) | x <- [m..n]]

-- La comprobación es
--   ghci> prop_CompletoRegular 1 30
--   True
-----

-- Ejercicio 20. Un grafo es k-regular si todos sus vértices son de
-- grado k. Definir la función
--   regularidad :: (Ix v, Num p) => Grafo v p -> Maybe Int
-- tal que (regularidad g) es la regularidad de g. Por ejemplo,
--   regularidad g1                == Nothing
--   regularidad (completo 4)      == Just 3
--   regularidad (completo 5)     == Just 4
--   regularidad (grafoCiclo 4)   == Just 2
--   regularidad (grafoCiclo 5)   == Just 2
-----

regularidad :: (Ix v, Num p) => Grafo v p -> Maybe Int
regularidad g | regular g = Just (grado g (head (nodos g)))
              | otherwise = Nothing

```



```
-----  
-- Ejercicio 21. Definir la propiedad  
--   prop_completoRegular :: Int -> Bool  
-- tal que (prop_completoRegular n) se verifica si el grafo completo de  
-- orden n es (n-1)-regular. Por ejemplo,  
--   prop_completoRegular 5 == True  
-- y usarla para comprobar que la cumplen todos los grafos completos  
-- desde orden 1 hasta 20.  
-----  
  
prop_completoRegular :: Int -> Bool  
prop_completoRegular n =  
  regularidad (completo n) == Just (n-1)  
  
-- La comprobación es  
--   ghci> and [prop_completoRegular n | n <- [1..20]]  
--   True  
  
-----  
-- Ejercicio 22. Definir la propiedad  
--   prop_cicloRegular :: Int -> Bool  
-- tal que (prop_cicloRegular n) se verifica si el grafo ciclo de orden  
-- n es 2-regular. Por ejemplo,  
--   prop_cicloRegular 2 == True  
-- y usarla para comprobar que la cumplen todos los grafos ciclos  
-- desde orden 3 hasta 20.  
-----  
  
prop_cicloRegular :: Int -> Bool  
prop_cicloRegular n =  
  regularidad (grafoCiclo n) == Just 2  
  
-- La comprobación es  
--   ghci> and [prop_cicloRegular n | n <- [3..20]]  
--   True  
  
-----  
-- § Generador de grafos  
-----
```

```

-- (generaGND n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
-- ghci> generaGND 3 [4,2,5]
-- (ND,array (1,3) [(1,[(2,4),(3,2)]),
--                  (2,[(1,4),(3,5)]),
--                  3,[(1,2),(2,5)]])
-- ghci> generaGND 3 [4,-2,5]
-- (ND,array (1,3) [(1,[(2,4)]), (2,[(1,4),(3,5)]), (3,[(2,5)])])
generaGND :: Int -> [Int] -> Grafo Int Int
generaGND n ps = creaGrafo ND (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n], x < y]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- (generaGD n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
-- ghci> generaGD 3 [4,2,5]
-- (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                 (2,[]),
--                 (3,[])])
-- ghci> generaGD 3 [4,2,5,3,7,9,8,6]
-- (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                 (2,[(1,3),(2,7),(3,9)]),
--                 (3,[(1,8),(2,6)])])
generaGD :: Int -> [Int] -> Grafo Int Int
generaGD n ps = creaGrafo D (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n]]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- genGD es un generador de grafos dirigidos. Por ejemplo,
-- ghci> sample genGD
-- (D,array (1,4) [(1,[(1,1)]), (2,[(3,1)]), (3,[(2,1),(4,1)]), (4,[(4,1)])])
-- (D,array (1,2) [(1,[(1,6)]), (2,[])])
-- ...
genGD :: Gen (Grafo Int Int)
genGD = do n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          return (generaGD n xs)

```

```

-- genGND es un generador de grafos dirigidos. Por ejemplo,
--   ghci> sample genGND
--   (ND,array (1,1) [(1,[])])
--   (ND,array (1,3) [(1,[(2,3),(3,13)]),(2,[(1,3)]),(3,[(1,13)])])
--   ...
genGND :: Gen (Grafo Int Int)
genGND = do n <- choose (1,10)
           xs <- vectorOf (n*n) arbitrary
           return (generaGND n xs)

-- genG es un generador de grafos. Por ejemplo,
--   ghci> sample genG
--   (D,array (1,3) [(1,[(2,1)]),(2,[(1,1),(2,1)]),(3,[(3,1)])])
--   (ND,array (1,3) [(1,[(2,2)]),(2,[(1,2)]),(3,[])])
--   ...
genG :: Gen (Grafo Int Int)
genG = do d <- choose (True,False)
          n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          if d then return (generaGD n xs)
            else return (generaGND n xs)

-- Los grafos está contenido en la clase de los objetos generables
-- aleatoriamente.
instance Arbitrary (Grafo Int Int) where
  arbitrary = genG

```

23.2. Ejercicios de exámenes

```

-----
-- Introducción
-----

-- En esta relación se presenta una recopilación de ejercicios sobre
-- grafos propuestos en exámenes de la asignatura.

-----
-- § Librerías auxiliares
-----

```

```

import Data.List
import Data.Array
import I1M.Grafo

```

```

-- -----
-- Ejercicio 1. Definir la función
--   recorridos :: [a] -> [[a]]
-- tal que (recorridos xs) es la lista de todos los posibles recorridos
-- por el grafo cuyo conjunto de vértices es xs y cada vértice se
-- encuentra conectado con todos los otros y los recorridos pasan por
-- todos los vértices una vez y terminan en el vértice inicial. Por
-- ejemplo,
--   ghci> recorridos [2,5,3]
--   [[2,5,3,2],[5,2,3,5],[3,5,2,3],[5,3,2,5],[3,2,5,3],[2,3,5,2]]
-- Indicación: No importa el orden de los recorridos en la lista.
-- -----

```

```

recorridos :: [a] -> [[a]]
recorridos xs = [(y:ys)++[y] | (y:ys) <- permutations xs]

```

```

-- -----
-- Ejercicio 2.1. Consideremos un grafo  $G = (V,E)$ , donde  $V$  es un
-- conjunto finito de nodos ordenados y  $E$  es un conjunto de arcos. En un
-- grafo, la anchura de un nodo es el máximo de los valores absolutos de
-- la diferencia entre el valor del nodo y los de sus adyacentes; y la
-- anchura del grafo es la máxima anchura de sus nodos. Por ejemplo, en
-- el grafo
--   grafo2 :: Grafo Int Int
--   grafo2 = creaGrafo ND (1,5) [(1,2,1),(1,3,1),(1,5,1),
--                                (2,4,1),(2,5,1),
--                                (3,4,1),(3,5,1),
--                                (4,5,1)]
-- su anchura es 4 y el nodo de máxima anchura es el 5.
--
-- Definir la función
--   anchura :: Grafo Int Int -> Int
-- tal que (anchuraG g) es la anchura del grafo g. Por ejemplo,
--   anchura g == 4

```

```

-----

grafo2 :: Grafo Int Int
grafo2 = creaGrafo ND (1,5) [(1,2,1),(1,3,1),(1,5,1),
                             (2,4,1),(2,5,1),
                             (3,4,1),(3,5,1),
                             (4,5,1)]

anchura :: Grafo Int Int -> Int
anchura g = maximum [anchuraN g x | x <- nodos g]

-- (anchuraN g x) es la anchura del nodo x en el grafo g. Por ejemplo,
--   anchuraN g 1 == 4
--   anchuraN g 2 == 3
--   anchuraN g 4 == 2
--   anchuraN g 5 == 4
anchuraN :: Grafo Int Int -> Int -> Int
anchuraN g x = maximum (0 : [abs (x-v) | v <- adyacentes g x])

-----

-- Ejercicio 2.2. Comprobar experimentalmente que la anchura del grafo
-- grafo cíclico de orden n es n-1.
-----

-- La conjetura
conjetura :: Int -> Bool
conjetura n = anchura (grafoCiclo n) == n-1

-- (grafoCiclo n) es el grafo cíclico de orden n. Por ejemplo,
--   ghci> grafoCiclo 4
--   G ND (array (1,4) [(1,[(4,0),(2,0)]),(2,[(1,0),(3,0)]),
--                    (3,[(2,0),(4,0])),(4,[(3,0),(1,0)])])
grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n = creaGrafo ND (1,n) xs
  where xs = [(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)]

-- La comprobación es
--   ghci> and [conjetura n | n <- [2..10]]
--   True

```

```

-----
-- Ejercicio 3. Un grafo no dirigido G se dice conexo, si para cualquier
-- par de vértices u y v en G, existe al menos una trayectoria (una
-- sucesión de vértices adyacentes) de u a v.
--
-- Definirla función
--   conexo :: (Ix a, Num p) => Grafo a p -> Bool
-- tal que (conexo g) se verifica si el grafo g es conexo. Por ejemplo,
--   conexo (creaGrafo ND (1,3) [(1,2,0),(3,2,0)]) == True
--   conexo (creaGrafo ND (1,4) [(1,2,0),(3,4,0)]) == False
-----

```

```

conexo :: (Ix a, Num p) => Grafo a p -> Bool
conexo g = length (recorridoEnAnchura i g) == n
  where xs = nodos g
        i  = head xs
        n  = length xs

```

```

-- (recorridoEnAnchura i g) es el recorrido en anchura del grafo g
-- desde el vértice i, usando colas. Por ejemplo,
--   recorridoEnAnchura 1 g == [1,4,3,2,6,5]
recorridoEnAnchura i g = reverse (ra [i] [])

```

```

where
  ra [] vis = vis
  ra (c:cs) vis
    | c 'elem' vis = ra cs vis
    | otherwise   = ra (cs ++ adyacentes g c) (c:vis)

```

```

-----
-- Ejercicio 4. Un mapa se puede representar mediante un grafo donde
-- los vértices son las regiones del mapa y hay una arista entre dos
-- vértices si las correspondientes regiones son vecinas. Por ejemplo,
-- el mapa siguiente

```

```

--   +-----+-----+
--   | 1     | 2     |
--   +---+---+---+---+
--   |  |   |   |   |
--   | 3   | 4   | 5   |
--   |  |   |   |   |
--   +---+---+---+---+

```

```

--      |   6   |   7   |
--      +-----+-----+
-- se pueden representar por
-- mapa :: Grafo Int Int
-- mapa = creaGrafo ND (1,7)
--           [(1,2,0),(1,3,0),(1,4,0),(2,4,0),(2,5,0),(3,4,0),
--           (3,6,0),(4,5,0),(4,6,0),(4,7,0),(5,7,0),(6,7,0)]
-- Para colorear el mapa se dispone de 4 colores definidos por
-- data Color = A | B | C | D deriving (Eq, Show)
--
-- Definir la función
-- correcta :: [(Int,Color)] -> Grafo Int Int -> Bool
-- tal que (correcta ncs m) se verifica si ncs es una coloración del
-- mapa m tal que todos las regiones vecinas tienen colores distintos.
-- Por ejemplo,
-- correcta [(1,A),(2,B),(3,B),(4,C),(5,A),(6,A),(7,B)] mapa == True
-- correcta [(1,A),(2,B),(3,A),(4,C),(5,A),(6,A),(7,B)] mapa == False
-----

```

```

mapa :: Grafo Int Int

```

```

mapa = creaGrafo ND (1,7)
           [(1,2,0),(1,3,0),(1,4,0),(2,4,0),(2,5,0),(3,4,0),
           (3,6,0),(4,5,0),(4,6,0),(4,7,0),(5,7,0),(6,7,0)]

```

```

data Color = A | B | C | E deriving (Eq, Show)

```

```

correcta :: [(Int,Color)] -> Grafo Int Int -> Bool

```

```

correcta ncs g =
  and [and [color x /= color y | y <- adyacentes g x] | x <- nodos g]
  where color x = head [c | (y,c) <- ncs, y == x]

```

```

-----
-- Ejercicio 5. Dado un grafo dirigido G, diremos que un nodo está
-- aislado si o bien de dicho nodo no sale ninguna arista o bien no
-- llega al nodo ninguna arista. Por ejemplo, en el siguiente grafo
-- (Tema 22, pag. 31)
-- grafo5 = creaGrafo D (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
--                             (5,4,0),(6,2,0),(6,5,0)]
-- podemos ver que del nodo 1 salen 3 aristas pero no llega ninguna, por
-- lo que lo consideramos aislado. Así mismo, a los nodos 2 y 4 llegan

```

```

-- aristas pero no sale ninguna, por tanto también estarán aislados.
--
-- Definir la función
--   aislados :: (Ix v, Num p) => Grafo v p -> [v]
-- tal que (aislados g) es la lista de nodos aislados del grafo g. Por
-- ejemplo,
--   aislados grafo5 == [1,2,4]
-----

grafo5 = creaGrafo D (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
                             (5,4,0),(6,2,0),(6,5,0)]

aislados :: (Ix v, Num p) => Grafo v p -> [v]
aislados g = [n | n <- nodos g, adyacentes g n == [] || incidentes g n == [] ]

-- (incidentes g v) es la lista de los nodos incidentes con v en el
-- grafo g. Por ejemplo,
--   incidentes g 2 == [1,6]
--   incidentes g 1 == []
incidentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

-----

-- Ejercicio 6. Consideremos una implementación del TAD de los grafos,
-- por ejemplo en la que los grafos se representan mediante listas. Un
-- ejemplo de grafo es el siguiente:
--   grafo6 :: Grafo Int Int
--   grafo6 = creaGrafo D (1,6) [(1,3,2),(1,5,4),(3,5,6),(5,1,8),(5,5,10),
--                               (2,4,1),(2,6,3),(4,6,5),(4,4,7),(6,4,9)]
--
-- Definir la función
--   conectados :: Grafo Int Int -> Int -> Int -> Bool
-- tal que (conectados g v1 v2) se verifica si los vértices v1 y v2
-- están conectados en el grafo g. Por ejemplo,
--   conectados grafo6 1 3 == True
--   conectados grafo6 1 4 == False
--   conectados grafo6 6 2 == False
--   conectados grafo6 2 6 == True
-----

```



```
grafo6 :: Grafo Int Int
```

```
grafo6 = creaGrafo D (1,6) [(1,3,2),(1,5,4),(3,5,6),(5,1,8),(5,5,10),  
                           (2,4,1),(2,6,3),(4,6,5),(4,4,7),(6,4,9)]
```

```
conectados :: Grafo Int Int -> Int -> Int -> Bool
```

```
conectados g v1 v2 = elem v2 (conectadosAux g [] [v1])
```

```
conectadosAux :: Grafo Int Int -> [Int] -> [Int] -> [Int]
```

```
conectadosAux g vs [] = vs
```

```
conectadosAux g vs (w:ws)
```

```
  | elem w vs = conectadosAux g vs ws
```

```
  | otherwise = conectadosAux g (union [w] vs) (union ws (adyacentes g w))
```


Relación 24

Implementación del TAD de los grafos mediante listas

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación es implementar el TAD de los grafos  
-- mediante listas, de manera análoga a las implementaciones estudiadas  
-- en el tema 22 que se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-22.pdf  
-- y usando la mismas signatura.  
  
-----  
-- Signatura --  
-----  
  
module Rel_26_sol  
  (Orientacion (...),  
   Grafo,  
   creaGrafo, -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->  
               -- Grafo v p  
   dirigido, -- (Ix v,Num p) => (Grafo v p) -> Bool  
   adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]  
   nodos, -- (Ix v,Num p) => (Grafo v p) -> [v]  
   aristas, -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]  
   aristaEn, -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool  
   peso -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
```

```

) where

-----
-- Librerías auxiliares                                     --
-----

import Data.Array
import Data.List

-----
-- Representación de los grafos mediante listas           --
-----

-- Orientación es D (dirigida) ó ND (no dirigida).
data Orientacion = D | ND
    deriving (Eq, Show)

-- (Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p.
data Grafo v p = G Orientacion ([v],[((v,v),p)])
    deriving (Eq, Show)

-----
-- Ejercicios                                             --
-----

-----
-- Ejercicio 1. Definir la función
--   creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
-- tal que (creaGrafo o cs as) es un grafo (dirigido o no, según el
-- valor de o), con el par de cotas cs y listas de aristas as (cada
-- arista es un tríó formado por los dos vértices y su peso). Por
-- ejemplo,
--   ghci> creaGrafo ND (1,3) [(1,2,12),(1,3,34)]
--   G ND ([1,2,3],[((1,2),12),((1,3),34),((2,1),12),((3,1),34)])
--   ghci> creaGrafo D (1,3) [(1,2,12),(1,3,34)]
--   G D ([1,2,3],[((1,2),12),((1,3),34)])
--   ghci> creaGrafo D (1,4) [(1,2,12),(1,3,34)]
--   G D ([1,2,3,4],[((1,2),12),((1,3),34)])
-----

```

```

creaGrafo :: (Ix v, Num p) =>
    Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs as =
    G o (range cs, [(x1,x2),w] | (x1,x2,w) <- as] ++
        if o == D then []
        else [(x2,x1),w] | (x1,x2,w) <- as, x1 /= x2])

```

```

-----
-- Ejercicio 2. Definir, con creaGrafo, la constante
-- ejGrafoND :: Grafo Int Int
-- para representar el siguiente grafo no dirigido

```

```

--
--      12
--      1 ----- 2
--      | \78    /|
--      |  \ 32/  |
--      |   \ /   |
--      34|    5   |55
--      |   /  \  |
--      |  /44  \ |
--      | /    93\|
--      3 ----- 4
--
--      61
-- ghci> ejGrafoND
-- G ND ([1,2,3,4,5],
--       [(1,2),12],[(1,3),34],[(1,5),78],[(2,4),55],[(2,5),32],
--       [(3,4),61],[(3,5),44],[(4,5),93],[(2,1),12],[(3,1),34],
--       [(5,1),78],[(4,2),55],[(5,2),32],[(4,3),61],[(5,3),44],
--       [(5,4),93]])

```

```

-----
ejGrafoND :: Grafo Int Int
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
    (2,4,55),(2,5,32),
    (3,4,61),(3,5,44),
    (4,5,93)]

```

```

-----
-- Ejercicio 3. Definir, con creaGrafo, la constante
-- ejGrafoD :: Grafo Int Int
-- para representar el grafo anterior donde se considera que las aristas

```

```
-- son los pares (x,y) con x < y. Por ejemplo,
-- ghci> ejGrafoD
-- G D ([1,2,3,4,5],
--      [((1,2),12),((1,3),34),((1,5),78),((2,4),55),((2,5),32),
--      ((3,4),61),((3,5),44),((4,5),93))])
-----
```

```
ejGrafoD :: Grafo Int Int
```

```
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                             (2,4,55),(2,5,32),
                             (3,4,61),(3,5,44),
                             (4,5,93)]
```

```
-- Ejercicio 4. Definir la función
-- dirigido :: (Ix v, Num p) => (Grafo v p) -> Bool
-- tal que (dirigido g) se verifica si g es dirigido. Por ejemplo,
-- dirigido ejGrafoD == True
-- dirigido ejGrafoND == False
-----
```

```
dirigido :: (Ix v, Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

```
-- Ejercicio 5. Definir la función
-- nodos :: (Ix v, Num p) => (Grafo v p) -> [v]
-- tal que (nodos g) es la lista de todos los nodos del grafo g. Por
-- ejemplo,
-- nodos ejGrafoND == [1,2,3,4,5]
-- nodos ejGrafoD == [1,2,3,4,5]
-----
```

```
nodos :: (Ix v, Num p) => (Grafo v p) -> [v]
nodos (G _ (ns, _)) = ns
```

```
-- Ejercicio 6. Definir la función
-- adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
-- tal que (adyacentes g v) es la lista de los vértices adyacentes al
```

```
-- nodo v en el grafo g. Por ejemplo,
--   adjacentes ejGrafoND 4 == [5,2,3]
--   adjacentes ejGrafoD  4 == [5]
```

```
-----
adjacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adjacentes (G _ (_,e)) v = nub [u | ((w,u),_) <- e, w == v]
```

```
-----
-- Ejercicio 7. Definir la función
--   aristaEn :: (Ix v, Num p) => Grafo v p -> (v,v) -> Bool
-- (aristaEn g a) se verifica si a es una arista del grafo g. Por
-- ejemplo,
--   aristaEn ejGrafoND (5,1) == True
--   aristaEn ejGrafoND (4,1) == False
--   aristaEn ejGrafoD  (5,1) == False
--   aristaEn ejGrafoD  (1,5) == True
```

```
-----
aristaEn :: (Ix v, Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adjacentes g x
```

```
-----
-- Ejercicio 8. Definir la función
--   peso :: (Ix v, Num p) => v -> v -> Grafo v p -> p
-- tal que (peso v1 v2 g) es el peso de la arista que une los vértices
-- v1 y v2 en el grafo g. Por ejemplo,
--   peso 1 5 ejGrafoND == 78
--   peso 1 5 ejGrafoD  == 78
```

```
-----
peso :: (Ix v, Num p) => v -> v -> Grafo v p -> p
peso x y (G _ (_,gs)) = head [c | ((x',y'),c) <- gs, x==x', y==y']
```

```
-----
-- Ejercicio 9. Definir la función
--   aristas :: (Ix v, Num p) => Grafo v p -> [(v,v,p)]
-- (aristasD g) es la lista de las aristas del grafo g. Por ejemplo,
--   ghci> aristas ejGrafoD
--   [(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
```

```
-- (3,5,44), (4,5,93)]
-- ghci> aristas ejGrafoND
-- [(1,2,12), (1,3,34), (1,5,78), (2,1,12), (2,4,55), (2,5,32),
-- (3,1,34), (3,4,61), (3,5,44), (4,2,55), (4,3,61), (4,5,93),
-- (5,1,78), (5,2,32), (5,3,44), (5,4,93)]
```

```
-----

aristas :: (Ix v, Num p) => Grafo v p -> [(v,v,p)]
aristas (G _ (_,g)) = [(v1,v2,p) | ((v1,v2),p) <- g]
```


Relación 25

Enumeraciones de los números racionales

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es construir dos enumeraciones de los  
-- números racionales. Concretamente,  
-- * una enumeración basada en las representaciones hiperbinarias y  
-- * una enumeración basada en los los árboles de Calkin-Wilf.  
-- También se incluye la comprobación de la igualdad de las dos  
-- sucesiones y una forma alternativa de calcular el número de  
-- representaciones hiperbinarias mediante la función fucs.  
--  
-- Esta relación se basa en los siguientes artículos:  
-- * Gaussianos "Sorpresa sumando potencias de 2" http://goo.gl/AHdAG  
-- * N. Calkin y H.S. Wilf "Recounting the rationals" http://goo.gl/gVZtW  
-- * Wikipedia "Calkin-Wilf tree" http://goo.gl/cB3vn  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
import Data.List  
import Test.QuickCheck  
  
-- -----
```

```
-- Numeración de los racionales mediante representaciones hiperbinarias
-- -----
--
-- Ejercicio 1. Definir la constante
--   potenciasDeDos :: [Integer]
-- tal que potenciasDeDos es la lista de las potencias de 2. Por
-- ejemplo,
--   take 10 potenciasDeDos == [1,2,4,8,16,32,64,128,256,512]
-- -----
```

```
potenciasDeDos :: [Integer]
potenciasDeDos = [2^n | n <- [0..]]
```

```
-- -----
-- Ejercicio 2. Definir la función
--   empiezaConDos :: Eq a => a -> [a] -> Bool
-- tal que (empiezaConDos x ys) se verifica si los dos primeros
-- elementos de ys son iguales a x. Por ejemplo,
--   empiezaConDos 5 [5,5,3,7] == True
--   empiezaConDos 5 [5,3,5,7] == False
--   empiezaConDos 5 [5,5,5,7] == True
-- -----
```

```
empiezaConDos :: Eq a => a -> [a] -> Bool
empiezaConDos x (y1:y2:ys) = y1 == x && y2 == x
empiezaConDos x _         = False
```

```
-- -----
-- Ejercicio 3. Definir la función
--   representacionesHB :: Integer -> [[Integer]]
-- tal que (representacionesHB n) es la lista de las representaciones
-- hiperbinarias del número n como suma de potencias de 2 donde cada
-- sumando aparece como máximo 2 veces. Por ejemplo
--   representacionesHB 5 == [[1,2,2],[1,4]]
--   representacionesHB 6 == [[1,1,2,2],[1,1,4],[2,4]]
-- -----
```

```
representacionesHB :: Integer -> [[Integer]]
representacionesHB n = representacionesHB' n potenciasDeDos
```

```
representacionesHB' n (x:xs)
  | n == 0    = [[]]
  | x == n    = [[x]]
  | x < n     = [x:ys | ys <- representacionesHB' (n-x) (x:xs),
                  not (empiezaConDos x ys)] ++
                representacionesHB' n xs
  | otherwise = []
```

```
-----
-- Ejercicio 4. Definir la función
--   nRepresentacionesHB :: Integer -> Integer
-- tal que (nRepresentacionesHB n) es el número de las representaciones
-- hiperbinarias del número n como suma de potencias de 2 donde cada
-- sumando aparece como máximo 2 veces. Por ejemplo,
--   ghci> [nRepresentacionesHB n | n <- [0..20]]
--   [1,1,2,1,3,2,3,1,4,3,5,2,5,3,4,1,5,4,7,3,8]
-----
```

```
nRepresentacionesHB :: Integer -> Integer
nRepresentacionesHB = genericLength . representacionesHB
```

```
-----
-- Ejercicio 5. Definir la función
--   termino :: Integer -> (Integer,Integer)
-- tal que (termino n) es el par formado por el número de
-- representaciones hiperbinarias de n y de n+1 (que se interpreta como
-- su cociente). Por ejemplo,
--   termino 4 == (3,2)
-----
```

```
termino :: Integer -> (Integer,Integer)
termino n = (nRepresentacionesHB n, nRepresentacionesHB (n+1))
```

```
-----
-- Ejercicio 6. Definir la función
--   sucesionHB :: [(Integer,Integer)]
-- sucesionHB es la sucesión cuyo término n-ésimo es (termino n); es
-- decir, el par formado por el número de representaciones hiperbinarias
-- de n y de n+1. Por ejemplo,
--   ghci> take 10 sucesionHB
```

```
-- [(1,1), (1,2), (2,1), (1,3), (3,2), (2,3), (3,1), (1,4), (4,3), (3,5)]
```

```
-----
sucesionHB :: [(Integer,Integer)]
sucesionHB = [termino n | n <- [0..]]
```

```
-----
-- Ejercicio 7. Comprobar con QuickCheck que, para todo n,
-- (nRepresentacionesHB n) y (nRepresentacionesHB (n+1)) son primos
-- entre sí.
```

```
-----
prop_irreducibles :: Integer -> Property
prop_irreducibles n =
  n >= 0 ==>
  gcd (nRepresentacionesHB n) (nRepresentacionesHB (n+1)) == 1
```

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

```
-----
-- Ejercicio 8. Comprobar con QuickCheck que todos los elementos de la
-- sucesionHB son distintos.
```

```
-----
prop_distintos :: Integer -> Integer -> Bool
prop_distintos n m =
  termino n' /= termino m'
  where n' = abs n
        m' = n' + abs m + 1
```

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

```
-----
-- Ejercicio 9. Definir la función
-- contenido :: Integer -> Integer -> Bool
-- tal que (contenido n) se verifica si la expresiones reducidas de
```

```
-- todas las fracciones x/y, con x e y entre 1 y n, pertenecen a la
-- sucesionHB. Por ejemplo,
--   contenidos 5 == True
```

```
-----
```

```
contenido :: Integer -> Bool
```

```
contenido n =
```

```
  and [pertenece (reducida (x,y)) sucesionHB |
        x <- [1..n], y <- [1..n]]
  where pertenece x (y:ys) = x == y || pertenece x ys
        reducida (x,y) = (x 'div' z, y 'div' z)
                where z = gcd x y
```

```
-----
```

```
-- Ejercicio 10. Definir la función
```

```
--   indice :: (Integer,Integer) -> Integer
-- tal que (indice (a,b)) es el índice del par (a,b) en la sucesión de
-- los racionales. Por ejemplo,
--   indice (3,2) == 4
```

```
-----
```

```
indice :: (Integer,Integer) -> Integer
```

```
indice (a,b) = head [n | (n,(x,y)) <- zip [0..] sucesionHB,
                       (x,y) == (a,b)]
```

```
-----
```

```
-- Numeraciones mediante árboles de Calkin-Wilf
```

```
-----
```

```
-- El árbol de Calkin-Wilf es el árbol definido por las siguientes
-- reglas:
```

```
--   * El nodo raíz es el (1,1)
```

```
--   * Los hijos del nodo (x,y) son (x,x+y) y (x+y,y)
```

```
-- Por ejemplo, los 4 primeros niveles del árbol de Calkin-Wilf son
```

```
--           (1,1)
--           |
--       +-----+-----+
--       |           |
--       | (1,2)     | (2,1)
--       |           |
```

```

--           +-----+-----+           +-----+-----+
--           |           |           |           |           |
--           (1,3)      (3,2)      (2,3)      (3,1)
--           |           |           |           |           |
--           +---+---+   +---+---+   +---+---+   +---+---+
--           |     |     |     |     |     |     |     |     |
--           (1,4) (4,3) (3,5) (5,2) (2,5) (5,3) (3,4) (4,1)

```

```

-----
-- Ejercicio 11. Definir la función
--   sucesores :: (Integer,Integer) -> [(Integer,Integer)]
--   tal que (sucesores (x,y)) es la lista de los hijos del par (x,y) en
--   el árbol de Calkin-Wilf. Por ejemplo,
--   sucesores (3,2) == [(3,5),(5,2)]
-----

```

```

sucesores :: (Integer,Integer) -> [(Integer,Integer)]
sucesores (x,y) = [(x,x+y),(x+y,y)]

```

```

-----
-- Ejercicio 12. Definir la función
--   siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
--   tal que (siguiente xs) es la lista formada por los hijos de los
--   elementos de xs en el árbol de Calkin-Wilf. Por ejemplo,
--   ghci> siguiente [(1,3),(3,2),(2,3),(3,1)]
--   [(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]
-----

```

```

siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
siguiente xs = [p | x <- xs, p <- sucesores x]

```

```

-----
-- Ejercicio 13. Definir la constante
--   nivelesCalkinWilf :: [(Integer,Integer)]
--   tal que nivelesCalkinWilf es la lista de los niveles del árbol de
--   Calkin-Wilf. Por ejemplo,
--   ghci> take 4 nivelesCalkinWilf
--   [(1,1)],
--   [(1,2),(2,1)],
--   [(1,3),(3,2),(2,3),(3,1)],

```

```
-- [(1,4), (4,3), (3,5), (5,2), (2,5), (5,3), (3,4), (4,1)]
```

```
nivelesCalkinWilf :: [(Integer, Integer)]
nivelesCalkinWilf = iterate siguiente [(1,1)]
```

```
-- -----
-- Ejercicio 14. Definir la constante
--   sucesionCalkinWilf :: [(Integer, Integer)]
-- tal que sucesionCalkinWilf es la lista correspondiente al recorrido
-- en anchura del árbol de Calkin-Wilf. Por ejemplo,
--   ghci> take 10 sucesionCalkinWilf
--   [(1,1), (1,2), (2,1), (1,3), (3,2), (2,3), (3,1), (1,4), (4,3), (3,5)]
```

```
sucesionCalkinWilf :: [(Integer, Integer)]
sucesionCalkinWilf = concat nivelesCalkinWilf
```

```
-- -----
-- Ejercicio 15. Definir la función
--   igual_sucesion_HB_CalkinWilf :: Int -> Bool
-- tal que (igual_sucesion_HB_CalkinWilf n) se verifica si los n
-- primeros términos de la sucesión HB son iguales que los de la
-- sucesión de Calkin-Wilf. Por ejemplo,
--   igual_sucesion_HB_CalkinWilf 20 == True
```

```
igual_sucesion_HB_CalkinWilf :: Int -> Bool
igual_sucesion_HB_CalkinWilf n =
  take n sucesionCalkinWilf == take n sucesionHB
```

```
-- -----
-- Número de representaciones hiperbinarias mediante la función fusc
```

```
-- -----
-- Ejercicio 16. Definir la función
--   fusc :: Integer -> Integer
-- tal que
--   fusc(0) = 1
```

```
-- fusc(2n+1) = fusc(n)
-- fusc(2n+2) = fusc(n+1)+fusc(n)
-- Por ejemplo,
-- fusc 4 == 3
```

```
-----
fusc :: Integer -> Integer
fusc 0 = 1
fusc n | odd n      = fusc ((n-1) 'div' 2)
      | otherwise = fusc(m+1) + fusc m
      where m = (n-2) 'div' 2
```

```
-----
-- Ejercicio 17. Comprobar con QuickCheck que, para todo n, (fusc n) es
-- el número de las representaciones hiperbinarias del número n como
-- suma de potencias de 2 donde cada sumando aparece como máximo 2
-- veces; es decir, que las funciones fusc y nRepresentacionesHB son
-- equivalentes.
```

```
-----
prop_fusc :: Integer -> Bool
prop_fusc n = nRepresentacionesHB n' == fusc n'
      where n' = abs n
```

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


Relación 26

Operaciones con el TAD de montículos

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir funciones sobre  
-- el TAD de las montículos, utilizando las implementaciones estudiadas  
-- en el tema 20 que se pueden descargar desde  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/codigos/I1M2014.zip  
--  
-- Las transparencias del tema 20 se encuentran en  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-20.pdf  
  
-----  
-- Importación de librerías --  
-----  
  
{-# LANGUAGE FlexibleInstances #-}  
  
import I1M.Monticulo      -- Se instala con http://bit.ly/1AKmUQB  
import Test.QuickCheck  
  
-----  
-- Ejemplos --  
-----
```

```

-- Para los ejemplos se usarán los siguientes montículos.
m1, m2, m3 :: Monticulo Int
m1 = foldr inserta vacio [6,1,4,8]
m2 = foldr inserta vacio [7,5]
m3 = foldr inserta vacio [6,1,4,8,7,5]

-----
-- Ejercicio 1. Definir la función
--   numeroDeNodos :: Ord a => Monticulo a -> Int
-- tal que (numeroDeNodos m) es el número de nodos del montículo m. Por
-- ejemplo,
--   numeroDeNodos m1 == 4
-----

numeroDeNodos :: Ord a => Monticulo a -> Int
numeroDeNodos m
  | esVacio m = 0
  | otherwise = 1 + numeroDeNodos (resto m)

-----
-- Ejercicio 2. Definir la función
--   filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
-- tal que (filtra p m) es el montículo con los nodos del montículo m
-- que cumplen la propiedad p. Por ejemplo,
--   ghci> m1
--   M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--   ghci> filtra even m1
--   M 4 1 (M 6 1 (M 8 1 Vacio Vacio) Vacio) Vacio
--   ghci> filtra odd m1
--   M 1 1 Vacio Vacio
-----

filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
filtra p m
  | esVacio m = vacio
  | p mm      = inserta mm (filtra p rm)
  | otherwise = filtra p rm
  where mm = menor m
        rm = resto m

```

```

-----
-- Ejercicio 3. Definir la función
--   menores :: Ord a => Int -> Monticulo a -> [a]
-- tal que (menores n m) es la lista de los n menores elementos del
-- montículo m. Por ejemplo,
--   ghci> m1
--   M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--   ghci> menores 3 m1
--   [1,4,6]
-----

```

```

menores :: Ord a => Int -> Monticulo a -> [a]
menores 0 m = []
menores n m | esVacio m = []
             | otherwise = menor m : menores (n-1) (resto m)

```

```

-----
-- Ejercicio 4. Definir la función
--   restantes :: Ord a => Int -> Monticulo a -> Monticulo a
-- tal que (restantes n m) es el montículo obtenido rliminando los n
-- menores elementos del montículo m. Por ejemplo,
--   ghci> m1
--   M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--   ghci> restantes 3 m1
--   M 8 1 Vacio Vacio
--   ghci> restantes 2 m1
--   M 6 1 (M 8 1 Vacio Vacio) Vacio
-----

```

```

restantes :: Ord a => Int -> Monticulo a -> Monticulo a
restantes 0 m = m
restantes n m | esVacio m = vacio
             | otherwise = restantes (n-1) (resto m)

```

```

-----
-- Ejercicio 5. Definir la función
--   lista2Monticulo :: Ord a => [a] -> Monticulo a
-- tal que (lista2Monticulo xs) es el montículo cuyos nodos son los
-- elementos de la lista xs. Por ejemplo,
--   ghci> lista2Monticulo [2,5,3,7]
-----

```

```
-- M 2 1 (M 3 2 (M 7 1 Vacio Vacio) (M 5 1 Vacio Vacio)) Vacio
```

```
-----
lista2Monticulo :: Ord a => [a] -> Monticulo a
lista2Monticulo = foldr inserta vacio
```

```
-----
-- Ejercicio 6. Definir la función
```

```
-- monticulo2Lista :: Ord a => Monticulo a -> [a]
-- tal que (monticulo2Lista m) es la lista ordenada de los nodos del
-- montículo m. Por ejemplo,
-- ghci> m1
-- M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
-- ghci> monticulo2Lista m1
-- [1,4,6,8]
```

```
-----
monticulo2Lista :: Ord a => Monticulo a -> [a]
monticulo2Lista m
  | esVacio m = []
  | otherwise = menor m : monticulo2Lista (resto m)
```

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

```
-- ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada de
-- forma creciente. Por ejemplo,
-- ordenada [3,5,9] == True
-- ordenada [3,5,4] == False
-- ordenada [7,5,4] == False
```

```
-----
ordenada :: Ord a => [a] -> Bool
ordenada (x:y:zs) = x <= y && ordenada (y:zs)
ordenada _      = True
```

```
-----
-- Ejercicio 8. Comprobar con QuickCheck que para todo montículo m,
-- (monticulo2Lista m) es una lista ordenada creciente.
```

```
-- La propiedad es
prop_monticulo2Lista_ordenada :: Monticulo Int -> Bool
prop_monticulo2Lista_ordenada m =
    ordenada (monticulo2Lista m)

-- La comprobación es
-- ghci> quickCheck prop_monticulo2Lista_ordenada
-- +++ OK, passed 100 tests.

-----
-- Ejercicio 10. Usando monticulo2Lista y lista2Monticulo, definir la
-- función
-- ordena :: Ord a => [a] -> [a]
-- tal que (ordena xs) es la lista obtenida ordenando de forma creciente
-- los elementos de xs. Por ejemplo,
-- ordena [7,5,3,6,5] == [3,5,5,6,7]
-----

ordena :: Ord a => [a] -> [a]
ordena = monticulo2Lista . lista2Monticulo

-----
-- Ejercicio 11. Comprobar con QuickCheck que para toda lista xs,
-- (ordena xs) es una lista ordenada creciente.
-----

-- La propiedad es
prop_ordena_ordenada :: [Int] -> Bool
prop_ordena_ordenada xs =
    ordenada (ordena xs)

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

-----
-- Ejercicio 12. Definir la función
-- borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
```

```

-- x en la lista xs. Por ejemplo,
--   borra 1 [1,2,1] == [2,1]
--   borra 3 [1,2,1] == [1,2,1]
-----

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

-- Ejercicio 14. Definir la función esPermutación tal que
-- (esPermutación xs ys) se verifique si xs es una permutación de
-- ys. Por ejemplo,
--   esPermutación [1,2,1] [2,1,1] == True
--   esPermutación [1,2,1] [1,2,2] == False
-----

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

-- Ejercicio 15. Comprobar con QuickCheck que para toda lista xs,
-- (ordena xs) es una permutación de xs.
-----

-- La propiedad es
prop_ordena_permutacion :: [Int] -> Bool
prop_ordena_permutacion xs =
  esPermutacion (ordena xs) xs

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

-- Generador de montículos
-----

```

```
-- genMonticulo es un generador de montículos. Por ejemplo,
-- ghci> sample genMonticulo
-- VacioM
-- M (-1) 1 (M 1 1 VacioM VacioM) VacioM
-- ...
genMonticulo :: Gen (Monticulo Int)
genMonticulo = do xs <- listOf arbitrary
                return (foldr inserta vacio xs)

-- Monticulo es una instancia de la clase arbitraria.
instance Arbitrary (Monticulo Int) where
    arbitrary = genMonticulo
```


Relación 27

Operaciones con conjuntos usando la librería Data.Set

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es hacer los ejercicios de la relación 19  
-- sobre operaciones con conjuntos usando la librería Data.Set  
  
-- -----  
-- § Librerías auxiliares --  
-- -----  
  
import Data.Set as S  
  
-- -----  
-- § Ejercicios --  
-- -----  
  
-- Ejercicio 1. Definir la función  
-- subconjunto :: Ord a => Set a -> Set a -> Bool  
-- tal que (subconjunto c1 c2) se verifica si todos los elementos de c1  
-- pertenecen a c2. Por ejemplo,  
-- subconjunto (fromList [2..100000]) (fromList [1..100000]) == True  
-- subconjunto (fromList [1..100000]) (fromList [2..100000]) == False  
-- -----
```

```

subconjunto :: Ord a => Set a -> Set a -> Bool
subconjunto = isSubsetOf

```

```

-----
-- Ejercicio 2. Definir la función
--   subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
-- tal (subconjuntoPropio c1 c2) se verifica si c1 es un subconjunto
-- propio de c2. Por ejemplo,
--   subconjuntoPropio (fromList [2..5]) (fromList [1..7]) == True
--   subconjuntoPropio (fromList [2..5]) (fromList [1..4]) == False
--   subconjuntoPropio (fromList [2..5]) (fromList [2..5]) == False
-----

```

```

subconjuntoPropio :: Ord a => Set a -> Set a -> Bool
subconjuntoPropio = isProperSubsetOf

```

```

-----
-- Ejercicio 3. Definir la función
--   unitario :: Ord a => a -> Set a
-- tal que (unitario x) es el conjunto {x}. Por ejemplo,
--   unitario 5 == fromList [5]
-----

```

```

unitario :: Ord a => a -> Set a
unitario = singleton

```

```

-----
-- Ejercicio 4. Definir la función
--   cardinal :: Set a -> Int
-- tal que (cardinal c) es el número de elementos del conjunto c. Por
-- ejemplo,
--   cardinal (fromList [3,2,5,1,2,3]) == 4
-----

```

```

cardinal :: Set a -> Int
cardinal = size

```

```

-----
-- Ejercicio 5. Definir la función

```

```
-- unión :: Ord a => Set a -> Set a -> Set a
-- tal (unión c1 c2) es la unión de ambos conjuntos. Por ejemplo,
-- ghci> unión (fromList [3,2,5]) (fromList [2,7,5])
-- fromList [2,3,5,7]
```

```
-----
unión :: Ord a => Set a -> Set a -> Set a
unión = union
```

```
-----
-- Ejercicio 6. Definir la función
-- unionG :: Ord a => [Set a] -> Set a
-- tal (unionG cs) calcule la unión de la lista de conjuntos cd. Por
-- ejemplo,
-- ghci> unionG [fromList [3,2], fromList [2,5], fromList [3,5,7]]
-- fromList [2,3,5,7]
```

```
-----
unionG :: Ord a => [Set a] -> Set a
unionG = unions
```

```
-----
-- Ejercicio 7. Definir la función
-- interseccion :: Ord a => Set a -> Set a -> Set a
-- tal que (interseccion c1 c2) es la intersección de los conjuntos c1 y
-- c2. Por ejemplo,
-- ghci> interseccion (fromList [1..7]) (fromList [4..9])
-- fromList [4,5,6,7]
-- ghci> interseccion (fromList [2..1000000]) (fromList [1])
-- fromList []
```

```
-----
interseccion :: Ord a => Set a -> Set a -> Set a
interseccion = intersection
```

```
-----
-- Ejercicio 8. Definir la función
-- interseccionG :: Ord a => [Set a] -> Set a
-- tal que (interseccionG cs) es la intersección de la lista de
-- conjuntos cs. Por ejemplo,
```

```
-- ghci> interseccionG [fromList [3,2], fromList [2,5,3], fromList [3,5,7]]
-- fromList [3]
```

```
-----
interseccionG :: Ord a => [Set a] -> Set a
interseccionG [c] = c
interseccionG (cs:css) = intersection cs (interseccionG css)
```

```
-- Se puede definir por plegado
interseccionG2 :: Ord a => [Set a] -> Set a
interseccionG2 = foldr1 interseccion
```

```
-----
-- Ejercicio 9. Definir la función
-- disjuntos :: Ord a => Set a -> Set a -> Bool
-- tal que (disjuntos c1 c2) se verifica si los conjuntos c1 y c2 son
-- disjuntos. Por ejemplo,
-- disjuntos (fromList [2..5]) (fromList [6..9]) == True
-- disjuntos (fromList [2..5]) (fromList [1..9]) == False
```

```
-----
disjuntos :: Ord a => Set a -> Set a -> Bool
disjuntos c1 c2 = S.null (intersection c1 c2)
```

```
-----
-- Ejercicio 10. Definir la función
-- diferencia :: Ord a => Set a -> Set a -> Set a
-- tal que (diferencia c1 c2) es el conjunto de los elementos de c1 que
-- no son elementos de c2. Por ejemplo,
-- ghci> diferencia (fromList [2,5,3]) (fromList [1,4,5])
-- fromList [2,3]
```

```
-----
diferencia :: Ord a => Set a -> Set a -> Set a
diferencia = difference
```

```
-----
-- Ejercicio 11. Definir la función
-- diferenciaSimetrica :: Ord a => Set a -> Set a -> Set a
-- tal que (diferenciaSimetrica c1 c2) es la diferencia simétrica de los
```

```
-- conjuntos c1 y c2. Por ejemplo,  
-- ghci> diferenciaSimetrica (fromList [3,2,5]) (fromList [1,5])  
-- fromList [1,2,3]  
-----
```

```
diferenciaSimetrica :: Ord a => Set a -> Set a -> Set a  
diferenciaSimetrica c1 c2 =  
  (union c1 c2) \\  
  (intersection c1 c2)
```

```
-- Ejercicio 12. Definir la función  
-- filtra :: (a -> Bool) -> Set a -> Set a  
-- tal (filtra p c) es el conjunto de elementos de c que verifican el  
-- predicado p. Por ejemplo,  
-- filtra even (fromList [3,2,5,6,8,9]) == fromList [2,6,8]  
-- filtra odd (fromList [3,2,5,6,8,9]) == fromList [3,5,9]  
-----
```

```
filtra :: (a -> Bool) -> Set a -> Set a  
filtra = S.filter
```

```
-- Ejercicio 13. Definir la función  
-- particion :: (a -> Bool) -> Set a -> (Set a, Set a)  
-- tal que (particion c) es el par formado por dos conjuntos: el de sus  
-- elementos que verifican p y el de los elementos que no lo verifica.  
-- Por ejemplo,  
-- ghci> particion even (fromList [3,2,5,6,8,9])  
-- (fromList [2,6,8], fromList [3,5,9])  
-----
```

```
particion :: (a -> Bool) -> Set a -> (Set a, Set a)  
particion = partition
```

```
-- Ejercicio 14. Definir la función  
-- divide :: (Ord a) => a -> Set a -> (Set a, Set a)  
-- tal que (divide x c) es el par formado por dos subconjuntos de c: el  
-- de los elementos menores que x y el de los mayores que x. Por ejemplo,  
-- ghci> divide 5 (fromList [3,2,9,5,8,6])  
-----
```

```
-- (fromList [2,3],fromList [6,8,9])
```

```
-----
divide :: Ord a => a-> Set a -> (Set a, Set a)
divide = split
```

```
-----
-- Ejercicio 15. Definir la función
```

```
-- mapC :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
-- tal que (map f c) es el conjunto formado por las imágenes de los
-- elementos de c, mediante f. Por ejemplo,
-- mapC (*2) (fromList [1..4]) == fromList [2,4,6,8]
```

```
-----
mapC :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
mapC = S.map
```

```
-----
-- Ejercicio 16. Definir la función
```

```
-- everyC :: Ord a => (a -> Bool) -> Set a -> Bool
-- tal que (everyC p c) se verifica si todos los elementos de c
-- verifican el predicado p. Por ejemplo,
-- everyC even (fromList [2,4..10]) == True
-- everyC even (fromList [2..10]) == False
```

```
-----
everyC :: Ord a => (a -> Bool) -> Set a -> Bool
everyC p c | S.null c = True
           | otherwise = p x && everyC p c1
           where (x,c1) = deleteFindMin c
```

```
-----
-- Ejercicio 17. Definir la función
```

```
-- someC :: Ord a => (a -> Bool) -> Set a -> Bool
-- tal que (someC p c) se verifica si algún elemento de c verifica el
-- predicado p. Por ejemplo,
-- someC even (fromList [1,4,7]) == True
-- someC even (fromList [1,3,7]) == False
```

```
someC :: Ord a => (a -> Bool) -> Set a -> Bool
someC p c | S.null c = False
          | otherwise = p x || someC p c1
          where (x,c1) = deleteFindMin c
```

```
-----
-- Ejercicio 18. Definir la función
--   productoC :: (Ord a, Ord b) => Set a -> Set b -> Set (a,b)
-- tal que (productoC c1 c2) es el producto cartesiano de los
-- conjuntos c1 y c2. Por ejemplo,
--   ghci> productoC (fromList [1,3]) (fromList [2,4])
--   fromList [(1,2),(1,4),(3,2),(3,4)]
-----
```

```
productoC :: (Ord a, Ord b) => Set a -> Set b -> Set (a,b)
productoC c1 c2 =
  fromList [(x,y) | x <- elems c1, y <- elems c2]
```

```
-----
-- Ejercicio 19. Definir la función
--   potencia :: Ord a => Set a -> Set (Set a)
-- tal que (potencia c) es el conjunto potencia de c; es decir, el
-- conjunto de todos los subconjuntos de c. Por ejemplo,
--   ghci> potencia (fromList [1..3])
--   fromList [fromList [],fromList [1],fromList [1,2],fromList [1,2,3],
--             fromList [1,3],fromList [2],fromList [2,3],fromList [3]]
-----
```



```
potencia :: Ord a => Set a -> Set (Set a)
potencia c | S.null c = singleton empty
           | otherwise = S.map (insert x) pr 'union' pr
  where (x,rc) = deleteFindMin c
        pr     = potencia rc
```


Relación 28

Relaciones binarias homogéneas con la librería Data.Set

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación de ejercicios es definir propiedades y  
-- operaciones sobre las relaciones binarias (homogéneas) usando la  
-- librería Data.Set.  
--  
-- Como referencia se puede usar el artículo de la wikipedia  
-- http://bit.ly/HVHOPS  
  
-- -----  
-- § Pragmas --  
-- -----  
  
{-# LANGUAGE OverlappingInstances,  
      TypeSynonymInstances,  
      FlexibleInstances #-}  
  
-- -----  
-- § Librerías auxiliares --  
-- -----  
  
import Test.QuickCheck  
import Data.Set
```

```

-----
-- Ejercicio 1. Una relación binaria R sobre un conjunto A puede
-- representar mediante un par (xs,ps) donde xs es el conjunto de los
-- elementos de A (el universo de R) y ps es el conjunto de pares de R
-- (el grafo de R). Definir el tipo de dato (Rel a) para representar las
-- relaciones binarias sobre a.
-----

```

```

type Rel a = (Set a, Set (a,a))

```

```

-----
-- Nota. En los ejemplos usaremos las siguientes relaciones binarias:
--   r1, r2, r3 :: Rel Int
--   r1 = (fromList [1..9],fromList [(1,3), (2,6), (8,9), (2,7)])
--   r2 = (fromList [1..9],fromList [(1,3), (2,6), (8,9), (3,7)])
--   r3 = (fromList [1..9],fromList [(1,3), (2,6), (8,9), (3,6)])
-----

```

```

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

```

```

-----
-- Ejercicio 2. Definir la función
--   universo :: Ord a => Rel a -> Set a
-- tal que (universo r) es el universo de la relación r. Por ejemplo,
--   universo r1 == fromList [1,2,3,4,5,6,7,8,9]
-----

```

```

universo :: Ord a => Rel a -> Set a
universo (u,_) = u

```

```

-----
-- Ejercicio 3. Definir la función
--   grafo :: Ord a => Rel a -> [(a,a)]
-- tal que (grafo r) es el grafo de la relación r. Por ejemplo,
--   grafo r1 == fromList [(1,3),(2,6),(2,7),(8,9)]
-----

```

```

grafo :: Ord a => Rel a -> Set (a,a)
grafo (_,g) = g

```

```

-----
-- Ejercicio 4. Definir la función
--   reflexiva :: Ord a => Rel a -> Bool
--   tal que (reflexiva r) se verifica si la relación r es reflexiva. Por
--   ejemplo,
--   ghci> reflexiva (fromList [1,3], fromList [(1,1),(1,3),(3,3)])
--   True
--   ghci> reflexiva (fromList [1,2,3], fromList [(1,1),(1,3),(3,3)])
--   False
-----

```

```

reflexiva :: Ord a => Rel a -> Bool
reflexiva (u,g) = and [(x,x) 'member' g | x <- elems u]

```

```

-----
-- Ejercicio 5. Definir la función
--   simetrica :: Ord a => Rel a -> Bool
--   tal que (simetrica r) se verifica si la relación r es simétrica. Por
--   ejemplo,
--   ghci> simetrica (fromList [1,3], fromList [(1,1),(1,3),(3,1)])
--   True
--   ghci> simetrica (fromList [1,3], fromList [(1,1),(1,3),(3,2)])
--   False
--   ghci> simetrica (fromList [1,3], fromList [])
--   True
-----

```

```

simetrica :: Ord a => Rel a -> Bool
simetrica (u,g) = and [(y,x) 'member' g | (x,y) <- elems g]

```

```

-----
-- Ejercicio 6. Definir la función
--   subconjunto :: Ord a => Set a -> Set a -> Bool
--   tal que (subconjunto c1 c2) se verifica si c1 es un subconjunto de
--   c2. Por ejemplo,
--   subconjunto (fromList [1,3]) (fromList [3,1,5]) == True
-----

```

```
-- subconjunto (fromList [3,1,5]) (fromList [1,3]) == False
```

```
-----
subconjunto :: Ord a => Set a -> Set a -> Bool
subconjunto = isSubsetOf
```

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

```
-- composicion :: Ord a => Rel a -> Rel a -> Rel a
-- tal que (composicion r s) es la composición de las relaciones r y
-- s. Por ejemplo,
-- ghci> let r1 = (fromList [1,2], fromList [(1,2),(2,2)])
-- ghci> let r2 = (fromList [1,2], fromList [(2,1)])
-- ghci> composicion r1 r2
-- (fromList [1,2],fromList [(1,1),(2,1)])
```

```
-----
composicion :: Ord a => Rel a -> Rel a -> Rel a
```

```
composicion (u,g1) (_,g2) =
  (u,fromList [(x,z) | (x,y1) <- elems g1,
                    (y2,z) <- elems g2,
                    y1 == y2])
```

```
-----
-- Ejercicio 8. Definir la función
```

```
-- transitiva :: Ord a => Rel a -> Bool
-- tal que (transitiva r) se verifica si la relación r es transitiva.
-- Por ejemplo,
-- ghci> transitiva (fromList [1,3,5],fromList [(1,1),(1,3),(3,1),(3,3),(5,5)])
-- True
-- ghci> transitiva (fromList [1,3,5],fromList [(1,1),(1,3),(3,1),(5,5)])
-- False
```

```
-----
transitiva :: Ord a => Rel a -> Bool
```

```
transitiva r@(u,g) =
  isSubsetOf (grafo (composicion r r)) g
```

```
-----
-- Ejercicio 9. Definir la función
```

```

--     esEquivalencia :: Ord a => Rel a -> Bool
-- tal que (esEquivalencia r) se verifica si la relación r es de
-- equivalencia. Por ejemplo,
--     ghci> esEquivalencia (fromList [1,3,5],
--                             fromList [(1,1),(1,3),(3,1),(3,3),(5,5)])
--     True
--     ghci> esEquivalencia (fromList [1,2,3,5],
--                             fromList [(1,1),(1,3),(3,1),(3,3),(5,5)])
--     False
--     ghci> esEquivalencia (fromList [1,3,5],
--                             fromList [(1,1),(1,3),(3,3),(5,5)])
--     False

```

```

esEquivalencia :: Ord a => Rel a -> Bool

```

```

esEquivalencia r = reflexiva r && simetrica r && transitiva r

```

```

-----
-- Ejercicio 10. Definir la función
--     irreflexiva :: Ord a => Rel a -> Bool
-- tal que (irreflexiva r) se verifica si la relación r es irreflexiva;
-- es decir, si ningún elemento de su universo está relacionado con
-- él mismo. Por ejemplo,
--     ghci> irreflexiva (fromList [1,2,3],fromList [(1,2),(2,1),(2,3)])
--     True
--     ghci> irreflexiva (fromList [1,2,3],fromList [(1,2),(2,1),(3,3)])
--     False

```

```

irreflexiva :: Ord a => Rel a -> Bool

```

```

irreflexiva (u,g) = and [(x,x) 'notMember' g | x <- elems u]

```

```

-----
-- Ejercicio 11. Definir la función
--     antisimetrica :: Ord a => Rel a -> Bool
-- tal que (antisimetrica r) se verifica si la relación r es
-- antisimétrica; es decir, si (x,y) e (y,x) están relacionado, entonces
-- x=y. Por ejemplo,
--     antisimetrica (fromList [1,2],fromList [(1,2)])           == True
--     antisimetrica (fromList [1,2],fromList [(1,2),(2,1)])    == False

```

```

--      antisimetrica (fromList [1,2],fromList [(1,1),(2,1)]) == True
-----

antisimetrica :: Ord a => Rel a -> Bool
antisimetrica (_,g) =
  [(x,y) | (x,y) <- elems g, x /= y, (y,x) 'member' g] == []

-- Otra definición es
antisimetrica2 :: Ord a => Rel a -> Bool
antisimetrica2 (u,g) =
  and [((x,y) 'member' g && (y,x) 'member' g) --> (x == y)
       | x <- elems u, y <- elems u]
  where p --> q = not p || q

-----

-- Ejercicio 12. Definir la función
--      total :: Ord a => Rel a -> Bool
-- tal que (total r) se verifica si la relación r es total; es decir, si
-- para cualquier par x, y de elementos del universo de r, se tiene que
-- x está relacionado con y ó y está relacionado con x. Por ejemplo,
--      total (fromList [1,3],fromList [(1,1),(3,1),(3,3)]) == True
--      total (fromList [1,3],fromList [(1,1),(3,1)])          == False
--      total (fromList [1,3],fromList [(1,1),(3,3)])          == False
-----

total :: Ord a => Rel a -> Bool
total (u,g) =
  and [(x,y) 'member' g || (y,x) 'member' g | x <- xs, y <- xs]
  where xs = elems u

-----

-- Ejercicio 13. Comprobar con QuickCheck que las relaciones totales son
-- reflexivas.
-----

prop_total_reflexiva :: Rel Int -> Property
prop_total_reflexiva r =
  total r ==> reflexiva r

-- La comprobación es

```



```

--      ghci> quickCheck prop_total_reflexiva
--      *** ** Gave up! Passed only 77 tests.

-----

-- § Clausuras
-----

-----

-- Ejercicio 14. Definir la función
--      clausuraReflexiva :: Ord a => Rel a -> Rel a
--      tal que (clausuraReflexiva r) es la clausura reflexiva de r; es
--      decir, la menor relación reflexiva que contiene a r. Por ejemplo,
--      ghci> clausuraReflexiva (fromList [1,3], fromList [(1,1),(3,1)])
--      (fromList [1,3],fromList [(1,1),(3,1),(3,3)])
-----

clausuraReflexiva :: Ord a => Rel a -> Rel a
clausuraReflexiva (u,g) =
    (u, g 'union' fromList [(x,x) | x <- elems u])

-----

-- Ejercicio 15. Comprobar con QuickCheck que clausuraReflexiva es
--      reflexiva.
-----

prop_ClausuraReflexiva :: Rel Int -> Bool
prop_ClausuraReflexiva r =
    reflexiva (clausuraReflexiva r)

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

-----

-- Ejercicio 16. Definir la función
--      clausuraSimetrica :: Ord a => Rel a -> Rel a
--      tal que (clausuraSimetrica r) es la clausura simétrica de r; es
--      decir, la menor relación simétrica que contiene a r. Por ejemplo,
--      ghci> clausuraSimetrica (fromList [1,3,5],fromList [(1,1),(3,1),(1,5)])
--      (fromList [1,3,5],fromList [(1,1),(1,3),(1,5),(3,1),(5,1)])

```

```
-----
clausuraSimetrica :: Ord a => Rel a -> Rel a
clausuraSimetrica (u,g) =
  (u, g 'union' fromList [(y,x) | (x,y) <- elems g])
```

```
-----
-- Ejercicio 17. Comprobar con QuickCheck que clausuraSimetrica es
-- simétrica.
-----
```

```
prop_ClausuraSimetrica :: Rel Int -> Bool
prop_ClausuraSimetrica r =
  simetrica (clausuraSimetrica r)
```

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

```
-----
-- Ejercicio 18. Definir la función
-- clausuraTransitiva :: Ord a => Rel a -> Rel a
-- tal que (clausuraTransitiva r) es la clausura transitiva de r; es
-- decir, la menor relación transitiva que contiene a r. Por ejemplo,
-- ghci> clausuraTransitiva (fromList [1..6],fromList [(1,2),(2,5),(5,6)])
-- (fromList [1,2,3,4,5,6],fromList [(1,2),(1,5),(1,6),(2,5),(2,6),(5,6)])
-----
```

```
clausuraTransitiva :: Ord a => Rel a -> Rel a
clausuraTransitiva (u,g) = (u, aux g)
  where aux r | cerradoTr r = r
            | otherwise    = aux (r 'union' (comp r r))
        cerradoTr r = isSubsetOf (comp r r) r
        comp r s    = fromList [(x,z) | (x,y1) <- elems r,
                                       (y2,z) <- elems s,
                                       y1 == y2]
```

```
-----
-- Ejercicio 19. Comprobar con QuickCheck que clausuraTransitiva es
-- transitiva.
-----
```

```

-----
prop_ClausuraTransitiva :: Rel Int -> Bool
prop_ClausuraTransitiva r =
    transitiva (clausuraTransitiva r)

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

-----
-- § Generador de relaciones
-----

-- genSet es un generador de relaciones binarias. Por ejemplo,
-- ghci> sample genRel
-- (fromList [0],fromList [])
-- (fromList [-1,1],fromList [(-1,1)])
-- (fromList [-3,-2],fromList [])
-- (fromList [-2,0,1,6],fromList [(0,0),(6,0)])
-- (fromList [-7,0,2],fromList [(-7,0),(2,0)])
-- (fromList [2,11],fromList [(2,2),(2,11),(11,2),(11,11)])
-- (fromList [-4,-2,1,4,5],fromList [(1,-2),(1,1),(1,5)])
-- (fromList [-4,-3,-2,6,7],fromList [(-3,-4),(7,-3),(7,-2)])
-- (fromList [-9,-7,0,10],fromList [(10,-9)])
-- (fromList [-10,3,8,10],fromList [(3,3),(10,-10)])
-- (fromList [-10,-9,-7,-6,-5,-4,-2,8,12],fromList [])
genRel :: (Arbitrary a, Integral a) => Gen (Rel a)
genRel = do xs <- listOf1 arbitrary
            ys <- listOf (elements [(x,y) | x <- xs, y <- xs])
            return (fromList xs, fromList ys)

instance (Arbitrary a, Integral a) => Arbitrary (Rel a) where
    arbitrary = genRel

```


Relación 29

Algoritmos de ordenación y complejidad

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es presentar una recopilación de los  
-- algoritmos de ordenación y el estudio de su complejidad.  
  
-- -----  
-- § Librerías auxiliares --  
-- -----  
  
import Data.List  
import qualified I1M.ColaDePrioridad as CP  
  
-- -----  
-- § Ordenación por selección --  
-- -----  
  
-- -----  
-- Ejercicio 1.1. Para ordenar una lista xs mediante el algoritmo de  
-- ordenación por selección se selecciona el menor elemento de xs y se  
-- le añade a la ordenación por selección de los restantes. Por ejemplo,  
-- para ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:  
-- ordenaPorSeleccion [3,1,4,1,5,9,2]  
-- = 1 : ordenaPorSeleccion [3,4,1,5,9,2]
```

```

--      = 1 : 1 : ordenaPorSeleccion [3,4,5,9,2]
--      = 1 : 1 : 2 : ordenaPorSeleccion [3,4,5,9]
--      = 1 : 1 : 2 : 3 : ordenaPorSeleccion [4,5,9]
--      = 1 : 1 : 2 : 3 : 4 : ordenaPorSeleccion [5,9]
--      = 1 : 1 : 2 : 3 : 4 : 5 : ordenaPorSeleccion [9]
--      = 1 : 1 : 2 : 3 : 4 : 5 : 9 : ordenaPorSeleccion []
--      = 1 : 1 : 2 : 3 : 4 : 5 : 9 : []
--      = [1,1,2,3,4,5,9]
--
-- Definir la función
--   ordenaPorSeleccion :: Ord a => [a] -> [a]
-- tal que (ordenaPorSeleccion xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--   ordenaPorSeleccion [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

```

```

ordenaPorSeleccion :: Ord a => [a] -> [a]
ordenaPorSeleccion [] = []
ordenaPorSeleccion xs = m : ordenaPorSeleccion (delete m xs)
  where m = minimum xs
-----

```

```

-----
-- Ejercicio 1.2. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorSeleccion [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000].
--
-- ¿Cuál es el orden de complejidad de ordenaPorSeleccion?
-----

```

```

-- El resumen de los tiempos es
--   k    | segs.
--   -----+-----
--   1000 | 0.05
--   2000 | 0.25
--   3000 | 0.58
--   4000 | 1.13

```

```

-- La complejidad de ordenaPorSeleccion es  $O(n^2)$ .
--
-- Las ecuaciones de recurrencia del coste de ordenaPorSeleccion son

```

```
--      T(0) = 1
--      T(n) = 1 + T(n-1) + 2n
--      Luego, T(n) = (n+1)^2 (ver http://bit.ly/1DGsMeW )
```

```
-- -----
-- Ejercicio 1.3. Definir la función
--   ordenaPorSeleccion2 :: Ord a => [a] -> [a]
--   tal que (ordenaPorSeleccion2 xs) es la lista xs ordenada por el
--   algoritmo de selección, pero usando un acumulador. Por ejemplo,
--   ordenaPorSeleccion2 [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-- -----
```

```
ordenaPorSeleccion2 :: Ord a => [a] -> [a]
ordenaPorSeleccion2 [] = []
ordenaPorSeleccion2 (x:xs) = aux xs x []
  where aux [] m r = m : ordenaPorSeleccion2 r
        aux (x:xs) m r | x < m      = aux xs x (m:r)
                       | otherwise = aux xs m (x:r)
```

```
-- -----
-- Ejercicio 1.4. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorSeleccion2 [n,n-1..1])
--   para k en [1000, 2000, 3000, 4000]
-- -----
```

```
-- El resumen de los tiempos es
--   k      | segs.
--   -----+-----
--   1000   | 0.39
--   2000   | 1.53
--   3000   | 3.48
--   4000   | 6.35
```

```
-- -----
-- § Ordenación rápida (Quicksort)
-- -----
```

```
-- -----
-- Ejercicio 2.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación rápida se selecciona el primer elemento x de xs, se divide
```

```

-- los restantes en los menores o iguales que x y en los mayores que x,
-- se ordena cada una de las dos partes y se unen los resultados. Por
-- ejemplo, para ordenar la lista [3,1,4,1,5,9,2] el proceso es el
-- siguiente:
--   or [3,1,4,1,5,9,2]
-- = or [1,1,2] ++ [3] ++ or [4,5,9]
-- = (or [1] ++ [1] ++ or [2]) ++ [3] ++ (or [] ++ [4] ++ or [5,9])
-- = ((or [] ++ [1] ++ or []) ++ [1] ++ (or [] ++ [2] ++ or []))
--   ++ [3] ++ ([4] ++ [5] ++ or [9])
-- = ([1] ++ [1] ++ [2] ++ [3] ++ ([4] ++ ([5] ++ or [9] ++ or [])))
-- = ([1] ++ [1] ++ [2] ++ [3] ++ ([4] ++ ([5] ++ or [9] ++ or [])))
-- = ([1] ++ [1] ++ [2] ++ [3] ++ ([4] ++ ([5] ++ or [9] ++ or [])))
-- = ([1] ++ [1] ++ [2] ++ [3] ++ ([4] ++ ([5] ++ or [9] ++ or [])))
-- = [1,1,2,3,4,5,9]
--
-- Definir la función
--   ordenaRapida :: Ord a => [a] -> [a]
-- tal que (ordenaRapida xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--   ordenaRapida [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

```

```

ordenaRapida :: Ord a => [a] -> [a]
ordenaRapida [] = []
ordenaRapida (x:xs) =
  ordenaRapida menores ++ [x] ++ ordenaRapida mayores
  where menores = [y | y <- xs, y <= x]
        mayores = [y | y <- xs, y > x]

```

```

-----
-- Ejercicio 2.2. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaRapida [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
--
-- ¿Cuál es el orden de complejidad de ordenaRapida?
-----

```



```

-- El resumen de los tiempos es
--   k    | segs.
--   -----+-----
--   1000 |  0.64
--   2000 |  2.57
--   3000 |  6.64
--   4000 | 12.33

-- La complejidad de ordenaRapida es  $O(n \log(n))$ .

-----
-- Ejercicio 2.3. Definir, usando un acumulador, la función
--   ordenaRapida2 :: Ord a => [a] -> [a]
-- tal que (ordenaRapida2 xs) es la lista obtenida ordenando xs
-- por el procedimiento de ordenación rápida. Por ejemplo,
--   ordenaRapida2 [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

ordenaRapida2 :: Ord a => [a] -> [a]
ordenaRapida2 xs = aux xs []
  where aux [] s      = s
        aux (x:xs) s = aux menores (x : (aux mayores s))
          where menores = [y | y <- xs, y <= x]
                mayores = [y | y <- xs, y > x]

-----
-- Ejercicio 2.4. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaRapida2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-----

-- El resumen de los tiempos es
--   k    | segs.
--   -----+-----
--   1000 |  0.56
--   2000 |  2.42
--   3000 |  5.87
--   4000 | 10.93

```

```

-----
-- § Ordenación por inserción
-----

-----
-- Ejercicio 3.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación por inserción se selecciona el primer elemento x de xs, se
-- ordena el resto de xs y se inserta x en su lugar. Por ejemplo, para
-- ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
--   ordenaPorInsercion [3,1,4,1,5,9,2]
--   = 3 : ordenaPorInsercion [1,4,1,5,9,2]
--   = 3 : 1 : ordenaPorInsercion [4,1,5,9,2]
--   = 3 : 1 : 4 : ordenaPorInsercion [1,5,9,2]
--   = 3 : 1 : 4 : 1 : ordenaPorInsercion [5,9,2]
--   = 3 : 1 : 4 : 1 : 5 : ordenaPorInsercion [9,2]
--   = 3 : 1 : 4 : 1 : 5 : 9 : ordenaPorInsercion [2]
--   = 3 : 1 : 4 : 1 : 5 : 9 : 2 : ordenaPorInsercion []
--   = 3 : 1 : 4 : 1 : 5 : 9 : 2 : []
--   = 3 : 1 : 4 : 1 : 5 : 9 : [2]
--   = 3 : 1 : 4 : 1 : 5 : [2,9]
--   = 3 : 1 : 4 : 1 : [2,5,9]
--   = 3 : 1 : 4 : [1,2,5,9]
--   = 3 : 1 : [1,2,4,5,9]
--   = 3 : [1,1,2,4,5,9]
--   = [1,1,2,3,4,5,9]
--
-- Definir la función
--   ordenaPorInsercion :: Ord a => [a] -> [a]
-- tal que (ordenaPorInsercion xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--   ordenaPorInsercion [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

ordenaPorInsercion :: Ord a => [a] -> [a]
ordenaPorInsercion [] = []
ordenaPorInsercion (x:xs) = inserta x (ordenaPorInsercion xs)

-- (inserta x xs) inserta el elemento x después de los elementos de xs
-- que son menores o iguales que x. Por ejemplo,
--   inserta 5 [3,2,6,4] == [3,2,5,6,4]

```

```

inserta :: Ord a => a -> [a] -> [a]
inserta y [] = [y]
inserta y l@(x:xs) | y <= x = y : l
                  | otherwise = x : (inserta y xs)

```

-- 2ª definición de inserta:

```

inserta2 :: Ord a => a -> [a] -> [a]
inserta2 x xs = takeWhile (<= x) xs ++ [x] ++ dropWhile (<=x) xs

```

```

-----
-- Ejercicio 3.2. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorInsercion [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
--
-- ¿Cuál es la complejidad de ordenaPorInsercion?
-----

```

-- El resumen de los tiempos es

```

--   k   | segs.
--   ----+-----
--   1000 | 0.39
--   2000 | 1.53
--   3000 | 3.49
--   4000 | 6.32

```

-- La complejidad de ordenaPorInsercion es $O(n^2)$

-- Las ecuaciones de recurrencia del coste de ordenaPorInsercion son

```

--    $T(0) = 1$ 
--    $T(n) = 4n + T(n-1)$ 
-- Luego,  $T(n) = 2n(n+1)+1$  (ver http://bit.ly/19FmQq4 )

```

-- Ejercicio 3.3. Definir, por plegados, la función

```

--   ordenaPorInsercion2 :: Ord a => [a] -> [a]
-- tal que (ordenaPorInsercion2 xs) es la lista obtenida ordenando xs
-- por el procedimiento de ordenación por inserción. Por ejemplo,
--   ordenaPorInsercion2 [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

```

```
ordenaPorInsercion2 :: Ord a => [a] -> [a]
ordenaPorInsercion2 xs = foldr inserta [] xs
```

```
-----
-- Ejercicio 3.2. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorInsercion2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-----
```

```
-- El resumen de los tiempos es
--   k      | segs.
--   -----+-----
--   1000   | 0.38
--   2000   | 1.54
--   3000   | 3.46
--   4000   | 6.29
```

```
-----
-- § Ordenación por mezcla ("Mergesort")
-----
```

```
-----
-- Ejercicio 4.1. Para ordenar una lista xs mediante el algoritmo de
-- ordenación por mezcla se divide xs por la mitad, se ordena cada una
-- de las partes y se mezclan los resultados. Por ejemplo, para
-- ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
```

```
--   om [3,1,4,1,5,9,2]
--   = m (om [3,1,4]) (om [1,5,9,2])
--   = m (m (om [3]) (om [1,4])) (m (om [1,5]) (om [9,2]))
--   = m (m [3] (m (om [1]) (om [4])))
--       (m (m (om [1]) (om [5])) (m (om [9]) (om [2])))
--   = m (m [3] (m [1] [4]))
--       (m (m [1] [5]) (m [9] [2])))
--   = m (m [3] [1,4]) (m [1,5] [2,9])
--   = m [1,3,4] [1,2,5,9]
--   = [1,1,2,3,4,5,9]
```

```
-- donde om es ordenaPorMezcla y m es mezcla.
```

```
-- Definir la función
```

```
--   ordenaPorMezcla :: Ord a => [a] -> [a]
```

```

-- tal que (ordenaPorMezcla xs) es la lista obtenida ordenando por
-- selección la lista xs. Por ejemplo,
--   ordenaPorMezcla [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

ordenaPorMezcla :: Ord a => [a] -> [a]
ordenaPorMezcla [] = []
ordenaPorMezcla [x] = [x]
ordenaPorMezcla l = mezcla (ordenaPorMezcla l1) (ordenaPorMezcla l2)
  where l1 = take k l
        l2 = drop k l
        k = length l `div` 2

-- (mezcla xs ys) es la lista obtenida mezclando xs e ys. Por ejemplo,
--   mezcla [1,3] [2,4,6] == [1,2,3,4,6]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] b = b
mezcla a [] = a
mezcla a@(x:xs) b@(y:ys) | x <= y = x : (mezcla xs b)
                          | otherwise = y : (mezcla a ys)

-----

-- Ejercicio 4.2. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorMezcla [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
--
-- ¿Cuál es la complejidad de ordenaPorMezcla?
-----

-- El resumen de los tiempos es
--   k    | segs.
--   -----+-----
--   1000 | 0.02
--   2000 | 0.03
--   3000 | 0.05
--   4000 | 0.06

-- La complejidad de ordenaPorMezcla es  $O(n \log(n))$ .
--
-- Las ecuaciones de recurrencia del coste de ordenaPorMezcla son

```

```

--      T(0) = 1
--      T(1) = 1
--      T(n) = n + 2*T(n/n)
--      Luego, T(n) = (c*n)/2+(n log(n))/(log(2)) (ver http://bit.ly/1EyUTYG )

-----
--      Ejercicio 4.3. Otra forma de ordenar una lista xs mediante el
--      algoritmo de ordenación por mezcla consiste en dividir xs en listas
--      unitarias y mezclar los resultados. Por ejemplo, para
--      ordenar la lista [3,1,4,1,5,9,2] el proceso es el siguiente:
--      om [3,1,4,1,5,9,2]
--      = mp [[3],[1],[4],[1],[5],[9],[2]]
--      = mp [[1,3],[1,4],[5,9],[2]]
--      = mp [[1,1,3,4],[2,5,9]]
--      = [1,1,2,3,4,5,9]
--      donde om es ordenaPorMezcla y mp es mezclaPares.
--
--      Definir la función
--      ordenaPorMezcla2 :: Ord a => [a] -> [a]
--      tal que (ordenaPorMezcla2 xs) es la lista obtenida ordenando por
--      selección la lista xs. Por ejemplo,
--      ordenaPorMezcla2 [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
-----

ordenaPorMezcla2 :: Ord a => [a] -> [a]
ordenaPorMezcla2 l = aux (divide l)
  where aux [r] = r
        aux l   = aux (mezclaPares l)

--      (divide xs) es la lista de de las listas unitarias formadas por los
--      elementos de xs. Por ejemplo,
--      divide [3,1,4,1,5,9,2,8] == [[3],[1],[4],[1],[5],[9],[2],[8]]
divide :: Ord a => [a] -> [[a]]
divide xs = [[x] | x <- xs]

--      También se puede definir por recursión
divide2 :: Ord a => [a] -> [[a]]
divide2 [] = []
divide2 (x:xs) = [x] : divide2 xs

```

```

-- (mezclaPares xs) es la lista obtenida mezclando los pares de
-- elementos consecutivos de xs. Por ejemplo,
--   ghci> mezclaPares [[3],[1],[4],[1],[5],[9],[2],[8]]
--   [[1,3],[1,4],[5,9],[2,8]]
--   ghci> mezclaPares [[1,3],[1,4],[5,9],[2,8]]
--   [[1,1,3,4],[2,5,8,9]]
--   ghci> mezclaPares [[1,1,3,4],[2,5,8,9]]
--   [[1,1,2,3,4,5,8,9]]
--   ghci> mezclaPares [[1],[3],[2]]
--   [[1,3],[2]]
mezclaPares :: (Ord a) => [[a]] -> [[a]]
mezclaPares []           = []
mezclaPares [x]         = [x]
mezclaPares (xs:ys:zss) = mezcla xs ys : (mezclaPares zss)

```

```

-----
-- Ejercicio 4.4. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorMezcla2 [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
-----

```

```

-- El resumen de los tiempos es
--   k    | segs.
--   -----+-----
--   1000 | 0.02
--   2000 | 0.03
--   3000 | 0.03
--   4000 | 0.05

```

```

-----
-- § Ordenación por montículos ("heapsort")
-----

```

```

-----
-- Ejercicio 5.1. El procedimiento de ordenación de una lista por
-- montículos consiste en almacenar todos los elementos del vector a
-- ordenar en un montículo (heap), y luego extraer el nodo que queda
-- como nodo raíz del montículo (cima) en sucesivas iteraciones
-- obteniendo el conjunto ordenado.
--

```

```
-- Usando la implementación de las colas de prioridad mediante
-- montículos (que se encuentra en la librería I1M.ColaDePrioridad),
-- definir la función
--   ordenaPorMonticulos :: Ord a => [a] -> [a]
-- tal que (ordenaPorMonticulos xs) es la lista obtenida ordenando xs
-- por el procedimiento de ordenación por montículos. Por ejemplo,
--   ordenaPorMonticulos [3,1,4,1,5,9,2] == [1,1,2,3,4,5,9]
```

```
-----
ordenaPorMonticulos :: Ord a => [a] -> [a]
ordenaPorMonticulos xs = aux (creaCP xs) where
  aux cp | CP.esVacia cp = []
         | otherwise     = CP.primeros cp : aux (CP.resto cp)
```

```
-- (creaCP xs) es la cola de prioridad correspondiente a la lista
-- xs. Por ejemplo,
```

```
ghci> creaCP [3,1,4,1,5,9,2,8]
CP (M 1 2
    (M 1 2
      (M 2 2
        (M 8 1 Vacio Vacio)
        (M 5 1
          (M 9 1 Vacio Vacio)
          Vacio))
      (M 4 1 Vacio Vacio))
    (M 3 1 Vacio Vacio))
```

```
creaCP :: Ord a => [a] -> CP.CPrioridad a
creaCP xs = foldr CP.inserta CP.vacia xs
```

```
-----
-- Ejercicio 5.2. Calcular los tiempos necesarios para calcular
--   let n = k in length (ordenaPorMonticulos [n,n-1..1])
-- para k en [1000, 2000, 3000, 4000]
```

```
--
-- ¿Cuál es la complejidad de ordenaPorMonticulos?
```

```
-----
-- El resumen de los tiempos es
--   k    | segs.
--   -----+-----
```

```
-- 1000 | 0.02
-- 2000 | 0.03
-- 3000 | 0.04
-- 4000 | 0.05
```

```
-- La complejidad de ordenaPorMonticulos es  $O(n \log(n))$ .
```


Relación 30

El problema del granjero mediante búsqueda en espacio de estado

-- *Introducción* -----

-- *Un granjero está parado en un lado del río y con él tiene un lobo,
-- una cabra y una repollo. En el río hay un barco pequeño. El granjero
-- desea cruzar el río con sus tres posesiones. No hay puentes y en el
-- barco hay solamente sitio para el granjero y un artículo. Si deja
-- la cabra con la repollo sola en un lado del río la cabra comerá la
-- repollo. Si deja el lobo y la cabra en un lado, el lobo se comerá a
-- la cabra. ¿Cómo puede cruzar el granjero el río con los tres
-- artículos, sin que ninguno se coma al otro?*

-- *El objetivo de esta relación de ejercicios es resolver el problema
-- del granjero mediante búsqueda en espacio de estados, utilizando las
-- implementaciones estudiadas en el tema 23 que se pueden descargar desde
-- <http://www.cs.us.es/~jalonso/cursos/ilm-14/codigos>*

-- *Las transparencias del tema 23 se encuentran en
-- <http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-23.pdf>*

-- *Importaciones* -----

```
-----  
import I1M.BusquedaEnEspaciosDeEstados -- Está en http://bit.ly/1AKmUQB  
  
-----  
-- Ejercicio 1. Definir el tipo Orilla con dos constructores I y D que  
-- representan las orillas izquierda y derecha, respectivamente.  
-----  
data Orilla = I | D  
             deriving (Eq, Show)  
  
-----  
-- Ejercicio 2. Definir el tipo Estado como abreviatura de una tupla que  
-- representan en qué orilla se encuentra cada uno de los elementos  
-- (granjero, lobo, cabra, repollo). Por ejemplo, (I,D,D,I) representa  
-- que el granjero está en la izquierda, que el lobo está en la derecha,  
-- que la cabra está en la derecha y el repollo está en la izquierda.  
-----  
type Estado = (Orilla,Orilla,Orilla,Orilla)  
  
-----  
-- Ejercicio 3. Definir  
--   inicial :: Estado  
-- tal que inicial representa el estado en el que todos están en la  
-- orilla izquierda.  
-----  
inicial:: Estado  
inicial = (I,I,I,I)  
  
-----  
-- Ejercicio 4. Definir  
--   final :: Estado  
-- tal que final representa el estado en el que todos están en la  
-- orilla derecha.  
-----  
final:: Estado
```

```
final = (D,D,D,D)
```

```
-----  
-- Ejercicio 5. Definir la función  
-- seguro :: Estado -> Bool  
-- tal que (seguro e) se verifica si el estado e es seguro; es decir,  
-- que no puede estar en una orilla el lobo con la cabra sin el granjero  
-- ni la cabra con el repollo sin el granjero. Por ejemplo,  
-- seguro (I,D,D,I) == False  
-- seguro (D,D,D,I) == True  
-- seguro (D,D,I,I) == False  
-- seguro (I,D,I,I) == True  
-----
```

```
seguro :: Estado -> Bool  
seguro (g,l,c,r)  
  | l == c    = g == l  
  | c == r    = g == c  
  | otherwise = True
```

```
-----  
-- Ejercicio 6. Definir la función  
-- opuesta :: Orilla -> Orilla  
-- tal que (opuesta x) es la opuesta de la orilla x. Por ejemplo  
-- opuesta I = D  
-----
```

```
opuesta :: Orilla -> Orilla  
opuesta I = D  
opuesta D = I
```

```
-----  
-- Ejercicio 7. Definir la función  
-- sucesoresE :: Estado -> [Estado]  
-- tal que (sucesoresE e) es la lista de los sucesores seguros del  
-- estado e. Por ejemplo,  
-- sucesoresE (D,I,D,I) == [(I,I,D,I),(I,I,I,I)]  
-----
```

```
sucesoresE :: Estado -> [Estado]
```

```

sucesoresE e = [mov e | mov <- [m1,m2,m3,m4], seguro (mov e)]
  where m1 (g,l,c,r) = (opuesta g, l, c, r)
        m2 (g,l,c,r) = (opuesta g, opuesta l, c, r)
        m3 (g,l,c,r) = (opuesta g, l, opuesta c, r)
        m4 (g,l,c,r) = (opuesta g, l, c, opuesta r)

-----
-- Ejercicio 8. Los nodos del espacio de búsqueda son lista de estados
--   [e_n, ..., e_2, e_1]
-- donde e_1 es el estado inicial y para cada i (2 <= i <= n), e_i es un
-- sucesor de e_(i-1).
--
-- Definir el tipo de datos NodoRio para representar los nodos del
-- espacio de búsqueda. Por ejemplo,
--   ghci> :type (Nodo [(I,I,D,I),(I,I,I,I)])
--   (Nodo [(I,I,D,I),(I,I,I,I)]) :: NodoRio
-----

data NodoRio = Nodo [Estado]
              deriving (Eq, Show)

-----
-- Ejercicio 9. Definir la función
--   sucesoresN :: NodoRio -> [NodoRio]
-- tal que (sucesoresN n) es la lista de los sucesores del nodo n. Por
-- ejemplo,
--   ghci> sucesoresN (Nodo [(I,I,D,I),(D,I,D,I),(I,I,I,I)])
--   [Nodo [(D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)],
--     Nodo [(D,I,D,D),(I,I,D,I),(D,I,D,I),(I,I,I,I)]]
-----

sucesoresN :: NodoRio -> [NodoRio]
sucesoresN (Nodo (n@(e:es))) =
  [Nodo (e':n) | e' <- sucesoresE e, notElem e' es]

-----
-- Ejercicio 10. Definir la función
--   esFinal :: NodoRio -> Bool
-- tal que (esFinal n) se verifica si n es un nodo final; es decir, su
-- primer elemento es el estado final. Por ejemplo,

```

```
--     esFinal (Nodo [(D,D,D,D),(I,I,I,I)]) == True
--     esFinal (Nodo [(I,I,D,I),(I,I,I,I)]) == False
-----

esFinal :: Nodorio -> Bool
esFinal (Nodo (n:_)) = n == final

-----

-- Ejercicio 11. Definir la función
-- granjeroEE :: [Nodorio]
-- tal que granjeroEE son las soluciones del problema del granjero
-- mediante el patrón de búsqueda en espacio de estados. Por ejemplo,
-- ghci> head granjeroEE
--     Nodo [(D,D,D,D),(I,D,I,D),(D,D,I,D),(I,D,I,I),
--           (D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)]
-----

granjeroEE :: [Nodorio]
granjeroEE = buscaEE sucesoresN
              esFinal
              (Nodo [inicial])
```


Relación 31

El problema del calendario mediante búsqueda en espacio de estado

```
-----  
-- Introducción --  
-----  
  
-- El problema del calendario, para una competición deportiva en la que  
-- se enfrentan n participantes, consiste en elaborar un calendario de  
-- forma que:  
-- + el campeonato dure n-1 días,  
-- + cada participante juegue exactamente un partido diario y  
-- + cada participante juegue exactamente una vez con cada adversario.  
-- Por ejemplo, con 8 participantes una posible solución es  
-- | 1 2 3 4 5 6 7  
-- ---+-----  
-- 1 | 2 3 4 5 6 7 8  
-- 2 | 1 4 3 6 5 8 7  
-- 3 | 4 1 2 7 8 5 6  
-- 4 | 3 2 1 8 7 6 5  
-- 5 | 6 7 8 1 2 3 4  
-- 6 | 5 8 7 2 1 4 3  
-- 7 | 8 5 6 3 4 1 2  
-- 8 | 7 6 5 4 3 2 1  
-- donde las filas indican los jugadores y las columnas los días; es  
-- decir, el elemento (i,j) indica el adversario del jugador i el día j;
```

```
-- por ejemplo, el adversario del jugador 2 el 4ª día es el jugador 6.
--
-- El objetivo de esta relación de ejercicios es resolver el problema
-- del calendario mediante búsqueda en espacio de estados, utilizando las
-- implementaciones estudiadas en el tema 23 que se pueden descargar desde
--   http://www.cs.us.es/~jalonso/cursos/ilm-14/codigos
--
-- Las transparencias del tema 23 se encuentran en
--   http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-23.pdf
```

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

```
import I1M.BusquedaEnEspaciosDeEstados
import Data.Matrix
import Data.List
```

```
-----
-- Ejercicio 1. Definir el tipo Calendario como una matriz de números
-- enteros.
-----
```

```
type Calendario = Matrix Int
```

```
-----
-- Ejercicio 2. Definir la función
--   inicial :: Int -> Calendario
-- tal que (inicial n) es el estado inicial para el problema del
-- calendario con n participantes; es decir, una matriz de n fila y n-1
-- columnas con todos sus elementos iguales a 0. Por ejemplo,
--   ghci> inicial 4
--   ( 0 0 0 )
--   ( 0 0 0 )
--   ( 0 0 0 )
--   ( 0 0 0 )
-----
```

```
inicial :: Int -> Calendario
inicial n = zero n (n-1)
```

```

-----
-- Ejercicio 3. Definir la función
--   sucesores :: Int -> Calendario -> [Calendario]
-- tal que (sucesores n c) es la lista de calendarios, para el problema
-- con n participantes, obtenidos poniendo en el lugar del primer
-- elemento nulo de c uno de los posibles jugadores de forma que se
-- cumplan las condiciones del problema. Por ejemplo,
--   ghci> sucesores 4 (fromLists [[2,3,0],[1,0,0],[0,1,0],[0,0,0]])
--   [( 2 3 4 )
--    ( 1 0 0 )
--    ( 0 1 0 )
--    ( 0 0 1 )]
--   ghci> sucesores 4 (fromLists [[2,3,4],[1,0,0],[0,1,0],[0,0,1]])
--   [( 2 3 4 )
--    ( 1 4 0 )
--    ( 0 1 0 )
--    ( 0 2 1 )]
-----

sucesores :: Int -> Calendario -> [Calendario]
sucesores n c =
  [setElem i (k,j) (setElem k (i,j) c) |
   k <- [1..n] \\ (i : [c!(k,j) | k <- [1..i-1]]
                 ++ [c!(i,k) | k <- [1..j-1]]),
   c!(k,j) == 0]
  where (i,j) = head [(i,j) | i <- [1..n], j <- [1..n-1], c!(i,j) == 0]

-----
-- Ejercicio 4. Definir la función
--   esFinal :: Int -> Calendario -> Bool
-- tal que (final n c) se verifica si c un estado final para el problema
-- del calendario con n participantes; es decir, no queda en c ningún
-- elemento igual a 0. Por ejemplo,
--   ghci> esFinal 4 (fromLists [[2,3,4],[1,4,3],[4,1,2],[3,2,1]])
--   True
--   ghci> esFinal 4 (fromLists [[2,3,4],[1,4,3],[4,1,2],[3,2,0]])
--   False
-----

```

```
esFinal :: Int -> Calendario -> Bool
esFinal n c = null [(i,j) | i <- [1..n], j <- [1..n-1], c!(i,j) == 0]

-----

-- Ejercicio 5. Definir la función
--   calendario :: Int -> [Calendario]
-- tal que (calendario n) son las soluciones del problema del calendario,
-- con n participantes, mediante el patrón de búsqueda en espacio de
-- estados. Por ejemplo,
--   ghci> head (calendario 6)
--   ( 2 3 4 5 6 )
--   ( 1 4 5 6 3 )
--   ( 5 1 6 4 2 )
--   ( 6 2 1 3 5 )
--   ( 3 6 2 1 4 )
--   ( 4 5 3 2 1 )
--
--   ghci> length (calendario 6)
--   720
--   ghci> length (calendario 5)
--   0

-----

calendario :: Int -> [Calendario]
calendario n = buscaEE (sucesores n)
                (esFinal n)
                (inicial n)
```

Relación 32

Estadística descriptiva

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación es definir las principales medidas  
-- estadísticas de centralización (medias, mediana y modas) y de  
-- dispersión (rango, desviación media, varianza y desviación típica)  
-- que se estudian en 3º de ESO (como en http://bit.ly/1yXc7mv ).  
  
-----  
-- Librerías auxiliares --  
-----  
  
import Data.List  
import Data.Map (Map)  
import qualified Data.Map as M  
import Test.QuickCheck  
  
-----  
-- Medidas de centralización --  
-----  
  
-----  
-- Ejercicio 1. Definir la función  
-- media :: Floating a => [a] -> a  
-- tal que (media xs) es la media aritmética de los números de la lista  
-- xs. Por ejemplo,
```

```

--      media [4,8,4,5,9] == 6.0
-----

media :: Floating a => [a] -> a
media xs = sum xs / genericLength xs

-----

-- Ejercicio 2. La mediana de una lista de valores es el valor de
-- la lista que ocupa el lugar central de los valores ordenados de menor
-- a mayor. Si el número de datos es impar se toma como valor de la
-- mediana el valor central. Si el número de datos es par se toma como
-- valor de la mediana la media aritmética de los dos valores
-- centrales.
--
-- Definir la función
--   mediana :: (Floating a, Ord a) => [a] -> a
-- tal que (mediana xs) es la mediana de la lista xs. Por ejemplo,
--   mediana [2,3,6,8,9]    == 6.0
--   mediana [2,3,4,6,8,9] == 5.0
--   mediana [9,6,8,4,3,2] == 5.0
-----

mediana :: (Floating a, Ord a) => [a] -> a
mediana xs | odd n  = head $ drop (n `div` 2) ys
           | even n = media $ take 2 $ drop i ys
  where ys = sort xs
        n  = length xs
        i  = (n `div` 2) - 1

-----

-- Ejercicio 3. Comprobar con QuickCheck que para cualquier lista no
-- vacía xs el número de elementos de xs menores que su median es menor
-- o igual que la mitad de los elementos de xs y lo mismo pasa con los
-- mayores o iguales que la mediana.
-----

-- La propiedad es
prop_mediana :: (Floating a, Ord a) => [a] -> Property
prop_mediana xs =
  not (null xs) ==>

```

```

genericLength [x | x <- xs, x < m] <= n/2 &&
genericLength [x | x <- xs, x > m] <= n/2
where m = mediana xs
      n = genericLength xs

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

-----

-- Ejercicio 4. Definir la función
--   frecuenciasD :: Ord a => [a] -> Map a Int
-- tal que (frecuenciasD xs) es el diccionario formado por los elementos
-- de xs junto con el número de veces que aparecen en xs. Por ejemplo,
--   frecuenciasD "sosos"           == M.fromList [('o',2),('s',3)]
--   frecuenciasD (show (10^100)) == M.fromList [('0',100),('1',1)]
-----

-- 1ª definición (por recursión):
frecuenciasD1 :: Ord a => [a] -> Map a Int
frecuenciasD1 []      = M.empty
frecuenciasD1 (x:xs) = M.insertWith (+) x 1 (frecuenciasD1 xs)

-- 2ª definición (por plegado)
frecuenciasD2 :: Ord a => [a] -> Map a Int
frecuenciasD2 = foldl' (\d x-> M.insertWith' (+) x 1 d) M.empty

-- Comparación de eficiencia
--   ghci> frecuenciasD1 (take (10^6) (cycle "abc"))
--   fromList [('a',333334),('b',333333),('c',333333)]
--   (3.85 secs, 479811608 bytes)
--   ghci> frecuenciasD2 (take (10^6) (cycle "abc"))
--   fromList [('a',333334),('b',333333),('c',333333)]
--   (0.72 secs, 218612192 bytes)
--
--   ghci> size (frecuenciasD1 (take (10^6) (cycle [1..10^6])))
--   1000000
--   (11.35 secs, 2435622280 bytes)
--   ghci> size (frecuenciasD2 (take (10^6) (cycle [1..10^6])))
--   1000000

```

```

--      (2.83 secs, 1621887768 bytes)

-- En lo sucesivo, usaremos la 2ª definición:
frecuenciasD :: Ord a => [a] -> Map a Int
frecuenciasD = frecuenciasD2

-----

-- Ejercicio 5. Definir la función
--   frecuencias :: Ord a => [a] -> [(a,Int)]
-- tal que (frecuencias xs) es la lista formada por los elementos de xs
-- junto con el número de veces que aparecen en xs. Por ejemplo,
--   frecuencias "sosos"      == [('o',2),('s',3)]
--   frecuencias (show (10^100)) == [('0',100),('1',1)]
-----

-- 1ª definición (usando frecuenciasD):
frecuencias1 :: Ord a => [a] -> [(a,Int)]
frecuencias1 = M.toList . frecuenciasD

-- 2ª definición (sin usar frecuenciasD):
frecuencias2 :: Ord a => [a] -> [(a,Int)]
frecuencias2 xs = [(x,1 + length xs) | (x:xs) <- group (sort xs)]

-- En lo sucesivo usaremos la 1ª definición
frecuencias :: Ord a => [a] -> [(a,Int)]
frecuencias = frecuencias1

-----

-- Ejercicio 6. Las modas de una lista son los elementos de la lista
-- que más se repiten.
--
-- Definir la función
--   modas :: Ord a => [a] -> [a]
-- tal que (modas xs) es la lista ordenada de las modas de xs. Por
-- ejemplo
--   modas [7,3,7,5,3,1,6,9,6] == [3,6,7]
-----

-- 1ª definición (con frecuenciasD):
modas1 :: Ord a => [a] -> [a]

```



```

modas1 xs = M.keys (M.filter (==m) d)
  where d = frecuenciasD xs
        m = maximum (M.elems d)

```

```
-- 2ª definición (con frecuencias):
```

```

modas2 :: Ord a => [a] -> [a]
modas2 xs = [y | (y,n) <- ys, n == m]
  where ys = frecuencias xs
        m = maximum (map snd ys)

```

```
-- Comparación de eficiencia:
```

```

-- ghci> modas1 (1:[1..10^6])
-- [1]
-- (3.83 secs, 1638455984 bytes)
-- ghci> modas2 (1:[1..10^6])
-- [1]
-- (4.72 secs, 1798442320 bytes)

```

```
-- En lo que sigue usaremos la 1ª definición
```

```

modas :: Ord a => [a] -> [a]
modas = modas1

```

```

-----
-- Ejercicio 7. La media geométrica de una lista de n números es la
-- raíz n-ésima del producto de todos los números.

```

```
-- Definir la función
```

```

-- mediaGeometrica :: Floating a => [a] -> a
-- tal que (mediaGeometrica xs) es la media geométrica de xs. Por
-- ejemplo,
-- mediaGeometrica [2,18] == 6.0
-- mediaGeometrica [3,1,9] == 3.0

```

```

-----
mediaGeometrica :: Floating a => [a] -> a
mediaGeometrica xs = (product xs)**(1 / genericLength xs)

```

```

-----
-- Ejercicio 8. Comprobar con QuickCheck que la media geométrica de
-- cualquier lista no vacía de números no negativos es siempre menor o

```

```

-- igual que la media aritmética.
-----

-- La propiedad es
prop_mediaGeometrica :: (Floating a, Ord a) => [a] -> Property
prop_mediaGeometrica xs =
  not (null xs) ==>
  mediaGeometrica ys <= media ys
  where ys = map abs xs

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

-----

-- Medidas de dispersión
-----

-----

-- Ejercicio 9. El recorrido (o rango) de una lista de valores es la
-- diferencia entre el mayor y el menor.
--
-- Definir la función
--   rango :: (Num a, Ord a) => [a] -> a
-- tal que (rango xs) es el rango de xs. Por ejemplo,
--   rango [4,2,4,7,3] == 5
-----

rango :: (Num a, Ord a) => [a] -> a
rango xs = maximum xs - minimum xs

-----

-- Ejercicio 10. La desviación media de una lista de datos xs es la
-- media de las distancias de los datos a la media xs, donde la
-- distancia entre dos elementos es el valor absoluto de su
-- diferencia. Por ejemplo, la desviación media de [4,8,4,5,9] es 2 ya
-- que la media de [4,8,4,5,9] es 6 y
--   (|4-6| + |8-6| + |4-6| + |5-6| + |9-6|) / 5
--   = (2 + 2 + 2 + 1 + 3) / 5
--   = 2

```

```
--
-- Definir la función
--   desviacionMedia :: Floating a => [a] -> a
-- tal que (desviacionMedia xs) es la desviación media de xs. Por
-- ejemplo,
--   desviacionMedia [4,8,4,5,9]      == 2.0
--   desviacionMedia (replicate 10 3) == 0.0
```

```
-----
desviacionMedia :: Floating a => [a] -> a
desviacionMedia xs = media $ map (\x -> abs(x - m)) xs
  where m = media xs
```

```
-----
-- Ejercicio 11. La varianza de una lista datos es la media de los
-- cuadrados de las distancias de los datos a la media. Por ejemplo, la
-- varianza de [4,8,4,5,9] es 4.4 ya que la media de [4,8,4,5,9] es 6 y
--   ((4-6)^2 + (8-6)^2 + (4-6)^2 + (5-6)^2 + (9-6)^2) / 5
--   = (4 + 4 + 4 + 1 + 9) / 5
--   = 4.4
```

```
--
-- Definir la función
--   varianza :: Floating a => [a] -> a
-- tal que (desviacionMedia xs) es la varianza de xs. Por ejemplo,
--   varianza [4,8,4,5,9]      == 4.4
--   varianza (replicate 10 3) == 0.0
```

```
-----
varianza :: Floating a => [a] -> a
varianza xs = media $ map (\x -> (x - m)^2) xs
  where m = media xs
```

```
-----
-- Ejercicio 12. La desviación típica de una lista de datos es la raíz
-- cuadrada de su varianza.
```

```
--
-- Definir la función
--   desviacionTipica :: Floating a => [a] -> a
-- tal que (desviacionTipica xs) es la desviación típica de xs. Por
-- ejemplo,
```

```
-- desviacionTipica [4,8,4,5,9]      == 2.0976176963403033  
-- desviacionTipica (replicate 10 3) == 0.0
```

```
desviacionTipica :: Floating a => [a] -> a  
desviacionTipica = sqrt . varianza
```

Relación 33

Números de Lychrel

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

```
-- Según la Wikipedia, un número de Lychrel es un número natural para  
-- el que nunca se obtiene un capicúa mediante el proceso de invertir  
-- las cifras y sumar los dos números. Por ejemplo, los siguientes  
-- números no son números de Lychrel:  
-- * 56, ya que en un paso se obtiene un capicúa:  $56+65=121$ .  
-- * 57, ya que en dos pasos se obtiene un capicúa:  $57+75=132$ ,  
--    $132+231=363$   
-- * 59, ya que en dos pasos se obtiene un capicúa:  $59+95=154$ ,  
--    $154+451=605$ ,  $605+506=1111$   
-- * 89, ya que en 24 pasos se obtiene un capicúa.  
-- En esta relación vamos a buscar el primer número de Lychrel.
```

```
-----  
-- Librerías auxiliares  
-----
```

```
import Test.QuickCheck
```

```
-----  
-- Ejercicio 1. Definir la función  
--   esCapicua :: Integer -> Bool  
-- tal que (esCapicua x) se verifica si x es capicúa. Por ejemplo,  
--   esCapicua 252 == True
```

```
-- esCapicua 253 == False
```

```
-----
esCapicua :: Integer -> Bool
esCapicua x = x' == reverse x'
  where x' = show x
```

```
-----
-- Ejercicio 2. Definir la función
-- inverso :: Integer -> Integer
-- tal que (inverso x) es el número obtenido escribiendo las cifras de x
-- en orden inverso. Por ejemplo,
-- inverso 253 == 352
-----
```

```
inverso :: Integer -> Integer
inverso = read . reverse . show
```

```
-----
-- Ejercicio 3. Definir la función
-- siguiente :: Integer -> Integer
-- tal que (siguiente x) es el número obtenido sumándole a x su
-- inverso. Por ejemplo,
-- siguiente 253 == 605
-----
```

```
siguiente :: Integer -> Integer
siguiente x = x + inverso x
```

```
-----
-- Ejercicio 4. Definir la función
-- busquedaDeCapicua :: Integer -> [Integer]
-- tal que (busquedaDeCapicua x) es la lista de los números tal que el
-- primero es x, el segundo es (siguiente de x) y así sucesivamente
-- hasta que se alcanza un capicúa. Por ejemplo,
-- busquedaDeCapicua 253 == [253,605,1111]
-----
```

```
busquedaDeCapicua :: Integer -> [Integer]
busquedaDeCapicua x | esCapicua x = [x]
```

```
| otherwise = x : busquedaDeCapicua (siguiente x)
```

```
-----  
-- Ejercicio 5. Definir la función  
--   capicuaFinal :: Integer -> Integer  
-- tal que (capicuaFinal x) es la capicúa con la que termina la búsqueda  
-- de capicúa a partir de x. Por ejemplo,  
--   capicuaFinal 253 == 1111  
-----
```

```
capicuaFinal :: Integer -> Integer  
capicuaFinal x = last (busquedaDeCapicua x)
```

```
-----  
-- Ejercicio 6. Definir la función  
--   orden :: Integer -> Integer  
-- tal que (orden x) es el número de veces que se repite el proceso de  
-- calcular el inverso a partir de x hasta alcanzar un número  
-- capicúa. Por ejemplo,  
--   orden 253 == 2  
-----
```

```
orden :: Integer -> Integer  
orden x | esCapicua x = 0  
        | otherwise   = 1 + orden (siguiente x)
```

```
-----  
-- Ejercicio 7. Definir la función  
--   ordenMayor :: Integer -> Integer -> Bool  
-- tal que (ordenMayor x n) se verifica si el orden de x es mayor o  
-- igual que n. Dar la definición sin necesidad de evaluar el orden de  
-- x. Por ejemplo,  
--   ghci> ordenMayor 1186060307891929990 2  
--   True  
--   ghci> orden 1186060307891929990  
--   261  
-----
```

```
ordenMayor :: Integer -> Integer -> Bool  
ordenMayor x n | esCapicua x = n == 0
```

```
| n <= 0      = True
| otherwise   = ordenMayor (siguiente x) (n-1)
```

```
-----
-- Ejercicio 8. Definir la función
--   ordenEntre :: Integer -> Integer -> [Integer]
-- tal que (ordenEntre m n) es la lista de los elementos cuyo orden es
-- mayor o igual que m y menor que n. Por ejemplo,
--   take 5 (ordenEntre 10 11) == [829,928,9059,9149,9239]
-----
```

```
ordenEntre :: Integer -> Integer -> [Integer]
ordenEntre m n = [x | x <- [1..], ordenMayor x m, not (ordenMayor x n)]
```

```
-----
-- Ejercicio 9. Definir la función
--   menorDeOrdenMayor :: Integer -> Integer
-- tal que (menorDeOrdenMayor n) es el menor elemento cuyo orden es
-- mayor que n. Por ejemplo,
--   menorDeOrdenMayor 2  == 19
--   menorDeOrdenMayor 20 == 89
-----
```

```
menorDeOrdenMayor :: Integer -> Integer
menorDeOrdenMayor n = head [x | x <- [1..], ordenMayor x n]
```

```
-----
-- Ejercicio 10. Definir la función
--   menoresdDeOrdenMayor :: Integer -> [(Integer,Integer)]
-- tal que (menoresdDeOrdenMayor m) es la lista de los pares (n,x) tales
-- que n es un número entre 1 y m y x es el menor elemento de orden
-- mayor que n. Por ejemplo,
--   menoresdDeOrdenMayor 5 == [(1,10),(2,19),(3,59),(4,69),(5,79)]
-----
```

```
menoresdDeOrdenMayor :: Integer -> [(Integer,Integer)]
menoresdDeOrdenMayor m = [(n,menorDeOrdenMayor n) | n <- [1..m]]
```

```
-----
-- Ejercicio 11. A la vista de los resultados de (menoresdDeOrdenMayor 5)
```



```
-- conjeturar sobre la última cifra de menorDeOrdenMayor.
-- -----

-- Solución: La conjetura es que para n mayor que 1, la última cifra de
-- (menorDeOrdenMayor n) es 9.

-- -----

-- Ejercicio 12. Decidir con QuickCheck la conjetura.
-- -----

-- La conjetura es
prop_menorDeOrdenMayor :: Integer -> Property
prop_menorDeOrdenMayor n =
  n > 1 ==> [last (show (menorDeOrdenMayor n))] == "9"

-- La comprobación es
-- ghci> quickCheck prop_menorDeOrdenMayor
-- *** Failed! Falsifiable (after 22 tests and 2 shrinks):
-- 25

-- Se puede comprobar que 25 es un contraejemplo,
-- ghci> menorDeOrdenMayor 25
-- 196

-- -----

-- Ejercicio 13. Calcular (menoresdDeOrdenMayor 50)
-- -----

-- Solución: El cálculo es
-- ghci> menoresdDeOrdenMayor 50
-- [(1,10), (2,19), (3,59), (4,69), (5,79), (6,79), (7,89), (8,89), (9,89),
-- (10,89), (11,89), (12,89), (13,89), (14,89), (15,89), (16,89), (17,89),
-- (18,89), (19,89), (20,89), (21,89), (22,89), (23,89), (24,89), (25,196),
-- (26,196), (27,196), (28,196), (29,196), (30,196), (31,196), (32,196),
-- (33,196), (34,196), (35,196), (36,196), (37,196), (38,196), (39,196),
-- (40,196), (41,196), (42,196), (43,196), (44,196), (45,196), (46,196),
-- (47,196), (48,196), (49,196), (50,196)]

-- -----

-- Ejercicio 14. A la vista de (menoresdDeOrdenMayor 50), conjeturar el
```

```
-- orden de 196.
```

```
-----  
-- Solución: El orden de 196 es infinito y, por tanto, 196 es un número  
-- del Lychrel.
```

```
-----  
-- Ejercicio 15. Comprobar con QuickCheck la conjetura sobre el orden de  
-- 196.
```

```
-----  
-- La propiedad es  
prop_ordenDe196 n =  
  ordenMayor 196 n
```

```
-- La comprobación es  
-- ghci> quickCheck prop_ordenDe196  
-- +++ OK, passed 100 tests.
```

```
-----  
-- Nota. En el ejercicio anterior sólo se ha comprobado la conjetura de  
-- que 196 es un número de Lychrel. Otra cuestión distinta es  
-- probarla. Hasta la fecha, no se conoce ninguna demostración ni  
-- refutación de la conjetura 196.  
-----
```

Relación 34

Resolución de problemas mediante búsqueda en espacios de estados

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es resolver problemas  
-- mediante búsqueda en espacio de estados, utilizando las  
-- implementaciones estudiadas en el tema 23 que se pueden descargar  
-- desde  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/codigos  
--  
-- Las transparencias del tema 23 se encuentran en  
--  http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-23.pdf  
  
-----  
-- § Librerías auxiliares --  
-----  
  
import I1M.BusquedaEnEspaciosDeEstados  
import Data.List (delete, nub, sort)  
  
-----  
-- Ejercicio 1. Las fichas del dominó se pueden representar por pares de  
-- números enteros. El problema del dominó consiste en colocar todas las
```

```

-- fichas de una lista dada de forma que el segundo número de cada ficha
-- coincida con el primero de la siguiente.
--
-- Definir, mediante búsqueda en espacio de estados, la función
--   domino :: [(Int,Int)] -> [[(Int,Int)]]
-- tal que (domino fs) es la lista de las soluciones del problema del
-- dominó correspondiente a las fichas fs. Por ejemplo,
--   ghci> domino [(1,2),(2,3),(1,4)]
--   [[(4,1),(1,2),(2,3)],[(3,2),(2,1),(1,4)]]
--   ghci> domino [(1,2),(1,1),(1,4)]
--   [[(4,1),(1,1),(1,2)],[(2,1),(1,1),(1,4)]]
--   ghci> domino [(1,2),(3,4),(2,3)]
--   [[(1,2),(2,3),(3,4)]]
--   ghci> domino [(1,2),(2,3),(5,4)]
--   []

```

```

type Ficha = (Int,Int)

```

```

-- Los estados son los pares formados por la listas sin colocar y las
-- colocadas.

```

```

type EstadoDomino = ([Ficha],[Ficha])

```

```

inicialDomino :: [Ficha] -> EstadoDomino

```

```

inicialDomino fs = (fs,[])

```

```

esFinalDomino :: EstadoDomino -> Bool

```

```

esFinalDomino (fs,_) = null fs

```

```

sucesoresDomino :: EstadoDomino -> [EstadoDomino]

```

```

sucesoresDomino (fs,[]) = [(delete f fs, [f]) | f <- fs]

```

```

sucesoresDomino (fs,n@((x,y):qs)) =

```

```

  [(delete (u,v) fs,(u,v):n) | (u,v) <- rs, u /= v, v == x] ++

```

```

  [(delete (u,v) fs,(v,u):n) | (u,v) <- rs, u /= v, u == x] ++

```

```

  [(delete (u,v) fs,(u,v):n) | (u,v) <- rs, u == v, u == x]

```

```

  where rs = [(u,v) | (u,v) <- fs, (u,v) 'notElem' n, (v,u) 'notElem' n]

```

```

solucionesDomino :: [Ficha] -> [EstadoDomino]

```

```

solucionesDomino ps = buscaEE sucesoresDomino
                        esFinalDomino

```

```
(inicialDomino ps)
```

```
domino :: [(Int,Int)] -> [[(Int,Int)]]
domino ps = map snd (solucionesDomino ps)
```

```
-----
-- Ejercicio 2. El problema de suma cero consiste en, dado el conjunto
-- de números enteros, encontrar sus subconjuntos no vacío cuyos
-- elementos sumen cero.
--
-- Definir, mediante búsqueda en espacio de estados, la función
-- suma0 :: [Int] -> [[Int]]
-- tal que (suma0 ns) es la lista de las soluciones del problema se suma
-- cero para ns. Por ejemplo,
-- ghci> suma0 [-7,-3,-2,5,8]
-- [[-3,-2,5]]
-- ghci> suma0 [-7,-3,-2,5,8,-1]
-- [[-7,-3,-2,-1,5,8],[-7,-1,8],[-3,-2,5]]
-- ghci> suma0 [-7,-3,1,5,8]
-- []
-----
```

```
-- Los estados son ternas formadas por los números seleccionados, su
-- suma y los restantes números.
```

```
type EstadoSuma0 = ([Int], Int, [Int])
```

```
inicialSuma0 :: [Int] -> EstadoSuma0
inicialSuma0 ns = ([],0,ns)
```

```
esFinalSuma0 :: EstadoSuma0 -> Bool
esFinalSuma0 (xs,s,_) = not (null xs) && s == 0
```

```
sucesoresSuma0 :: EstadoSuma0 -> [EstadoSuma0]
sucesoresSuma0 (xs,s,ns) = [(n:xs, n+s, delete n ns) | n <- ns]
```

```
solucionesSuma0 :: [Int] -> [EstadoSuma0]
solucionesSuma0 ns = buscaEE sucesoresSuma0
                    esFinalSuma0
                    (inicialSuma0 ns)
```

```

suma0 :: [Int] -> [[Int]]
suma0 ns = nub [sort xs | (xs,_,_) <- solucionesSuma0 ns]

-----
-- Ejercicio 3. Se tienen dos jarras, una de 4 litros de capacidad y
-- otra de 3. Ninguna de ellas tiene marcas de medición. Se tiene una
-- bomba que permite llenar las jarras de agua. El problema de las
-- jarras consiste en determinar cómo se puede lograr tener exactamente
-- 2 litros de agua en la jarra de 4 litros de capacidad.
--
-- Definir, mediante búsqueda en espacio de estados, la función
--   jarras :: [(Int,Int)]
-- tal que su valor es la lista de las soluciones del problema de las
-- jarras, Por ejemplo,
--   ghci> jarras !! 4
--   [(0,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)]
-- La interpretación de primera solución es:
--   (0,0) se inicia con las dos jarras vacías,
--   (4,0) se llena la jarra de 4 con el grifo,
--   (1,3) se llena la de 3 con la de 4,
--   (1,0) se vacía la de 3,
--   (0,1) se pasa el contenido de la primera a la segunda,
--   (4,1) se llena la primera con el grifo,
--   (2,3) se llena la segunda con la primera.
-----

-- Un estado es una lista de dos números. El primero es el contenido de
-- la jarra de 4 litros y el segundo el de la de 3 litros.
type EstadoJarras = (Int,Int)

inicialJarras :: EstadoJarras
inicialJarras = (0,0)

esFinalJarras :: EstadoJarras -> Bool
esFinalJarras (x,_) = x == 2

sucesoresEjarras :: EstadoJarras -> [EstadoJarras]
sucesoresEjarras (x,y) =
    [(4,y) | x < 4] ++
    [(x,3) | y < 3] ++

```

```
[(0,y) | x > 0] ++
[(x,0) | y > 0] ++
[(4,y-(4-x)) | x < 4, y > 0, x + y > 4] ++
[(x-(3-y),3) | x > 0, y < 3, x + y > 3] ++
[(x+y,0) | y > 0, x + y <= 4] ++
[(0,x+y) | x > 0, x + y <= 3]
```

-- *Los nodos son las soluciones parciales*

```
type NodoJarras = [EstadoJarras]
```

```
inicialNjarras :: NodoJarras
```

```
inicialNjarras = [inicialJarras]
```

```
esFinalNjarras :: NodoJarras -> Bool
```

```
esFinalNjarras (e:_) = esFinalJarras e
```

```
sucesoresNjarras :: NodoJarras -> [NodoJarras]
```

```
sucesoresNjarras n@(e:es) =
  [e':n | e' <- sucesoresEjarras e,
        e' 'notElem' n]
```

```
solucionesJarras :: [NodoJarras]
```

```
solucionesJarras = buscaEE sucesoresNjarras
                  esFinalNjarras
                  inicialNjarras
```

```
jarras :: [[(Int,Int)]]
```

```
jarras = map reverse solucionesJarras
```


Relación 35

Rompecabeza del triominó mediante divide y vencerás

```
-- -----  
-- § Introducción --  
-- -----  
  
-- Un poliomínó es una figura geométrica plana formada conectando dos o  
-- más cuadrados por alguno de sus lados. Los cuadrados se conectan lado  
-- con lado, pero no se pueden conectar ni por sus vértices, ni juntando  
-- solo parte de un lado de un cuadrado con parte de un lado de otro. Si  
-- unimos dos cuadrados se obtiene un dominó, si se juntan tres  
-- cuadrados se construye un triominó.  
--  
-- Sólo existen dos triominós, el I-triomino (por tener forma de I) y el  
-- L-triominó (por su forma de L) como se observa en la siguiente figura  
--  
--      X  
--     X   X  
--     X   XX  
--  
-- El rompecabeza del triominó consiste en cubrir un tablero cuadrado  
-- con  $2^n$  filas y  $2^n$  columnas, en el que se ha eliminado una casilla,  
-- con L-triominós de formas que cubran todas las casillas excepto la  
-- eliminada y los triominós no se solapen.  
--  
-- La casilla eliminada se representará con -1 y los L-triominós con  
-- sucesiones de tres números consecutivos en forma de L. Con esta
```

```

-- representación una solución del rompecabeza del triominó con 4 filas
-- y la fila eliminada en la posición (4,4) es
--   ( 3 3 2 2 )
--   ( 3 1 1 2 )
--   ( 4 1 5 5 )
--   ( 4 4 5 -1 )
--
-- En esta relación resolveremos el rompecabeza del triominó mediante
-- divide y vencerás, utilizando las implementaciones estudiadas en el
-- tema 23 que se pueden descargar desde
--   http://www.cs.us.es/~jalonso/cursos/ilm-14/codigos
--
-- Las transparencias del tema 23 se encuentran en
--   http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-23.pdf
--
-- La técnica "divide y vencerás" consta de los siguientes pasos:
-- 1. Dividir el problema en subproblemas menores.
-- 2. Resolver por separado cada uno de los subproblemas; si los
--   subproblemas son complejos, usar la misma técnica recursivamente;
--   si son simples, resolverlos directamente.
-- 3. Combinar todas las soluciones de los subproblemas en una solución
--   simple.
--
-- Con (divideVencerás ind resuelve divide combina pbInicial) se
-- resuelve el problema pbInicial mediante la técnica de divide y
-- vencerás, donde
-- * (ind pb) se verifica si el problema pb es indivisible
-- * (resuelve pb) es la solución del problema indivisible pb
-- * (divide pb) es la lista de subproblemas de pb
-- * (combina pb ss) es la combinación de las soluciones ss de los
--   subproblemas del problema pb.
-- * pbInicial es el problema inicial
--
-- En los distintos apartados de esta relación se irán definiendo las
-- anteriores funciones.
--
-- -----
-- § Librerías auxiliares
-- -----

```

```
import I1M.DivideVencerás
import Data.Matrix
import Data.List (delete)

-- -----
-- § Tipos
-- -----

-- Los tableros son matrices de números enteros donde -1 representa el
-- hueco, 0 las posiciones sin rellenar y los números mayores que 0
-- representan los triominós.

type Tablero = Matrix Int

-- Los problemas se representarán mediante pares formados por un número
-- natural mayor que 0 (que indica el número con el que se formará el
-- siguiente triominó que se coloque) y un tablero.

type Problema = (Int, Tablero)

-- Las posiciones son pares de números enteros

type Posicion = (Int, Int)

-- -----
-- § Problema inicial
-- -----

-- Ejercicio 1. Definir la función
--   tablero :: Int -> Posicion -> Tablero
-- tal que (tablero n p) es el tablero inicial del problema del triominó
-- en un cuadrado nxn en el que se ha eliminado la casilla de la
-- posición (i,j). Por ejemplo,
--   ghci> tablero 4 (3,4)
--   ( 0 0 0 0 )
--   ( 0 0 0 0 )
--   ( 0 0 0 -1 )
--   ( 0 0 0 0 )
-- -----
```

```
tablero :: Int -> Posicion -> Tablero
```

```
tablero n (i,j) =
    setElem (-1) (i,j) (zero n n)
```

```
-----
-- Ejercicio 2. Definir la función
--   pbInicial :: Int -> Posicion -> Problema
-- tal que (pbInicial n p) es el problema inicial del rompecabeza del
-- triominó en un cuadrado nxn en el que se ha eliminado la casilla de
-- la posición p. Por ejemplo,
--   ghci> pbInicial 4 (4,4)
--   (1,( 0 0 0 0 )
--      ( 0 0 0 0 )
--      ( 0 0 0 0 )
--      ( 0 0 0 -1 ))
-----
```

```
pbInicial :: Int -> Posicion -> Problema
```

```
pbInicial n p = (1,tablero n p)
```

```
-----
-- § Problemas indivisibles
-----
```

```
-----
-- Ejercicio 3. Definir la función
--   ind :: Problema -> Bool
-- tal que (ind pb) se verifica si el problema pb es indivisible. Por
-- ejemplo,
--   ind (pbInicial 2 (1,2)) == True
--   ind (pbInicial 4 (1,2)) == False
-----
```

```
ind :: Problema -> Bool
```

```
ind (_,p) = ncols p == 2
```

```
-----
-- § Resolución de problemas indivisibles
-----
```

```

-----
-- Ejercicio 4. Definir la función
--   posicionHueco :: Tablero -> Posicion
-- tal que (posicionHueco t) es la posición del hueco en el tablero
-- t. Por ejemplo,
--   posicionHueco (tablero 8 (5,2)) == (5,2)
-----

```

```

posicionHueco :: Tablero -> Posicion
posicionHueco p =
  head [(i,j) | i <- [1..nrows p],
              j <- [1..ncols p],
              p!(i,j) /= 0]

```

```

-----
-- Ejercicio 5. Definir la función
--   cuadranteHueco :: Tablero -> Int
-- tal que (cuadranteHueco p) es el cuadrante donde se encuentra el
-- hueco del tablero t (donde la numeración de los cuadrantes es 1 el
-- superior izquierdo, 2 el inferior izquierdo, 3 el superior derecho y 4
-- el inferior derecho). Por ejemplo,
--   cuadranteHueco (tablero 8 (4,4)) == 1
--   cuadranteHueco (tablero 8 (5,2)) == 2
--   cuadranteHueco (tablero 8 (3,6)) == 3
--   cuadranteHueco (tablero 8 (6,6)) == 4
-----

```

```

cuadranteHueco :: Tablero -> Int
cuadranteHueco t
  | i <= x && j <= x = 1
  | i >  x && j <= x = 2
  | i <= x && j >  x = 3
  | otherwise       = 4
  where (i,j) = posicionHueco t
        x    = nrows t `div` 2

```

```

-----
-- Ejercicio 6. Definir la función
--   centralHueco :: Tablero -> Posicion

```

```
-- tal que (centralHueco t) es la casilla central del cuadrante del
-- tablero t donde se encuentra el hueco. Por ejemplo,
--   centralHueco (tablero 8 (5,2)) == (5,4)
--   centralHueco (tablero 8 (4,4)) == (4,4)
--   centralHueco (tablero 8 (3,6)) == (4,5)
--   centralHueco (tablero 8 (6,6)) == (5,5)
```

```
centralHueco :: Tablero -> Posicion
centralHueco t = case (cuadranteHueco t) of
    1 -> (x,x)
    2 -> (x+1,x)
    3 -> (x,x+1)
    4 -> (x+1,x+1)
  where x = nrows t `div` 2
```

```
-- -----
-- Ejercicio 7. Definir la función
--   centralesSinHueco :: Tablero -> [Posicion]
-- (centralesSinHueco t) son las posiciones centrales del tablero t de
-- los cuadrantes sin hueco. Por ejemplo,
--   centralesSinHueco (tablero 8 (5,2)) == [(4,4),(4,5),(5,5)]
```

```
centralesSinHueco :: Tablero -> [Posicion]
centralesSinHueco t =
  delete (i,j) [(x,x),(x+1,x),(x,x+1),(x+1,x+1)]
  where x      = nrows t `div` 2
        (i,j) = centralHueco t
```

```
-- -----
-- Ejercicio 8. Definir la función
--   actualiza :: Matrix a -> [((Int,Int),a)] -> Matrix a
-- tal que (actualiza t ps) es la matriz obtenida cambiando en t los
-- valores de las posiciones indicadas en ps por sus correspondientes
-- valores. Por ejemplo,
--   ghci> actualiza (identity 3) [((1,2),4),((3,1),5)]
--   ( 1 4 0 )
--   ( 0 1 0 )
--   ( 5 0 1 )
```

```

-----
actualiza :: Matrix a -> [((Int,Int),a)] -> Matrix a
actualiza p [] = p
actualiza p (((i,j),x):zs) = setElem x (i,j) (actualiza p zs)

```

```

-----
-- Ejercicio 9. Definir la función
--   triominoCentral :: Problema -> Tablero
-- tal que (triominoCentral (n,t) es el tablero obtenido colocando el
-- triominó formado por el número n en las posiciones centrales de los 3
-- cuadrantes que no contienen el hueco. Por ejemplo,
--   ghci> triominoCentral (7,tablero 4 (4,4))
--   ( 0 0 0 0 )
--   ( 0 7 7 0 )
--   ( 0 7 0 0 )
--   ( 0 0 0 -1 )
-----

```

```

triominoCentral :: Problema -> Tablero
triominoCentral (n,t) =
  actualiza t [((i,j),n) | (i,j) <- centralesSinHueco t]

```

```

-----
-- Ejercicio 10. Definir la función
--   resuelve :: Problema -> Tablero
-- tal que (resuelve p) es la solución del problema indivisible p. Por
-- ejemplo,
--   ghci> tablero 2 (2,2)
--   ( 0 0 )
--   ( 0 -1 )
--
--   ghci> resuelve (5,tablero 2 (2,2))
--   ( 5 5 )
--   ( 5 -1 )
-----

```

```

resuelve :: Problema -> Tablero
resuelve = triominoCentral

```

```

-----
-- § División en subproblemas                                     --
-----

-- Ejercicio 11. Definir la función
--   divide :: Problema -> [Problema]
-- tal que (divide (n,t)) es la lista de de los problemas obtenidos
-- colocando el triominó n en las casillas centrales de t que no
-- contienen el hueco y dividir el tablero en sus cuatros cuadrantes y
-- aumentar en uno el número del correspondiente triominó. Por ejemplo,
--   ghci> divide (3,tablero 4 (4,4))
--   [(4,( 0 0 )
--      ( 3 0 )),
--    (5,( 0 0 )
--      ( 0 3 )),
--    (6,( 0 3 )
--      ( 0 0 )),
--    (7,( 0 0 )
--      ( 0 -1 ))]
-----

```

```

divide :: Problema -> [Problema]
divide (n,t) =
  [(n+1, submatrix 1      x (x+1) m q),
   (n+2, submatrix 1      x 1      x q),
   (n+3, submatrix (x+1) m 1      x q),
   (n+4, submatrix (x+1) m (x+1) m q)]
  where q = triominoCentral (n,t)
        m = nrows t
        x = m `div` 2

```

```

-----
-- § Combinación de soluciones                                     --
-----

-- Ejercicio 12. Definir la función
--   combina :: Problema -> [Tablero] -> Tablero
-- tal que (combina p ts) es la combinación de las soluciones ts de los

```



```
-- subproblemas del problema p. Por ejemplo,
-- ghci> let inicial = (1,tablero 4 (4,4)) :: (Int,Matrix Int)
-- ghci> let [p1,p2,p3,p4] = divide inicial
-- ghci> let [s1,s2,s3,s4] = map resuelve [p1,p2,p3,p4]
-- ghci> combina 1 [s1,s2,s3,s4]
-- ( 3 3 2 2 )
-- ( 3 1 1 2 )
-- ( 4 1 5 5 )
-- ( 4 4 5 -1 )
```

```
-----
combina :: Problema -> [Tablero] -> Tablero
combina _ [s1,s2,s3,s4] = joinBlocks (s2,s1,s3,s4)
```

```
-----
-- § Solución mediante divide y vencerás
-----
```

```
-----
-- Ejercicio 13. Definir la función
-- triomino :: Int -> Posicion -> Tablero
-- tal que (triomino n p) es la solución, mediante divide y vencerás,
-- del rompecabeza del triominó en un cuadrado nxn en el que se ha
-- eliminado la casilla de la posición p. Por ejemplo,
-- ghci> triomino 4 (4,4)
-- ( 3 3 2 2 )
-- ( 3 1 1 2 )
-- ( 4 1 5 5 )
-- ( 4 4 5 -1 )
--
-- ghci> triomino 4 (2,3)
-- ( 3 3 2 2 )
-- ( 3 1 -1 2 )
-- ( 4 1 1 5 )
-- ( 4 4 5 5 )
--
-- ghci> triomino 16 (5,6)
-- ( 7 7 6 6 6 6 5 5 6 6 5 5 5 5 4 4 )
-- ( 7 5 5 6 6 4 4 5 6 4 4 5 5 3 3 4 )
-- ( 8 5 9 9 7 7 4 8 7 4 8 8 6 6 3 7 )
```

```
-- ( 8 8 9 3 3 7 8 8 7 7 8 2 2 6 7 7 )
-- ( 8 8 7 3 9 -1 8 8 7 7 6 6 2 8 7 7 )
-- ( 8 6 7 7 9 9 7 8 7 5 5 6 8 8 6 7 )
-- ( 9 6 6 10 10 7 7 11 8 8 5 9 9 6 6 10 )
-- ( 9 9 10 10 10 10 11 11 1 8 9 9 9 9 10 10 )
-- ( 8 8 7 7 7 7 6 1 1 9 8 8 8 8 7 7 )
-- ( 8 6 6 7 7 5 6 6 9 9 7 8 8 6 6 7 )
-- ( 9 6 10 10 8 5 5 9 10 7 7 11 9 9 6 10 )
-- ( 9 9 10 4 8 8 9 9 10 10 11 11 5 9 10 10 )
-- ( 9 9 8 4 4 10 9 9 10 10 9 5 5 11 10 10 )
-- ( 9 7 8 8 10 10 8 9 10 8 9 9 11 11 9 10 )
-- ( 10 7 7 11 11 8 8 12 11 8 8 12 12 9 9 13 )
-- ( 10 10 11 11 11 11 12 12 11 11 12 12 12 12 13 13 )
```

```
triomino :: Int -> Posicion -> Tablero
```

```
triomino n p =
```

```
    divideVenceras ind resuelve divide combina (pbInicial n p)
```

```
-- -----
-- § Referencias
```

```
-- + Raúl Ibáñez "Embaldosando con L-triominós (Un ejemplo de
--   demostración por inducción)" http://bit.ly/1DKPBbt
-- + "Algorithmic puzzles" pp. 10.
```

```
-- Programas interactivos
```

```
-- =====
```

```
-- + "Interactive 8-by-8 Tromino Puzzle" http://bit.ly/1DKRNjn
-- + "Tromino Puzzle: Interactive Illustration of Golomb's Theorem"
--   http://bit.ly/1DKS0mL
```

Relación 36

Analizadores sintácticos

-- § Introducción --

-- En esta relación construiremos analizadores sintácticos, utilizando
-- las implementaciones estudiadas en el tema 12, cuyas transparencias
-- se encuentran en

-- <http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-12.pdf>
--

-- Usaremos la librería I1M.Analizador que se encuentra en

-- <http://www.cs.us.es/~jalonso/cursos/ilm/codigos/I1M2014.zip>
--

-- § Librerías auxiliares --

import I1M.Analizador

-- Ejercicio 1. Un número entero es un signo menos seguido por un número
-- natural o un número natural. Definir el analizador

-- `int :: Analizador Int`

-- para reconocer los números enteros. Por ejemplo,

-- `analiza int "14DeAbril" ==> [(14,"DeAbril")]`

-- `analiza int "-14DeAbril" ==> [(-14,"DeAbril")]`

```

int :: Analizador Int
int = (caracter '-' >*> \_ ->
      nat          >*> \n ->
      resultado (-n))
+++ nat

```

```

-----
-- Ejercicio 2. Definir el analizador
-- comentario :: Analizador ()
-- para reconocer los comentarios simples de Haskell que comienzan con
-- el símbolo -- y terminan al final de la línea, que se representa por
-- el carácter de control '\n'. Por ejemplo,
-- ghci> analiza comentario "-- 14DeAbril\nSiguiente"
-- [((), "Siguiente")]
-- ghci> analiza comentario "- 14DeAbril\nSiguiente"
-- []
-----

```

```

comentario :: Analizador ()
comentario = cadena "--"          >*> \_ ->
            varios (sat (/= '\n')) >*> \_ ->
            elemento              >*> \_ ->
            resultado ()

```

```

-----
-- Ejercicio 3. Extender el analizador de expresiones aritméticas para
-- incluir restas y divisiones basándose en la siguiente extensión de
-- la gramática:
-- expr1 ::= term1 (+ expr1 | - expr1 | vacía)
-- term1 ::= factor1 (* term1 | / term1 | vacía)
-- Por ejemplo,
-- analiza expr1 "2*3+5"    => [(11, "")]
-- analiza expr1 "2*(3+5)" => [(16, "")]
-- analiza expr1 "2+3*5"   => [(17, "")]
-- analiza expr1 "2*3+5abc" => [(11, "abc")]
-- analiza expr1 "24/4-2"   => [(4, "")]
-- analiza expr1 "24/(4-2)" => [(12, "")]
-- analiza expr1 "24-(4/2)" => [(22, "")]
-- analiza expr1 "24/4-2abc" => [(4, "abc")]
-----

```

expr1 :: Analizador Int

```

expr1 = term1          >*> \t ->
      (simbolo "+"     >*> \_ ->
        expr1          >*> \e ->
        resultado (t+e))
+++ (simbolo "-"      >*> \_ ->
      expr1            >*> \e ->
      resultado (t-e))
+++ resultado t

```

-- *term1* analiza un término de una expresión aritmética devolviendo su
 -- valor. Por ejemplo,

```

-- analiza term1 "2*3+5"      => [(6,"+5")]
-- analiza term1 "2+3*5"      => [(2,"+3*5")]
-- analiza term1 "(2+3)*5+7"  => [(25,"+7")]
-- analiza term1 "2*3-6/3"    => [(6,"-6/3")]
-- analiza term1 "24/4-2"     => [(6,"-2")]
-- analiza term1 "24-4/2"     => [(24,"-4/2")]
-- analiza term1 "(24-4)/2+7" => [(10,"+7")]
-- analiza term1 "24/4-2^3"   => [(6,"-2^3")]

```

term1 :: Analizador Int

```

term1 = factor1       >*> \f ->
      (simbolo "*"     >*> \_ ->
        term1          >*> \t ->
        resultado (f*t))
+++ (simbolo "/"      >*> \_ ->
      term1            >*> \t ->
      resultado (f 'div' t))
+++ resultado f

```

-- *factor1* analiza un factor de una expresión aritmética devolviendo su
 -- valor. Por ejemplo,

```

-- analiza factor1 "2*3+5"    => [(2,"*3+5")]
-- analiza factor1 "(2+3)*5"  => [(5,"*5")]
-- analiza factor1 "(2+3*7)*5" => [(23,"*5")]
-- analiza factor1 "24/4-2"   => [(24,"/4-2")]
-- analiza factor1 "(24-4)/2" => [(20,"/2")]
-- analiza factor1 "(24-4*2)/2" => [(16,"/2")]

```

factor1 :: Analizador Int

```
factor1 = (simbolo "(" >*> \_ ->
          expr1      >*> \e ->
          simbolo ")" >*> \_ ->
          resultado e)
+++ natural
```

```
-----
-- Ejercicio 4. Extender el analizador de expresiones aritméticas para
-- incluir exponenciación, que asocie por la derecha y tenga mayor
-- prioridad que la multiplicación y la división, pero menor que los
-- paréntesis y los números. Por ejemplo,
-- analiza expr2 "2^3*4" => [(32,"")]
-- Indicación: El nuevo nivel de prioridad requiere una nueva regla en
-- la gramática.
-----
```

```
-- Las nuevas reglas son
-- factor2 ::= atomo (^ factor2 | epsilon)
-- atomo   ::= (expr) | nat
```

```
-- Las definiciones correspondientes son
```

```
-- expr2 analiza una expresión aritmética devolviendo su valor. Por
-- ejemplo,
-- analiza expr2 "2*3+5"      => [(11,"")]
-- analiza expr2 "2*(3+5)"    => [(16,"")]
-- analiza expr2 "2+3*5"      => [(17,"")]
-- analiza expr2 "2*3+5abc"   => [(11,"abc")]
-- analiza expr2 "24/4-2"     => [(4,"")]
-- analiza expr2 "24/(4-2)"   => [(12,"")]
-- analiza expr2 "24-(4/2)"   => [(22,"")]
-- analiza expr2 "24/4-2abc"  => [(4,"abc")]
-- analiza expr2 "2^3*4"      => [(32,"")]
```

```
expr2 :: Analizador Int
```

```
expr2 = term2 >*> \t ->
      (simbolo "+" >*> \_ ->
       expr2      >*> \e ->
       resultado (t+e))
+++ (simbolo "-" >*> \_ ->
     expr2      >*> \e ->
```

```

        resultado (t-e))
    +++ resultado t

-- term2 analiza un término de una expresión aritmética devolviendo su
-- valor. Por ejemplo,
-- analiza term2 "2*3+5"      => [(6,"+5")]
-- analiza term2 "2+3*5"      => [(2,"+3*5")]
-- analiza term2 "(2+3)*5+7"  => [(25,"+7")]
-- analiza term2 "2*3-6/3"    => [(6,"-6/3")]
-- analiza term2 "24/4-2"     => [(6,"-2")]
-- analiza term2 "24-4/2"     => [(24,"-4/2")]
-- analiza term2 "(24-4)/2+7" => [(10,"+7")]
-- analiza term2 "24/4-2^3"   => [(6,"-2^3")]
-- analiza term2 "2^3*4"      => [(32,"")]
term2 :: Analizador Int
term2 = factor2 >*> \f ->
    (simbolo "*" >*> \_ ->
      term2 >*> \t ->
        resultado (f*t))
  +++ (simbolo "/" >*> \_ ->
    term2 >*> \t ->
      resultado (f 'div' t))
  +++ resultado f

-- factor2 analiza un factor de una expresión aritmética devolviendo su
-- valor. Por ejemplo,
-- analiza factor2 "2*3+5"      => [(2,"*3+5")]
-- analiza factor2 "(2+3)*5"    => [(5,"*5")]
-- analiza factor2 "(2+3*7)*5"  => [(23,"*5")]
-- analiza factor2 "24/4-2"     => [(24,"/4-2")]
-- analiza factor2 "(24-4)/2"   => [(20,"/2")]
-- analiza factor2 "(24-4*2)/2" => [(16,"/2")]
-- analiza factor2 "2^3*4"      => [(8,"*4")]
factor2 :: Analizador Int
factor2 = (atomo >*> \a ->
    (simbolo "^" >*> \_ ->
      factor >*> \f ->
        resultado (a ^ f))
  +++ resultado a)

```

```

-- atomo analiza un átomo de una expresión aritmética devolviendo su
-- valor. Por ejemplo,
--   analiza atomo "2^3*4"   => [(2,"^3*4")]
--   analiza atomo "(2^3)*4" => [(8,"*4")]
atomo :: Analizador Int
atomo = (simbolo "(" >*> \_ ->
        expr2 >*> \e ->
        simbolo ")") >*> \_ ->
        resultado e)
+++ natural

```

```

-----
-- Ejercicio 5.1. Definir el analizador
--   expr3 :: Analizador Arbol
-- tal que (analiza expr3 c) es el árbol e la expresión correspondiente
-- a la cadena c. Por ejemplo,
--   ghci> analiza expr3 "2*3+5"
--   [(N '+' (N '*' (H 2) (H 3)) (H 5)), ""]
--   ghci> analiza expr3 "2*(3+5)"
--   [(N '*' (H 2) (N '+' (H 3) (H 5))), ""]
--   ghci> analiza expr3 "2+3*5"
--   [(N '+' (H 2) (N '*' (H 3) (H 5))), ""]
--   ghci> analiza expr3 "2*3+5abc"
--   [(N '+' (N '*' (H 2) (H 3)) (H 5)), "abc"]
-----

```

```

data Arbol = H Int | N Char Arbol Arbol
deriving Show

```

```

expr3 :: Analizador Arbol
expr3 = term3 >*> \t ->
        (simbolo "+" >*> \_ ->
         expr3 >*> \e ->
         resultado (N '+' t e))
+++ resultado t

```

```

-- analiza term3 "2*3+5" => [(N '*' (H 2) (H 3)), "+5"]
term3 :: Analizador Arbol
term3 = factor3 >*> \f ->
        (simbolo "*" >*> \_ ->

```



```

term3      >*> \t ->
resultado (N '*' f t)
+++ resultado f

-- analiza factor3 "2*3+5" => [(H 2,"*3+5")]
factor3 :: Analizador Arbol
factor3 = (simbolo "(" >*> \_ ->
          expr3      >*> \e ->
          simbolo ")" >*> \_ ->
          resultado e)
+++ natural'

-- analiza nat3 "14DeAbril" => [(H 14,"DeAbril")]
-- analiza nat3 " 14DeAbril" => []
nat3 :: Analizador Arbol
nat3 = varios1 digito >*> \xs ->
      resultado (H (read xs))

-- analiza natural' " 14DeAbril" => [(H 14,"DeAbril")]
natural' :: Analizador Arbol
natural' = unidad nat3

-----
-- Ejercicio 5.2. Definir la función
-- arbolAnalisis :: String -> Arbol
-- tal que (arbolAnalisis c) es el árbol de análisis correspondiente a
-- la cadena c, si c representa a una expresión aritmética y error en
-- caso contrario. Por ejemplo,
-- ghci> arbolAnalisis "2*3+5"
-- N '+' (N '*' (H 2) (H 3)) (H 5)
-- ghci> arbolAnalisis "2*(3+5)"
-- N '*' (H 2) (N '+' (H 3) (H 5))
-- ghci> arbolAnalisis "2 * 3 + 5"
-- N '+' (N '*' (H 2) (H 3)) (H 5)
-- ghci> arbolAnalisis "2*3x+5y"
-- *** Exception: entrada sin usar x+5y
-- ghci> arbolAnalisis "-1"
-- *** Exception: entrada no valida
-----

```

```

arbolAnálisis :: String -> Arbol
arbolAnálisis xs = case (analiza expr3 xs) of
    [(t,[])] -> t
    [(_,sal)] -> error ("entrada sin usar " ++ sal)
    [] -> error "entrada no valida"

```

```

-----
-- Ejercicio 6. Definir la función
-- listaNV :: Analizador a -> Analizador [a]
-- tal que (listaNV p) es un analizador de listas no vacías de elementos
-- reconocibles por el analizador p. Por ejemplo,
-- ghci> analiza (listaNV natural) "[3, 5,4]"
-- [[3,5,4],"" ]
-- ghci> analiza (listaNV natural) "[3, 5,4.0]"
-- []
-- ghci> analiza (listaNV identificador) "[hoy , es,lunes ]"
-- [(["hoy","es","lunes"],"")]
-- ghci> analiza (listaNV identificador) "[hoy , es,lunes,18 ]"
-- []
-----

```

```

listaNV :: Analizador a -> Analizador [a]
listaNV p = simbolo "["          >*> \_ ->
    p                            >*> \x ->
    varios (simbolo ","          >*> \_ ->
            p)                  >*> \xs ->
    simbolo "]"                 >*> \_ ->
    resultado (x:xs)

```

```

-----
-- Ejercicio 7.1. Definir el analizador
-- exprPBA :: Analizador ()
-- para reconocer cadenas de paréntesis bien anidados. Por ejemplo,
-- analiza exprPBA "((()))" == [(()),""]
-- analiza exprPBA "((()))(" == [(()),""]
-----

```

```

-- La gramática es
-- exprPBA := '(' exprPBA ')' exprPBA | vacía

```

```

exprPBA :: Analizador ()
exprPBA = (simbolo "(" >*> \i ->
           exprPBA >*> \xs ->
           simbolo ")" >*> \f ->
           exprPBA >*> \ys ->
           resultado ())
+++
(simbolo "" >*> \_ ->
 resultado ())

```

```

-----
-- Ejercicio 7.2. Definir el analizador
--   exprPBA :: Analizador ()
-- para reconocer simbolos de paréntesis bien anidados con cálculo de la
-- mayor profundidad de anidamiento. Por ejemplo,
--   analiza exprPBA2 ""           == [(0,"")]
--   analiza exprPBA2 "()"         == [(1,"")]
--   analiza exprPBA2 "()()"       == [(1,"")]
--   analiza exprPBA2 "(()())"     == [(2,"")]
--   analiza exprPBA2 "((())())"   == [(3,"")]
--   analiza exprPBA2 "())()"      == [(1,"")("")]
-----

```

```

exprPBA2 :: Analizador Int
exprPBA2 = (simbolo "(" >*> \_ ->
           exprPBA2 >*> \n ->
           simbolo ")" >*> \_ ->
           exprPBA2 >*> \m ->
           resultado (max (n+1) m))
+++
(simbolo "" >*> \_ ->
 resultado 0)

```


Relación 37

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/ilm-14/temas/tema-13.pdf>*

--

-- *Nota: El juego debe de ejecutarse en una consola, no en la shell de
-- emacs.*

-- *§ Librerías auxiliares* -----

```

import Data.Char

-----
-- § 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 t = all (== 0) t

-----

-- Ejercicio 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
```

```
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..5] 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 ()
-- 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 = do putStr (show f)
                    putStr ": "
                    putStrLn (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 [a,b,c,d,e] = do escribeFila 1 a
                                escribeFila 2 b
                                escribeFila 3 c
                                escribeFila 4 d
                                escribeFila 5 e
```

```
-----
-- Ejercicio 8. Definir la acción
-- leeDigito :: String -> IO Int
-- tal que (leeDigito c) escribe una nueva línea con l cadena "prueba",
-- lee un carácter y comprueba que es un dígito. Además, si el carácter
-- leído 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
```

```
-----
-- 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 "
                         putStrLn (show 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 38

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
```

```
-----
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 (-1.0) 1.0) (aleatorios (-1.0) 1.0)
```

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


Relación 39

Demostración de propiedades de programas por inducción sobre números y listas

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se plantean ejercicios de demostración por inducción  
-- de propiedades de programas. La inducción se realiza sobre números  
-- naturales y sobre listas.  
--  
-- Las transparencias del tema correspondiente se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-8.pdf  
-- -----  
-- Importación de librerías --  
-- -----  
  
import Data.List  
import Test.QuickCheck  
  
-- -----  
-- Ejercicio 1.1. Definir, por recursión, la función  
-- sumaImpares :: Int -> Int  
-- tal que (sumaImpares n) es la suma de los n primeros números  
-- impares. Por ejemplo,
```

```
-- sumaImpares 5 == 25
```

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

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

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

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

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

```
-- La comprobación es
-- ghci> sumaImparesIguales 1 100
-- True
```

```
-----
-- Ejercicio 1.4. Definir la función
-- grafoSumaImpares :: Int -> Int -> [(Int,Int)]
```

```

-- tal que (grafoSumaImpares m n) es la lista formadas por los números x
-- entre m y n y los valores de (sumaImpares x).
--
-- Calcular (grafoSumaImpares 1 9).
-----

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

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

-- Ejercicio 1e. Demostrar por inducción que para todo n,
-- (sumaImpares n) es igual a n^2.
-----

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

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

```

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

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

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

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

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

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

Caso inductivo: Se supone la hipótesis de inducción (H.I.)

```
  sumaPotenciasDeDosMasUno n = 2^(n+1)
```

Hay que demostrar que

```

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

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

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

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

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

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

-- La propiedad es
prop_copia :: Eq a => Int -> a -> Bool

```

```

prop_copia n x =
  todos (==x) (copia n' x)
  where n' = abs n

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

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

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

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

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

-----
-- Ejercicio 3.5. Definir, por plegado, la función
-- todos' :: (a -> Bool) -> [a] -> Bool

```



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

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

```
-----
-- Ejercicio 5.1. Definir, por recursión, la función
--   factR :: Integer -> Integer
-- tal que (factR n) es el factorial de n. Por ejemplo,
--   factR 4 == 24
```

```
-----
factR :: Integer -> Integer
factR 0 = 1
factR n = n * factR (n-1)
```

```
-----
-- Ejercicio 5.2. Definir, por comprensión, la función
--   factC :: Integer -> Integer
-- tal que (factC n) es el factorial de n. Por ejemplo,
--   factC 4 == 24
```

```
-----
factC :: Integer -> Integer
factC n = product [1..n]
```

```
-----
-- Ejercicio 1.3. Comprobar con QuickCheck que las funciones factR y
-- factC son equivalentes sobre los números naturales.
```

```
-----
-- La propiedad es
prop_factR_factC :: Integer -> Bool
prop_factR_factC n =
    factR n' == factC n'
  where n' = abs n
```

```

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

-----

-- Ejercicio 5.4. Comprobar con QuickCheck si las funciones factR y
-- factC son equivalentes sobre los números enteros.
-----

-- La propiedad es
prop_factR_factC_Int :: Integer -> Bool
prop_factR_factC_Int n =
    factR n == factC n

-- La comprobación es
-- ghci> quickCheck prop_factR_factC_Int
-- *** Exception: Non-exhaustive patterns in function factR

-- No son iguales ya que factR no está definida para los números
-- negativos y factC de cualquier número negativo es 0.

-----

-- Ejercicio 5.5. Se considera la siguiente definición iterativa de la
-- función factorial
-- factI :: Integer -> Integer
-- factI n = factI' n 1
--
-- factI' :: Integer -> Integer -> Integer
-- factI' 0 x = x -- factI'.1
-- factI' n x = factI' (n-1) n*x -- factI'.2
-- Comprobar con QuickCheck que factI y factR son equivalentes sobre los
-- números naturales.
-----

factI :: Integer -> Integer
factI n = factI' n 1

factI' :: Integer -> Integer -> Integer
factI' 0 x = x

```

```
factI' n x = factI' (n-1) n*x
```

```
-- La propiedad es
```

```
prop_factI_factR n =
  factI n' == factR n'
  where n' = abs n
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_factI_factR
-- OK, passed 100 tests.
```

```
-----
-- Ejercicio 5.6. Comprobar con QuickCheck que para todo número natural
-- n, (factI' n x) es igual al producto de x y (factR n).
-----
```

```
-- La propiedad es
```

```
prop_factI' :: Integer -> Integer -> Bool
prop_factI' n x =
  factI' n' x == x * factR n'
  where n' = abs n
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_factI'
-- OK, passed 100 tests.
```

```
-----
-- Ejercicio 5.7. Demostrar por inducción que para todo número natural
-- n, (factI' n x) es igual x*n!
-----
```

```
{-
```

```
  Demostración (por inducción en n)
```

```
  Caso base: Hay que demostrar que factI' 0 x = x*0!
```

```
  En efecto,
```

```
    factI' 0 x
  = x           [por factI'.1]
  = x*0!       [por álgebra]
```

Caso inductivo: Se supone la hipótesis de inducción: para todo x ,
 *$factI' n x = x*n!$*

hay que demostrar que para todo x

$factI' (n+1) x = x(n+1)!$*

En efecto,

$factI' (n+1) x$

*$= factI' n (n+1)*x$ [por $factI'.2$]*

*$= (n+1)*x*n!$ [por hipótesis de inducción]*

$= x(n+1)!$ [por álgebra]*

-}

 -- *Ejercicio 6.1. Definir, recursivamente y sin usar (++)*. la función
 -- *$amplia :: [a] -> a -> [a]$*
 -- *tal que ($amplia xs y$) es la lista obtenida añadiendo el elemento y al*
 -- *final de la lista xs . Por ejemplo,*
 -- *$amplia [2,5] 3 == [2,5,3]$*

```
amplia :: [a] -> a -> [a]
amplia [] y = [y]           -- amplia.1
amplia (x:xs) y = x : amplia xs y -- amplia.2
```

 -- *Ejercicio 6.2. Definir, mediante plegado*. la función
 -- *$ampliaF :: [a] -> a -> [a]$*
 -- *tal que ($ampliaF xs y$) es la lista obtenida añadiendo el elemento y al*
 -- *final de la lista xs . Por ejemplo,*
 -- *$ampliaF [2,5] 3 == [2,5,3]$*

```
ampliaF :: [a] -> a -> [a]
ampliaF xs y = foldr (:) [y] xs
```

 -- *Ejercicio 6.3. Comprobar con QuickCheck que $amplia$ y $ampliaF$ son*
 -- *equivalentes.*

-- *La propiedad es*

```
prop_amplia_ampliaF :: Eq a => [a] -> a -> Bool
prop_amplia_ampliaF xs y =
    amplia xs y == ampliaF xs y
```

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

```
-----
-- Ejercicio 6.4. Comprobar con QuickCheck que
-- amplia xs y = xs ++ [y]
-----
```

```
-- La propiedad es
prop_amplia :: Eq a => [a] -> a -> Bool
prop_amplia xs y =
    amplia xs y == xs ++ [y]
```

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

```
-----
-- Ejercicio 6.5. Demostrar por inducción que
-- amplia xs y = xs ++ [y]
-----
```

```
{-
  Demostración: Por inducción en xs.
```

```
  Caso base: Hay que demostrar que
    amplia [] y = [] ++ [y]
  En efecto,
    amplia [] y
    = [y]           [por amplia.1]
    = [] ++ [y]    [por (++).1]
```

```
  Caso inductivo: Se supone la hipótesis de inducción
    amplia xs y = xs ++ [y]
  Hay que demostrar que
```

```

    amplia (x:xs) y = (x:xs) ++ [y]
En efecto,
    amplia (x:xs) y
= x : amplia xs y    [por amplia.2]
= x : (xs ++ [y])    [por hipótesis de inducción]
= (x:xs) ++ [y]      [por (++).2]
-}

-----
-- Ejercicio 7.1. Definir la función
-- listaConSuma :: Int -> [[Int]]
-- que, dado un número natural n, devuelve todas las listas de enteros
-- positivos (esto es, enteros mayores o iguales que 1) cuya suma sea
-- n. Por ejemplo,
-- Main> listaConSuma 4
-- [[1,1,1,1],[1,1,2],[1,2,1],[1,3],[2,1,1],[2,2],[3,1],[4]]
-----

listaConSuma :: Int -> [[Int]]
listaConSuma 0 = [[]]
listaConSuma n = [x:xs | x <- [1..n], xs <- listaConSuma (n-x)]

-----
-- Ejercicio 7.2. Definir la función
-- numeroDeListasConSuma :: Int -> Int
-- tal que (numeroDeListasConSuma n) es el número de elementos de
-- (listaConSuma n). Por ejemplo,
-- numeroDeListasConSuma 10 = 512
-----

numeroDeListasConSuma :: Int -> Int
numeroDeListasConSuma = length . listaConSuma

-----
-- Ejercicio 7.3. Definir la constante
-- numerosDeListasConSuma :: [(Int,Int)]
-- tal que numerosDeListasConSuma es la lista de los pares formado por un
-- número natural n mayor que 0 y el número de elementos de
-- (listaConSuma n).
--

```

```

-- Calcular el valor de
--   take 10 numerosDeListasConSuma
-----

-- La constante es
numerosDeListasConSuma :: [(Int,Int)]
numerosDeListasConSuma = [(n,numeroDeListasConSuma n) | n <- [1..]]

-- El cálculo es
--   ghci> take 10 numerosDeListasConSuma
--   [(1,1),(2,2),(3,4),(4,8),(5,16),(6,32),(7,64),(8,128),(9,256),(10,512)]
-----

-- Ejercicio 7.4. A partir del ejercicio anterior, encontrar una fórmula
-- para calcular el valor de (numeroDeListasConSuma n) para los
-- números n mayores que 0.
--
-- Demostrar dicha fórmula por inducción fuerte.
-----

{-
La fórmula es
  numeroDeListasConSuma n = 2^(n-1)
La demostración, por inducción fuerte en n, es la siguiente:

Caso base (n=1):
  numeroDeListasConSuma 1
= length (listaConSuma 1)
  [por numeroDeListasConSuma]
= length [[x:xs | x <- [1..1], xs <- listaConSuma [[]]]
  [por listaConSuma.2]
= length [[1]]
  [por def. de listas de comprensión]
= 1
  [por def. de length]
= 2^(1-1)
  [por aritmética]

Paso de inducción: Se supone que
  para todo x en [1..n-1],

```

```

    numeroDeListasConSuma x = 2^(x-1)
Hay que demostrar que
    numeroDeListasConSuma n = 2^(n-1)
En efecto,
    numeroDeListasConSuma n
= length (listaConSuma n)
  [por numeroDeListasConSuma]
= length [x:xs | x <- [1..n], xs <- listaConSuma (n-x)]
  [por listaConSuma.2]
= sum [numeroDeListasConSuma (n-x) | x <- [1..n]]
  [por length y listas de comprensión]
= sum [2^(n-x-1) | x <- [1..n-1]] + 1
  [por hip. de inducción y numeroDeListasConSuma]
= 2^(n-2) + 2^(n-3) + ... + 2^1 + 2^0 + 1
= 2^(n-1)
  [por el ejercicio 2c de la relación 15]
-}

-----
-- Ejercicio 7.5. A partir del ejercicio anterior, definir de manera más
-- eficiente la función numeroDeListasConSuma.
-----

numeroDeListasConSuma' :: Int -> Int
numeroDeListasConSuma' 0 = 1
numeroDeListasConSuma' n = 2^(n-1)

-----
-- Ejercicio 7.6. Comparar la eficiencia de las dos definiciones
-- comparando el tiempo y el espacio usado para calcular
-- (numeroDeListasConSuma 20) y (numeroDeListasConSuma' 20).
-----

-- La comparación es
-- ghci> :set +s
-- ghci> numeroDeListasConSuma 20
-- 524288
-- (9.99 secs, 519419824 bytes)
-- ghci> numeroDeListasConSuma' 20
-- 524288

```



```

--      (0.01 secs, 0 bytes)

-----
-- Ejercicio 8.0. La sucesión de Fibonacci
--      0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
-- puede definirse por recursión como
--      fib :: Int -> Int
--      fib 0      = 0                -- fib.1
--      fib 1      = 1                -- fib.2
--      fib (n+2) = (fib (n+1)) + fib n -- fib.3
-- También puede definirse por recursión iterativa como
--      fibIt :: Int -> Int
--      fibIt n = fibItAux n 0 1
-- donde la función auxiliar se define por
--      fibItAux :: Int -> Int -> Int -> Int
--      fibItAux 0      a b = a                -- fibItAux.1
--      fibItAux (n+1) a b = fibItAux n b (a+b) -- fibItAux.2
-----

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

fibIt :: Int -> Int
fibIt n = fibItAux n 0 1

fibItAux :: Int -> Int -> Int -> Int
fibItAux 0 a b = a
fibItAux n a b = fibItAux (n-1) b (a+b)

-----
-- Ejercicio 8.1. Comprobar con QuickCheck que para todo número natural
-- n tal que n <= 20, se tiene que
--      fib n = fibIt n
-----

-- La propiedad es
prop_fib :: Int -> Property
prop_fib n =

```

```

n >= 0 && n <= 20 ==> fib n == fibIt n

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

-----

-- Ejercicio 8.2. Sea  $f$  la función definida por
--  $f :: Int -> Int -> Int$ 
--  $f n k = fibItAux n (fib k) (fib (k+1))$ 
-- Definir la función
--  $grafoDeF :: Int -> [(Int,Int)]$ 
-- tal que  $(grafoDeF n)$  es la lista de los pares formados por un número
-- natural  $k$  y el valor de  $(f n k)$ , para  $k \geq 1$ . Por ejemplo,
-- ghci> take 7 (grafoDeF 3)
-- [(1,3), (2,5), (3,8), (4,13), (5,21), (6,34), (7,55)]
-- ghci> take 7 (grafoDeF 5)
-- [(1,8), (2,13), (3,21), (4,34), (5,55), (6,89), (7,144)]

-----

f :: Int -> Int -> Int
f n k = fibItAux n (fib k) (fib (k+1))

grafoDeF :: Int -> [(Int,Int)]
grafoDeF n = [(k, f n k) | k <- [1..]]

-----

-- Ejercicio 8.3. Comprobar con QuickCheck que para todo par de números
-- naturales  $n, k$  tales que  $n+k \leq 20$ , se tiene que
--  $fibItAux n (fib k) (fib (k+1)) = fib (k+n)$ 

-----

-- La propiedad es
prop_fibItAux :: Int -> Int -> Property
prop_fibItAux n k =
  n >= 0 && k >= 0 && n+k <= 20 ==>
  fibItAux n (fib k) (fib (k+1)) == fib (k+n)

-- La comprobación es
-- ghci> quickCheck prop_fibItAux

```

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

```
-----
-- Ejercicio 8.4. Demostrar por inducción que para todo n y todo k,
--      fibItAux n (fib k) (fib (k+1)) = fib (k+n)
-----
```

```
{-
  Demostración: Por inducción en n se prueba que
    para todo k, fibItAux n (fib k) (fib (k+1)) = fib (k+n)
```

```
  Caso base (n=0): Hay que demostrar que
    para todo k, fibItAux 0 (fib k) (fib (k+1)) = fib k
  En efecto, sea k un número natural. Se tiene
    fibItAux 0 (fib k) (fib (k+1))
    = fib k                                [por fibItAux.1]
```

```
  Paso de inducción: Se supone la hipótesis de inducción
    para todo k, fibItAux n (fib k) (fib (k+1)) = fib (k+n)
  Hay que demostrar que
    para todo k, fibItAux (n+1) (fib k) (fib (k+1)) = fib (k+n+1)
  En efecto. Sea k un número natural,
    fibItAux (n+1) (fib k) (fib (k+1))
    = fibItAux n (fib (k+1)) ((fib k) + (fib (k+1)))
      [por fibItAux.2]
    = fibItAux n (fib (k+1)) (fib (k+2))
      [por fib.3]
    = fib (n+k+1)
      [por hipótesis de inducción]
```

```
-}
```

```
-----
-- Ejercicio 8.5. Demostrar que para todo n,
--      fibIt n = fib n
-----
```

```
{-
  Demostración
    fibIt n
    = fibItAux n 0 1                        [por fibIt]
```

```

    = fibItAux n (fib 0) (fib 1)  [por fib.1 y fib.2]
    = fib n                       [por ejercicio 8.4]
-}

```

```

-----
-- Ejercicio 8.1. La función potencia puede definirse por
-- potencia :: Int -> Int -> Int
-- potencia x 0 = 1
-- potencia x n | even n    = potencia (x*x) (div n 2)
--               | otherwise = x * potencia (x*x) (div n 2)
-- Comprobar con QuickCheck que para todo número natural n y todo
-- número entero x, (potencia x n) es x^n.
-----

```

```

potencia :: Integer -> Integer -> Integer
potencia x 0 = 1
potencia x n | even n    = potencia (x*x) (div n 2)
              | otherwise = x * potencia (x*x) (div n 2)

```

-- La propiedad es

```

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

```

-- La comprobación es

```

-- ghci> quickCheck prop_potencia
-- OK, passed 100 tests.

```

```

-----
-- Ejercicio 8.2. Demostrar por inducción que que para todo número
-- natural n y todo número entero x, (potencia x n) es x^n
-----

```

{-

Demostración: Por inducción en n.

*Caso base: Hay que demostrar que
 para todo x, potencia x 0 = 2^0
 Sea x un número entero, entonces
 potencia x 0*

```
= 1           [por potencia.1]
= 2^0        [por aritmética]
```

*Paso de inducción: Se supone que $n > 0$ y la hipótesis de inducción:
para todo $m < n$ y para todo x , potencia $x (n-1) = x^{(n-1)}$*

Tenemos que demostrar que

para todo x , potencia $x n = x^n$

Lo haremos distinguiendo casos según la paridad de n .

Caso 1: Supongamos que n es par. Entonces, existe un k tal que

*$n = 2*k$. (1)*

Por tanto,

```
potencia n
= potencia (x*x) (div n 2) [por potencia.2]
= potencia (x*x) k       [por (1)]
= (x*x)^k                [por hip. de inducción]
= x^(2*k)                [por aritmética]
= x^n                    [por (1)]
```

Caso 2: Supongamos que n es impar. Entonces, existe un k tal que

*$n = 2*k+1$. (2)*

Por tanto,

```
potencia n
= x * potencia (x*x) (div n 2) [por potencia.3]
= x * potencia (x*x) k       [por (1)]
= x * (x*x)^k                [por hip. de inducción]
= x^(2*k+1)                  [por aritmética]
= x^n                        [por (1)]
```

-}

```
-----
-- Ejercicio 9.1. Comprobar con QuickCheck que para todo par de listas
-- xs, ys se tiene que
--   reverse (xs ++ ys) == reverse ys ++ reverse xs
-----
```

-- La propiedad es

```
prop_reverse_conc :: [Int] -> [Int] -> Bool
```

```
prop_reverse_conc xs ys =
```

```
  reverse (xs ++ ys) == reverse ys ++ reverse xs
```

```

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

-----
-- Ejercicio 9.2. Demostrar por inducción que para todo par de listas
-- xs, ys se tiene que
--   reverse (xs ++ ys) == reverse ys ++ reverse xs
--
-- Las definiciones de reverse y (++) son
--   reverse [] = []                -- reverse.1
--   reverse (x:xs) = reverse xs ++ [x]  -- reverse.2
--
--   [] ++ ys      = ys                -- ++.1
--   (x:xs) ++ ys = x : (xs ++ ys)    -- ++.2
-----

```

```
{-
```

Demostración por inducción en xs.

Caso base: Hay que demostrar que para toda ys,
`reverse ([] ++ ys) == reverse ys ++ reverse []`

En efecto,

```

reverse ([] ++ ys)
= reverse ys                [por ++.1]
= reverse ys ++ []         [por propiedad de ++]
= reverse ys ++ reverse [] [por reverse.1]

```

Paso de inducción: Se supone que para todo ys,
`reverse (xs ++ ys) == reverse ys ++ reverse xs`

Hay que demostrar que para todo ys,

```
reverse ((x:xs) ++ ys) == reverse ys ++ reverse (x:xs)
```

En efecto,

```

reverse ((x:xs) ++ ys)
= reverse (x:(xs ++ ys))           [por ++.2]
= reverse (xs ++ ys) ++ [x]        [por reverse.2]
= (reverse ys ++ reverse xs) ++ [x] [por hip. de inducción]
= reverse ys ++ (reverse xs ++ [x]) [por asociativa de ++]
= reverse ys ++ reverse (x:xs)     [por reverse.2]

```

-}

 -- *Ejercicio 9.3. Demostrar por inducción que para toda lista xs,*
 -- *reverse (reverse xs) = xs*

{-

Demostración por inducción en xs.

Caso Base: Hay que demostrar que
 reverse (reverse []) = []

En efecto,
 reverse (reverse [])
 = *reverse []* [por reverse.1]
 = *[]* [por reverse.1]

Paso de inducción: Se supone que
 reverse (reverse xs) = xs

Hay que demostrar que
 reverse (reverse (x:xs)) = x:xs

En efecto,
 reverse (reverse (x:xs))
 = *reverse (reverse xs ++ [x])* [por reverse.2]
 = *reverse [x] ++ reverse (reverse xs)* [por ejercicio 9.2]
 = *[x] ++ reverse (reverse xs)* [por reverse]
 = *[x] ++ xs* [por hip. de inducción]
 = *x:xs* [por ++.2]

-}

Relación 40

Demostración de propiedades de programas por inducción sobre árboles

```
-----  
-- Introducción                                                    --  
-----  
  
-- En esta relación se plantean ejercicios de demostración por inducción  
-- de propiedades de programas por inducción sobre árboles.  
--  
-- Las transparencias del tema correspondiente se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-14/temas/tema-8.pdf  
  
-----  
-- Importación de librerías                                        --  
-----  
  
import Control.Monad  
import Data.List  
import Test.QuickCheck  
  
-----  
-- Nota 1. En los siguientes ejercicios se demostrarán  
-- propiedades de los árboles binarios definidos como sigue  
-- data Arbol a = Hoja  
--           | Nodo a (Arbol a) (Arbol a)
```

```

--           deriving (Show, Eq)
-- En los ejemplos se usará el siguiente árbol
--   arbol = Nodo 9
--           (Nodo 3
--             (Nodo 2 Hoja Hoja)
--             (Nodo 4 Hoja Hoja))
--           (Nodo 7 Hoja Hoja)

```

```

data Arbol a = Hoja
  | Nodo a (Arbol a) (Arbol a)
  deriving (Show, Eq)

```

```

arbol = Nodo 9
      (Nodo 3
       (Nodo 2 Hoja Hoja)
       (Nodo 4 Hoja Hoja))
      (Nodo 7 Hoja Hoja)

```

```

-- Nota 2. Para comprobar propiedades de árboles con QuickCheck se
-- utilizará el siguiente generador.

```

```

instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbol
  where
    arbol 0      = return Hoja
    arbol n | n>0 = oneof [return Hoja,
                          liftM3 Nodo arbitrary subarbol subarbol]
  where subarbol = arbol (div n 2)

```

```

-- Ejercicio 1. Definir la función
--   espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--   ghci> espejo arbol
--   Nodo 9
--     (Nodo 7 Hoja Hoja)
--     (Nodo 3

```

```

--          (Nodo 4 Hoja Hoja)
--          (Nodo 2 Hoja Hoja))
-----

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

-----

-- Ejercicio 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 3. Demostrar por inducción que para todo árbol x,
--   espejo (espejo x) = x
-----

{-
  Demostración por inducción en x

  Caso base: Hay que demostrar que
    espejo (espejo Hoja) = Hoja
  En efecto,
    espejo (espejo Hoja)
  = espejo Hoja           [por espejo.1]
  = Hoja                  [por espejo.1]

  Paso de inducción: Se supone la hipótesis de inducción
    espejo (espejo i) = i
    espejo (espejo d) = d
-}

```

Hay que demostrar que

`espejo (espejo (Nodo x i d)) = Nodo x i d`

En efecto,

`espejo (espejo (Nodo x i d))`

`= espejo (Nodo x (espejo d) (espejo i))` [por espejo.2]

`= Nodo x (espejo (espejo i)) (espejo (espejo d))` [por espejo.2]

`= Nodo x i d` [por hip. inducción]

-}

```
-----
-- Ejercicio 4. Definir la función
--   preorden :: Arbol a -> [a]
-- tal que (preorden x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> preorden arbol
--   [9,3,2,4,7]
-----
```

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

```
-----
-- Ejercicio 5. Definir la función
--   postorden :: Arbol a -> [a]
-- tal que (postorden x) es la lista correspondiente al recorrido
-- postorden del árbol x; es decir, primero recorre el subárbol
-- izquierdo, a continuación el subárbol derecho y, finalmente, la raíz
-- del árbol. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> postorden arbol
--   [2,4,3,7,9]
-----
```

```
postorden :: Arbol a -> [a]
```

```

postorden Hoja      = []
postorden (Nodo x i d) = postorden i ++ postorden d ++ [x]

-----

-- Ejercicio 6. Comprobar con QuickCheck que para todo árbol x,
--   postorden (espejo x) = reverse (preorden x)
-----

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

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

-----

-- Ejercicio 7. Demostrar por inducción que para todo árbol x,
--   postorden (espejo x) = reverse (preorden x)
-----

{-
  Demostración por inducción en x.

  Caso base: Hay que demostrar que
    postorden (espejo Hoja) = reverse (preorden Hoja)
  En efecto,
    postorden (espejo Hoja)
    = postorden Hoja           [por espejo.1]
    = []                       [por postorden.1]
    = reverse []              [por reverse.1]
    = reverse (preorden Hoja) [por preorden.1]

  Paso de inducción: Se supone la hipótesis de inducción
    postorden (espejo i) = reverse (preorden i)
    postorden (espejo d) = reverse (preorden d)
  Hay que demostrar que
    postorden (espejo (Nodo x i d)) = reverse (preorden (Nodo x i d))
  En efecto,

```

```

postorden (espejo (Nodo x i d))
= postorden (Nodo x (espejo d) (espejo i)) [por espejo.2]
= postorden (espejo d) ++ postorden (espejo i) ++ [x]
                                           [por postorden.2]
= reverse (preorden d) ++ reverse (preorden i) ++ x
                                           [por hip. inducción]
= reverse ([x] ++ preorden (espejo i) ++ preorden (espejo d))
                                           [por ejercicio 1]
= reverse (preorden (Nodo x i d))         [por preorden.1]
-}

```

```

-----
-- Ejercicio 8. Comprobar con QuickCheck que para todo árbol binario
-- x, se tiene que
--   reverse (preorden (espejo x)) = postorden x
-----

```

```

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

```

```

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

```

```

-----
-- Ejercicio 9. Demostrar que para todo árbol binario x, se tiene que
--   reverse (preorden (espejo x)) = preorden x
-----

```

```

{-
  Demostración:
    reverse (preorden (espejo x))
  = postorden (espejo (espejo x)) [por ejercicio 7]
  = postorden x                   [por ejercicio 3]
-}

```

```

-----
-- Ejercicio 10. Definir la función

```

```

--      nNodos :: Arbol a -> Int
--      tal que (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--      ghci> arbol
--      Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--      ghci> nNodos arbol
--      5

```

```

-----
nNodos :: Arbol a -> Int
nNodos Hoja          = 0
nNodos (Nodo x i d) = 1 + nNodos i + nNodos d

```

```

-----
-- Ejercicio 11. Comprobar con QuickCheck que el número de nodos de la
-- imagen especular de un árbol es el mismo que el número de nodos del
-- árbol.

```

```

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

```

```

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

```

```

-----
-- Ejercicio 12. Demostrar por inducción que el número de nodos de la
-- imagen especular de un árbol es el mismo que el número de nodos del
-- árbol.

```

```

{-
  Demostración: Hay que demostrar, por inducción en x, que
    nNodos (espejo x) == nNodos x

```

```

  Caso base: Hay que demostrar que
    nNodos (espejo Hoja) == nNodos Hoja
  En efecto,

```

```
nNodos (espejo Hoja)
= nNodos Hoja          [por espejo.1]
```

Paso de inducción: Se supone la hipótesis de inducción

```
nNodos (espejo i) == nNodos i
nNodos (espejo d) == nNodos d
```

Hay que demostrar que

```
nNodos (espejo (Nodo x i d)) == nNodos (Nodo x i d)
```

En efecto,

```
nNodos (espejo (Nodo x i d))
= nNodos (Nodo x (espejo d) (espejo i))      [por espejo.2]
= 1 + nNodos (espejo d) + nNodos (espejo i)  [por nNodos.2]
= 1 + nNodos d + nNodos i                    [por hip.de inducción]
= 1 + nNodos i + nNodos d                    [por aritmética]
= nNodos (Nodo x i d)                        [por nNodos.2]
```

```
-}
```

```
-----
-- Ejercicio 13. Comprobar con QuickCheck que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol.
-----
```

```
-- La propiedad es
```

```
prop_length_preorden :: Arbol Int -> Bool
```

```
prop_length_preorden x =
```

```
  length (preorden x) == nNodos x
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_length_preorden
```

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

```
-----
-- Ejercicio 14. Demostrar por inducción que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol.
-----
```

```
{-
```

Demostración: Por inducción en x, hay que demostrar que


```
length (preorden x) == nNodos x
```

Caso base: Hay que demostrar que

```
length (preorden Hoja) = nNodos Hoja
```

En efecto,

```
length (preorden Hoja)
= length []           [por preorden.1]
= 0                   [por length.1]
= nNodos Hoja        [por nNodos.1]
```

Paso de inducción: Se supone la hipótesis de inducción

```
length (preorden i) == nNodos i
```

```
length (preorden d) == nNodos d
```

Hay que demostrar que

```
length (preorden (Nodo x i d)) == nNodos (Nodo x i d)
```

En efecto,

```
length (preorden (Nodo x i d))
= length ([x] ++ (preorden i) ++ (preorden d))
  [por preorden.2]
= length [x] + length (preorden i) + length (preorden d)
  [propiedad de length: length (xs++ys) = length xs + length ys]
= 1 + length (preorden i) + length (preorden d)
  [por def. de length]
= 1 + nNodos i + nNodos d
  [por hip. de inducción]
= nNodos (x i d)
  [por nNodos.2]
```

```
-}
```

```
-- -----
-- Ejercicio 15. Definir la función
```

```
-- profundidad :: Arbol a -> Int
```

```
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
```

```
-- ghci> arbol
```

```
-- Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
```

```
-- ghci> profundidad arbol
```

```
-- 3
-- -----
```

```
profundidad :: Arbol a -> Int
```

```

profundidad Hoja = 0
profundidad (Nodo x i d) = 1 + max (profundidad i) (profundidad d)

```

```

-----
-- Ejercicio 16. Comprobar con QuickCheck que para todo árbol binario
-- 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 17. Demostrar por inducción que para todo árbol binario
-- x, se tiene que
--   nNodos x <= 2^(profundidad x) - 1
-----

```

```

{-
  Demostración por inducción en x

```

```

  Caso base: Hay que demostrar que
    nNodos Hoja <= 2^(profundidad Hoja) - 1

```

```

  En efecto,

```

```

    nNodos Hoja
  = 0                [por nNodos.1]
  = 2^0 - 1          [por aritmética]
  = 2^(profundidad Hoja) - 1 [por profundidad.1]

```

```

  Paso de inducción: Se supone la hipótesis de inducción

```

```

    nNodos i <= 2^(profundidad i) - 1
    nNodos d <= 2^(profundidad d) - 1

```

```

  Hay que demostrar que

```

```

    nNodos (Nodo x i d) <= 2^(profundidad (Nodo x i d)) - 1

```

En efecto,

```

nNodos (Nodo x i d)
= 1 + nNodos i + nNodos d
  [por nNodos.1]
<= 1 + (2^(profundidad i) - 1) + (2^(profundidad d) - 1)
  [por hip. de inducción]
= 2^(profundidad i) + 2^(profundidad d) - 1
  [por aritmética]
<= 2^máx(profundidad i,profundidad d)+2^máx(profundidad i,profundidad d)-1
  [por aritmética]
= 2*2^máx(profundidad i,profundidad d) - 1
  [por aritmética]
= 2^(1+máx(profundidad i,profundidad d)) - 1
  [por aritmética]
= 2^profundidad(Nodo x i d) - 1
  [por profundidad.2]
-}

```

```

-----
-- Ejercicio 18. Definir la función
--   nHojas :: Arbol a -> Int
-- tal que (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> nHojas arbol
--   6
-----

```

```

nHojas :: Arbol a -> Int
nHojas Hoja      = 1
nHojas (Nodo x i d) = nHojas i + nHojas d

```

```

-----
-- Ejercicio 19. Comprobar con QuickCheck que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-----

```

```

-- La propiedad es
prop_nHojas :: Arbol Int -> Bool
prop_nHojas x =

```

```

nHojas x == nNodos x + 1

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

-----

-- Ejercicio 20. Demostrar por inducción que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-----

{-
  Demostración: Hay que demostrar, por inducción en  $x$ , que
     $nHojas\ x = nNodos\ x + 1$ 

  Caso base: Hay que demostrar que
     $nHojas\ Hoja = nNodos\ Hoja + 1$ 
  En efecto,
     $nHojas\ Hoja$ 
    = 1                [por nHojas.1]
    = 0 + 1            [por aritmética]
    = nNodos Hoja + 1 [por nNodos.1]

  Paso de inducción: Se supone la hipótesis de inducción
     $nHojas\ i = nNodos\ i + 1$ 
     $nHojas\ d = nNodos\ d + 1$ 
  Hay que demostrar que
     $nHojas\ (Nodo\ x\ i\ d) = nNodos\ (Nodo\ x\ i\ d) + 1$ 
  En efecto,
     $nHojas\ (Nodo\ x\ i\ d)$ 
    =  $nHojas\ i + nHojas\ d$                 [por nHojas.2]
    =  $(nNodos\ i + 1) + (nNodos\ d + 1)$     [por hip. de inducción]
    =  $(1 + nNodos\ i + nNodos\ d) + 1$       [por aritmética]
    =  $nNodos\ (Nodo\ x\ i\ d) + 1$           [por nNodos.2]
-}

-----

-- Ejercicio 21. Definir, usando un acumulador, la función
--   preordenIt :: Arbol a -> [a]
-- tal que (preordenIt x) es la lista correspondiente al recorrido

```

```

-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
-- ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
-- ghci> preordenIt arbol
--   [9,3,2,4,7]
-- Nota: No usar (++) en la definición
-----

```

```

preordenIt :: Arbol a -> [a]
preordenIt x = preordenItAux x []

```

```

preordenItAux :: Arbol a -> [a] -> [a]
preordenItAux Hoja xs          = xs
preordenItAux (Nodo x i d) xs = x : preordenItAux i (preordenItAux d xs)

```

```

-----
-- Ejercicio 22. Comprobar con QuickCheck que preordenIt es
-- equivalente a preorden.
-----

```

```

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

```

```

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

```

```

-----
-- Ejercicio 23. Demostrar que preordenIt es equivalente a preorden.
-----

```

```

prop_preordenItAux :: Arbol Int -> [Int] -> Bool
prop_preordenItAux x ys =
  preordenItAux x ys == preorden x ++ ys

```

```
{-
```

Demostración: La propiedad es consecuencia del siguiente lema:

*Lema: Para todo árbol binario x , se tiene que
para toda ys , $\text{preordenItAux } x \text{ } ys = \text{preorden } x \text{ } ++ \text{ } ys$*

Demostración de la propiedad usando el lema:

```
preordenIt x
= preordenItAux x []      [por preordnIt]
= preorden x ++ []      [por el lema]
= preorden x             [propiedad de ++]
```

Demostración del lema: Por inducción en x .

Caso base: Hay que demotrar que

para toda ys , $\text{preordenItAux Hoja } ys = \text{preorden Hoja } ++ \text{ } ys$

En efecto,

```
preordenItAux Hoja ys
= ys                                [por preordenItAux.1]
= [] ++ ys                          [por propiedad de ++]
= preorden Hoja ++ ys               [por preorden.1]
```

Paso de inducción: Se supone la hipótesis de inducción

para toda ys , $\text{preordenItAux } i \text{ } ys = \text{preorden } i \text{ } ++ \text{ } ys$

para toda ys , $\text{preordenItAux } d \text{ } ys = \text{preorden } d \text{ } ++ \text{ } ys$

Hay que demostrar que

para toda ys , $\text{preordenItAux (Nodo } x \text{ } i \text{ } d) \text{ } ys = \text{preorden (Nodo } x \text{ } i \text{ } d) \text{ } ++ \text{ } ys$

En efecto,

```
preordenItAux (Nodo x i d) ys
= x : (preordenItAux i (preordenItAux d ys)) [por preordenItAux.2]
= x : (preordenItAux i (preorden d ++ ys)) [por hip. de inducción]
= x : (preorden i ++ (preorden d ++ ys)) [por hip. de inducción]
= ([x] ++ preorden i ++ preorden d) ++ ys [por prop. de listas]
= preorden (Nodo x i d) ++ ys           [por preorden.2]
```

-}

Relación 41

Mayorías parlamentarias

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

```
-- En esta relación se presenta un caso de estudio de los tipos  
-- de datos algebraicos para estudiar las mayorías parlamentarias.  
-- Además, con QuickCheck, se comprueban propiedades de las funciones  
-- definidas.
```

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

```
import Data.List  
import Test.QuickCheck
```

```
-----  
-- Ejercicio 1. Definir el tipo de datos Partido para representar los  
-- partidos de un Parlamento. Los partidos son P1, P2, ..., P8. La clase  
-- Partido está contenida en Eq, Ord y Show.  
-----
```

```
data Partido  
  = P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8  
  deriving ( Eq, Ord, Show )
```

```
-- Ejercicio 2. Definir el tipo Parlamentarios para representar el
-- número de parlamentarios que posee un partido en el parlamento.
```

```
type Parlamentarios = Integer
```

```
-- -----
-- Ejercicio 3. Definir el tipo (Tabla a b) para representar una lista
-- de pares de elementos el primero de tipo a y el segundo de tipo
-- b. Definir Asamblea para representar una tabla de partidos y
-- parlamentarios.
```

```
type Tabla a b = [(a,b)]
```

```
type Asamblea = Tabla Partido Parlamentarios
```

```
-- -----
-- Ejercicio 4. Definir la función
-- partidos :: Asamblea -> [Partido]
-- tal que (partidos a) es la lista de partidos en la asamblea a. Por
-- ejemplo,
-- partidos [(P1,3),(P3,5),(P4,3)] ==> [P1,P3,P4]
```

```
partidos :: Asamblea -> [Partido]
```

```
partidos a = [p | (p,_) <- a]
```

```
-- -----
-- Ejercicio 5. Definir la función
-- parlamentarios :: Asamblea -> Integer
-- tal que (parlamentarios a) es el número de parlamentarios en la
-- asamblea a. Por ejemplo,
-- parlamentarios [(P1,3),(P3,5),(P4,3)] ==> 11
```

```
parlamentarios :: Asamblea -> Integer
```

```
parlamentarios a = sum [e | (_,e) <- a]
```

```
-- -----
-- Ejercicio 6. Definir la función
```



```

-- busca :: Eq a => a -> Tabla a b -> b
-- tal que (busca x t) es el valor correspondiente a x en la tabla
-- t. Por ejemplo,
-- Main> busca P3 [(P1,2),(P3,19)]
-- 19
-- Main> busca P8 [(P1,2),(P3,19)]
-- Program error: no tiene valor en la tabla

```

```

busca :: Eq a => a -> Tabla a b -> b
busca x t | null xs = error "no tiene valor en la tabla"
          | otherwise = head xs
  where xs = [b | (a,b) <- t, a == x]

```

```

buscaR :: Eq a => a -> Tabla a b -> b
buscaR x [] = error "no tiene valor en la tabla"
buscaR x ((x',y):xys)
  | x == x' = y
  | otherwise = buscaR x xys

```

```

-- Ejercicio 7. Definir la función
-- busca' :: Eq a => a -> Table a b -> Maybe b
-- tal que (busca' x t) es justo el valor correspondiente a x en la
-- tabla t, o Nothing si x no tiene valor. Por ejemplo,
-- busca' P3 [(P1,2),(P3,19)] == Just 19
-- busca' P8 [(P1,2),(P3,19)] == Nothing

```

```

busca' :: Eq a => a -> Tabla a b -> Maybe b
busca' x t | null xs = Nothing
          | otherwise = Just (head xs)
  where xs = [b | (a,b) <- t, a == x]

```

```

-- Definición por recursión
busca'R :: Eq a => a -> Tabla a b -> Maybe b
busca'R x [] = Nothing
busca'R x ((x',y):xys)
  | x == x' = Just y

```

```
| otherwise      = busca'R x xys
```

```
-----
-- Ejercicio 8. Comprobar con QuickCheck que si (busca' x t) es
-- Nothing, entonces x es distinto de todos los elementos de t.
-----
```

```
-- La propiedad es
```

```
prop_BuscaNothing :: Integer -> [(Integer,Integer)] -> Property
prop_BuscaNothing x t =
  busca' x t == Nothing ==>
  x 'notElem' [ a | (a,_) <- t ]
```

```
-- La comprobación es
```

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

```
-----
-- Ejercicio 9. Comprobar que la función busca' es equivalente a la
-- función lookup del Prelude.
-----
```

```
-- La propiedad es
```

```
prop_BuscaEquivLookup :: Integer -> [(Integer,Integer)] -> Bool
prop_BuscaEquivLookup x t =
  busca' x t == lookup x t
```

```
-- La comprobación es
```

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

```
-----
-- Ejercicio 10. Definir el tipo Coalicion como una lista de partidos.
-----
```

```
type Coalicion = [Partido]
```

```
-----
-- Ejercicio 11. Definir la función
```

```
-- mayoria :: Asamblea -> Integer
```

```

-- tal que (mayoria xs) es el número de parlamentarios que se necesitan
-- para tener la mayoría en la asamblea xs. Por ejemplo,
--   mayoria [(P1,3),(P3,5),(P4,3)] == 6
--   mayoria [(P1,3),(P3,6)]       == 5
-----

mayoria :: Asamblea -> Integer
mayoria xs = parlamentarios xs 'div' 2 + 1

-----

-- Ejercicio 12. Definir la función
--   coaliciones :: Asamblea -> Integer -> [Coalicion]
-- tal que (coaliciones xs n) es la lista de coaliciones necesarias para
-- alcanzar n parlamentarios. Por ejemplo,
--   coaliciones [(P1,3),(P3,5),(P4,3)] 6 == [[P3,P4],[P1,P4],[P1,P3]]
--   coaliciones [(P1,3),(P3,5),(P4,3)] 9 == [[P1,P3,P4]]
--   coaliciones [(P1,3),(P3,5),(P4,3)] 14 == []
--   coaliciones [(P1,3),(P3,5),(P4,3)] 2 == [[P4],[P3],[P1]]
-----

coaliciones :: Asamblea -> Integer -> [Coalicion]
coaliciones _ n | n <= 0 = [[]]
coaliciones [] n       = []
coaliciones ((p,m):xs) n =
  coaliciones xs n ++ [p:c | c <- coaliciones xs (n-m)]

-----

-- Ejercicio 13. Definir la función
--   mayorias :: Asamblea -> [Coalicion]
-- tal que (mayorias a) es la lista de coaliciones mayoritarias en la
-- asamblea a. Por ejemplo,
--   mayorias [(P1,3),(P3,5),(P4,3)] == [[P3,P4],[P1,P4],[P1,P3]]
--   mayorias [(P1,2),(P3,5),(P4,3)] == [[P3],[P1,P4],[P1,P3]]
-----

mayorias :: Asamblea -> [Coalicion]
mayorias asamblea =
  coaliciones asamblea (mayoria asamblea)
-----

```

```
-- Ejercicio 14. Definir el tipo de datos Asamblea.
```

```
-----
data Asamblea2 = A Asamblea
deriving Show
```

```
-----
-- Ejercicio 15. Definir la propiedad
```

```
--   esMayoritaria :: Coalicion -> Asamblea -> Bool
-- tal que (esMayoritaria c a) se verifica si la coalición c es
-- mayoritaria en la asamblea a. Por ejemplo,
--   esMayoritaria [P3,P4] [(P1,3),(P3,5),(P4,3)] == True
--   esMayoritaria [P4] [(P1,3),(P3,5),(P4,3)]    == False
-----
```

```
esMayoritaria :: Coalicion -> Asamblea -> Bool
```

```
esMayoritaria c a =
  sum [busca p a | p <- c] >= mayoria a
```

```
-----
-- Ejercicio 16. Comprobar con QuickCheck que las coaliciones
-- obtenidas por (mayorias asamblea) son coaliciones mayoritarias en la
-- asamblea.
-----
```

```
-- La propiedad es
```

```
prop_MayoriasSonMayoritarias :: Asamblea2 -> Bool
```

```
prop_MayoriasSonMayoritarias (A asamblea) =
  and [esMayoritaria c asamblea | c <- mayorias asamblea]
```

```
-- La comprobación es
```

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

```
-----
-- Ejercicio 17. Definir la función
```

```
--   esMayoritariaMinimal :: Coalicion -> Asamblea -> Bool
-- tal que (esMayoritariaMinimal c a) se verifica si la coalición c es
-- mayoritaria en la asamblea a, pero si se quita a c cualquiera de sus
-- partidos la coalición resultante no es mayoritaria. Por ejemplo,
```

```
-- esMayoritariaMinimal [P3,P4] [(P1,3),(P3,5),(P4,3)] == True
-- esMayoritariaMinimal [P1,P3,P4] [(P1,3),(P3,5),(P4,3)] == False
-----
```

```
esMayoritariaMinimal :: Coalicion -> Asamblea -> Bool
```

```
esMayoritariaMinimal c a =
  esMayoritaria c a &&
  and [not(esMayoritaria (delete p c) a) | p <-c]
```

```
-----
-- Ejercicio 18. Comprobar con QuickCheck si las coaliciones obtenidas
-- por (mayorias asamblea) son coaliciones mayoritarias minimales en la
-- asamblea.
-----
```

```
-- La propiedad es
```

```
prop_MayoriasSonMayoritariasMinimales :: Asamblea2 -> Bool
```

```
prop_MayoriasSonMayoritariasMinimales (A asamblea) =
  and [esMayoritariaMinimal c asamblea | c <- mayorias asamblea]
```

```
-- La comprobación es
```

```
-- Main> quickCheck prop_MayoriasSonMayoritariasMinimales
-- Falsifiable, after 0 tests:
-- A [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
```

```
-- Por tanto, no se cumple la propiedad. Para buscar una coalición no
-- minimal generada por mayorias, definimos la función
```

```
contraejemplo a =
  head [c | c <- mayorias a, not(esMayoritariaMinimal c a)]
```

```
-- el cálculo del contraejemplo es
```

```
-- Main> contraejemplo [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
-- [P4,P6,P7,P8]
```

```
-- La coalición [P4,P6,P7,P8] no es minimal ya que [P4,P6,P8] también es
-- mayoritaria. En efecto,
```

```
-- Main> esMayoritaria [P4,P6,P8]
-- [(P1,1),(P2,0),(P3,1),(P4,1),
-- (P5,0),(P6,1),(P7,0),(P8,1)]
-- True
```

```

-----
-- Ejercicio 19. Definir la función
--   coalicionesMinimales :: Asamblea -> Integer -> [Coalicion,Parlamentarios]
-- tal que (coalicionesMinimales xs n) es la lista de coaliciones
-- minimales necesarias para alcanzar n parlamentarios. Por ejemplo,
--   Main> coalicionesMinimales [(P1,3),(P3,5),(P4,3)] 6
--   [[(P3,P4),8],[(P1,P4),6],[(P1,P3),8]]
--   Main> coalicionesMinimales [(P1,3),(P3,5),(P4,3)] 5
--   [[(P3),5],[(P1,P4),6]]
-----

```

```

coalicionesMinimales :: Asamblea -> Integer -> [(Coalicion,Parlamentarios)]
coalicionesMinimales _ n | n <= 0 = [([],0)]
coalicionesMinimales [] n         = []
coalicionesMinimales ((p,m):xs) n =
  coalicionesMinimales xs n ++
  [(p:ys, t+m) | (ys,t) <- coalicionesMinimales xs (n-m), t<n]

```

```

-----
-- Ejercicio 20. Definir la función
--   mayoriasMinimales :: Asamblea -> [Coalicion]
-- tal que (mayoriasMinimales a) es la lista de coaliciones mayoritarias
-- minimales en la asamblea a. Por ejemplo,
--   mayoriasMinimales [(P1,3),(P3,5),(P4,3)] == [[P3,P4],[P1,P4],[P1,P3]]
-----

```

```

mayoriasMinimales :: Asamblea -> [Coalicion]
mayoriasMinimales asamblea =
  [c | (c,_) <- coalicionesMinimales asamblea (mayoria asamblea)]

```

```

-----
-- Ejercicio 21. Comprobar con QuickCheck que las coaliciones
-- obtenidas por (mayoriasMinimales asamblea) son coaliciones
-- mayoritarias minimales en la asamblea.
-----

```

```

-- La propiedad es
prop_MayoriasMinimalesSonMayoritariasMinimales :: Asamblea2 -> Bool
prop_MayoriasMinimalesSonMayoritariasMinimales (A asamblea) =

```

```

and [esMayoritariaMinimal c asamblea
     | c <- mayoriasMinimales asamblea]

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

-----
-- Funciones auxiliares
-----

-- (listaDe n g) es una lista de n elementos, donde cada elemento es
-- generado por g. Por ejemplo,
--   Main> muestra (listaDe 3 (arbitrary :: Gen Int))
--   [-1,1,-1]
--   [-2,-4,-1]
--   [1,-1,0]
--   [1,-1,1]
--   [1,-1,1]
--   Main> muestra (listaDe 3 (arbitrary :: Gen Bool))
--   [False,True,False]
--   [True,True,False]
--   [False,False,True]
--   [False,False,True]
--   [True,False,True]
listaDe :: Int -> Gen a -> Gen [a]
listaDe n g = sequence [g | i <- [1..n]]

-- paresDeIgualLongitud genera pares de listas de igual longitud. Por
-- ejemplo,
--   Main> muestra (paresDeIgualLongitud (arbitrary :: Gen Int))
--   ([-4,5],[-4,2])
--   ([],[ ])
--   ([0,0],[-2,-3])
--   ([2,-2],[-2,1])
--   ([0],[-1])
--   Main> muestra (paresDeIgualLongitud (arbitrary :: Gen Bool))
--   ([False,True,False],[True,True,True])
--   ([True],[True])
--   ([],[ ])

```

```

--      ([False],[False])
--      ([],[ ])
paresDeIgualLongitud :: Gen a -> Gen ([a],[a])
paresDeIgualLongitud gen =
  do n <- arbitrary
     xs <- listaDe (abs n) gen
     ys <- listaDe (abs n) gen
     return (xs,ys)

-- generaAsamblea es un generador de datos de tipo Asamblea. Por ejemplo,
--      Main> muestra generaAsamblea
--      A [(P1,1),(P2,1),(P3,0),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
--      A [(P1,0),(P2,1),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
--      A [(P1,1),(P2,2),(P3,0),(P4,1),(P5,0),(P6,1),(P7,2),(P8,0)]
--      A [(P1,1),(P2,0),(P3,1),(P4,0),(P5,0),(P6,1),(P7,1),(P8,1)]
--      A [(P1,1),(P2,0),(P3,0),(P4,0),(P5,1),(P6,1),(P7,1),(P8,0)]
generaAsamblea :: Gen Asamblea2
generaAsamblea =
  do xs <- listaDe 8 (arbitrary :: Gen Integer)
     return (A (zip [P1,P2,P3,P4,P5,P6,P7,P8] (map abs xs)))

instance Arbitrary Asamblea2 where
  arbitrary = generaAsamblea
  -- coarbitrary = undefined

```