Exámenes de “Programación funcional con Haskell”

Vol. 2 (Curso 2010–2011)

José A. Alonso Jiménez

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

Sevilla, 22 de noviembre de 2011

Índice general

Introducción **5**

1 Exámenes del grupo 1 **7**

José A. Alonso y Agustín Riscos

1.1 Examen 1 (25 de Octubre de 2010) 7
1.2 Examen 2 (22 de Noviembre de 2010) 8
1.3 Examen 3 (20 de Diciembre de 2010) 11
1.4 Examen 4 (11 de Febrero de 2011) 14
1.5 Examen 5 (14 de Marzo de 2011) 16
1.6 Examen 6 (11 de Abril de 2011) 20
1.7 Examen 7 (23 de Mayo de 2011) 23
1.8 Examen 8 (24 de Junio de 2011) 27
1.9 Examen 9 (8 de Julio de 2011) 34
1.10 Examen 10 (16 de Septiembre de 2011) 40
1.11 Examen 11 (22 de Noviembre de 2011) 45

2 Exámenes del grupo 2 **53**

María J. Hidalgo

2.1 Examen 1 (29 de Octubre de 2010) 53
2.2 Examen 2 (26 de Noviembre de 2010) 56
2.3 Examen 3 (17 de Diciembre de 2010) 59
2.4 Examen 4 (11 de Febrero de 2011) 65
2.5 Examen 5 (14 de Marzo de 2011) 65
2.6 Examen 6 (15 de abril de 2011) 65
2.7 Examen 7 (27 de mayo de 2011) 70
2.8 Examen 8 (24 de Junio de 2011) 80
2.9 Examen 9 (8 de Julio de 2011) 80
2.10 Examen 10 (16 de Septiembre de 2011) 80
2.11 Examen 11 (22 de Noviembre de 2011) 80

A Resumen de funciones predefinidas de Haskell **81**
A.1 Resumen de funciones sobre TAD en Haskell 83

B Método de Pólya para la resolución de problemas **87**
B.1 Método de Pólya para la resolución de problemas matemáticos . 87
B.2 Método de Pólya para resolver problemas de programación . . . 88

Bibliografía **91**

Introducción

Este libro es una recopilación de las soluciones de ejercicios de los exámenes de programación funcional con Haskell de la asignatura de Informática (curso 2010–11) del Grado en Matemática de la Universidad de Sevilla.

Los exámenes se realizaron en el aula de informática y su duración fue de 2 horas. La materia de cada examen es la impartida desde el comienzo del curso (generalmente, el 1 de octubre) hasta la fecha del examen. Dicha materia se encuentra en los libros de temas y ejercicios del curso:

- Temas de programación funcional (curso 2010–11) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2010–11) [2]
- Piensa en Haskell (Ejercicios de programación funcional con Haskell) [3]

El libro consta de 2 capítulos correspondientes a 2 grupos de la asignatura. En cada capítulo hay una sección por cada uno de los exámenes del grupo. Los ejercicios de cada examen han sido propuestos por los profesores de su grupo (cuyos nombres aparecen en el título del capítulo). Sin embargo, los he modificado para unificar el estilo de su presentación.

Finalmente, el libro contiene dos apéndices. Uno con el método de Polya de resolución de problemas (sobre el que se hace énfasis durante todo el curso) y el otro con un resumen de las funciones de Haskell de uso más frecuente.

Los códigos del libro están disponibles en GitHub [4]

Este libro es el segundo volumen de la serie de recopilaciones de exámenes de programación funcional con Haskell. El volumen anterior es

- Exámenes de "Programación funcional con Haskell". Vol. 1 (Curso 2009–2010) [5]

[1] https://www.cs.us.es/~jalonso/cursos/i1m-10/temas/2010-11-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-10/ejercicios/ejercicios-I1M-2010.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol2
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

José A. Alonso
Sevilla, 22 de novoembre de 2011

1

Exámenes del grupo 1

José A. Alonso y Agustín Riscos

1.1. Examen 1 (25 de Octubre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (25 de octubre de 2010)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. 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 2. 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)

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

```haskell
-- Ejercicio 3. 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
-- ---------------------------------------------------------------------

-- 1ª definición:
mediano x y z = x + y + z- minimum [x,y,z] - maximum [x,y,z]

-- 2ª definición:
mediano2 x y z
    | a <= x && x <= b = x
    | a <= y && y <= b = y
    | otherwise        = z
    where a = minimum [x,y,z]
          b = maximum [x,y,z]

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

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

1.2. Examen 2 (22 de Noviembre de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (22 de noviembre de 2010)
-- ---------------------------------------------------------------------

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

```
--    8!! = 8*6*4*2   = 384
--    9!! = 9*7*5*3*1 = 945
--
-- Definir, por recursión, la función
--    dobleFactorial :: Integer -> Integer
-- tal que (dobleFactorial n) es el doble factorial de n. Por ejemplo,
--    dobleFactorial 8  ==  384
--    dobleFactorial 9  ==  945
-- ---------------------------------------------------------------------

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

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

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

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

```haskell
--    distancia "roma"   "comino"  ==  2
--    distancia "roma"   "camino"  ==  3
--    distancia "romano" "ron"     ==  1
--    distancia "romano" "cama"    ==  2
--    distancia "romano" "rama"    ==  1
-- ---------------------------------------------------------------------

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

-- 2ª definición (por recursión):
distancia2 :: Eq a => [a] -> [a] -> Int
distancia2 [] ys = 0
distancia2 xs [] = 0
distancia2 (x:xs) (y:ys) | x /= y    = 1 + distancia2 xs ys
                         | otherwise = distancia2 xs ys

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

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

-- 2ª definición (por recursión):
aproximaPi2 n = sqrt(6 * aux n)
    where aux 1 = 1
          aux n = 1/n^2 + aux (n-1)
```

1.3. Examen 3 (20 de Diciembre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 3º examen de evaluación continua (20 de diciembre de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por recursión, la función
--    sumaR :: Num b => (a -> b) -> [a] -> b
-- tal que (suma f xs) es la suma de los valores obtenido aplicando la
-- función f a lo elementos de la lista xs. Por ejemplo,
--    sumaR (*2)  [3,5,10]  ==  36
--    sumaR (/10) [3,5,10]  ==  1.8
 -- ---------------------------------------------------------------------

sumaR :: Num b => (a -> b) -> [a] -> b
sumaR f []     = 0
sumaR f (x:xs) = f x + sumaR f xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por plegado, la función
--    sumaP :: Num b => (a -> b) -> [a] -> b
-- tal que (suma f xs) es la suma de los valores obtenido aplicando la
-- función f a lo elementos de la lista xs. Por ejemplo,
--    sumaP (*2)  [3,5,10]  ==  36
--    sumaP (/10) [3,5,10]  ==  1.8
-- ---------------------------------------------------------------------

sumaP :: Num b => (a -> b) -> [a] -> b
sumaP f = foldr (\x y -> f x + y) 0

-- ---------------------------------------------------------------------
-- Ejercicio 3. El enunciado del problema 1 de la Olimpiada
-- Iberoamericana de Matemática Universitaria del 2006 es el siguiente:
--    Sean m y n números enteros mayores que 1. Se definen los conjuntos
--    P(m) = {1/m, 2/m,..., (m-1)/m} y P(n) = {1/n, 2/n,..., (n-1)/n}.
--    La distancia entre P(m) y P(n) es
--    mín {|a - b| : a en P(m), b en P(n)}.
```

```haskell
--
-- Definir la función
--    distancia :: Float -> Float -> Float
-- tal que (distancia m n) es la distancia entre P(m) y P(n). Por
-- ejemplo,
--    distancia 2 7 == 7.142857e-2
--    distancia 2 8 == 0.0
-- ---------------------------------------------------------------------

distancia :: Float -> Float -> Float
distancia m n =
    minimum [abs (i/m - j/n) | i <- [1..m-1], j <- [1..n-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El enunciado del problema 580 de "Números y algo
-- más.." es el siguiente:
--    ¿Cuál es el menor número que puede expresarse como la suma de 9,
--    10 y 11 números consecutivos?
-- (El problema se encuentra en http://goo.gl/1K3t7 )
--
-- A lo largo de los distintos apartados de este ejercicio se resolverá
-- el problema.
--
-- Definir la función
--    consecutivosConSuma :: Int -> Int -> [[Int]]
-- tal que (consecutivosConSuma x n) es la lista de listas de n números
-- consecutivos cuya suma es x. Por ejemplo,
--    consecutivosConSuma 12 3  ==  [[3,4,5]]
--    consecutivosConSuma 10 3  ==  []
-- ---------------------------------------------------------------------

consecutivosConSuma :: Int -> Int -> [[Int]]
consecutivosConSuma x n =
    [[y..y+n-1] | y <- [1..x], sum [y..y+n-1] == x]

-- Se puede hacer una definición sin búsqueda, ya que por la fórmula de
-- la suma de progresiones aritméticas, la expresión
--    sum [y..y+n-1] == x
-- se reduce a
--    (y+(y+n-1))n/2 = x
```

```haskell
-- De donde se puede despejar la y, ya que
--    2yn+n^2-n = 2x
--    y = (2x-n^2+n)/2n
-- De la anterior anterior se obtiene la siguiente definición de
-- consecutivosConSuma que no utiliza búsqueda.

consecutivosConSuma2 :: Int -> Int -> [[Int]]
consecutivosConSuma2 x n
    | z >= 0 && mod z (2*n) == 0 = [[y..y+n-1]]
    | otherwise                  = []
    where z = 2*x-n^2+n
          y = div z (2*n)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    esSuma :: Int -> Int -> Bool
-- tal que (esSuma x n) se verifica si x es la suma de n números
-- naturales consecutivos. Por ejemplo,
--    esSuma 12 3  ==  True
--    esSuma 10 3  ==  False
-- ---------------------------------------------------------------------

esSuma :: Int -> Int -> Bool
esSuma x n = consecutivosConSuma x n /= []

-- También puede definirse directamente sin necesidad de
-- consecutivosConSuma como se muestra a continuación.
esSuma2 :: Int -> Int -> Bool
esSuma2 x n = or [sum [y..y+n-1] == x | y <- [1..x]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    menorQueEsSuma :: [Int] -> Int
-- tal que (menorQueEsSuma ns) es el menor número que puede expresarse
-- como suma de tantos números consecutivos como indica ns. Por ejemplo,
--    menorQueEsSuma [3,4]  ==  18
-- Lo que indica que 18 es el menor número se puede escribir como suma
-- de 3 y de 4 números consecutivos. En este caso, las sumas son
-- 18 = 5+6+7 y 18 = 3+4+5+6.
-- ---------------------------------------------------------------------
```

```haskell
menorQueEsSuma :: [Int] -> Int
menorQueEsSuma ns =
    head [x | x <- [1..], and [esSuma x n | n <- ns]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Usando la función menorQueEsSuma calcular el menor
-- número que puede expresarse como la suma de 9, 10 y 11 números
-- consecutivos.
-- ---------------------------------------------------------------------

-- La solución es
--    ghci> menorQueEsSuma [9,10,11]
--    495
```

1.4. Examen 4 (11 de Febrero de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (11 de febrero de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. (Problema 303 del proyecto Euler)
-- Definir la función
--    multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
-- tal que (multiplosRestringidos n x) es la lista de los múltiplos de n
-- cuyas cifras verifican la propiedad p. Por ejemplo,
--    take 4 (multiplosRestringidos 5 (<=3))  ==  [10,20,30,100]
--    take 5 (multiplosRestringidos 3 (<=4))  ==  [3,12,21,24,30]
--    take 5 (multiplosRestringidos 3 even)   ==  [6,24,42,48,60]
-- ---------------------------------------------------------------------

multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
multiplosRestringidos n p =
    [y | y <- [n,2*n..], and [p x | x <- cifras y]]

-- (cifras n) es la lista de las cifras de n, Por ejemplo,
--    cifras 327  ==  [3,2,7]
cifras :: Int -> [Int]
cifras n = [read [x] | x <- show n]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumaDeDosPrimos :: Int -> [(Int,Int)]
-- tal que (sumaDeDosPrimos n) es la lista de las distintas
-- descomposiciones de n como suma de dos nuúmeros primos. Por ejemplo,
--    sumaDeDosPrimos 30  ==  [(7,23),(11,19),(13,17)]
-- Calcular, usando la función sumaDeDosPrimos, el menor número que
-- puede escribirse de 10 formas distintas como suma de dos primos.
-- ---------------------------------------------------------------------

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

-- primos es la lista de los números primos
primos :: [Int]
primos = criba [2..]
    where criba []     = []
          criba (n:ns) = n : criba (elimina n ns)
          elimina n xs = [x | x <- xs, x `mod` n /= 0]

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se consideran los árboles binarios
-- definidos por
--    data Arbol = H Int
--               | N Arbol Int Arbol
--               deriving (Show, Eq)
-- Por ejemplo, el árbol
--         5
--        / \
--       /   \
--      9     7
--     / \   / \
--    1   4 6   8
```

```haskell
-- se representa por
--    N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8))
-- Definir la función
--    maximoArbol :: Arbol -> Int
-- tal que (maximoArbol a) es el máximo valor en el árbol a. Por
-- ejemplo,
--    maximoArbol (N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8)))  ==  9
-- ---------------------------------------------------------------------

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

maximoArbol :: Arbol -> Int
maximoArbol (H x) = x
maximoArbol (N i x d) = maximum [x, maximoArbol i, maximoArbol d]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    segmentos :: (a -> Bool) -> [a] -> [a]
-- tal que (segmentos p xs) es la lista de los segmentos de xs de 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 _ [] = []
segmentos p xs =
    takeWhile p xs : segmentos p (dropWhile (not.p) (dropWhile p xs))
```

1.5. Examen 5 (14 de Marzo de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 5º examen de evaluación continua (14 de marzo de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Se consideran las funciones
--    duplica  :: [a] -> [a]
--    longitud :: [a] -> Int
-- tales que
```

```
--    (duplica xs) es la lista obtenida duplicando los elementos de xs y
--    (longitud xs) es el número de elementos de xs.
-- Por ejemplo,
--    duplica  [7,2,5]  ==  [7,7,2,2,5,5]
--    longitud [7,2,5]  ==  3
--
-- Las definiciones correspondientes son
--    duplica [] = []                     -- duplica.1
--    duplica (x:xs) = x:x:duplica xs     -- duplica.2
--
--    longitud [] = 0                     -- longitud.1
--    longitud (x:xs) = 1 + longitud xs   -- longitud.2
-- Demostrar por inducción que
--    longitud (duplica xs) = 2 * longitud xs
-- ----------------------------------------------------------------------

{-
 Demostración: Hay que demostrar que
    longitud (duplica xs) = 2 * longitud xs
 Lo haremos por inducción en xs.

 Caso base: Hay que demostrar que
    longitud (duplica []) = 2 * longitud []
 En efecto
    longitud (duplica xs)
    = longitud []            [por duplica.1]
    = 0                      [por longitud.1]
    = 2 * 0                  [por aritmética]
    = longitud []            [por longitud.1]

 Paso de inducción: Se supone la hipótesis de inducción
    longitud (duplica xs) = 2 * longitud xs
 Hay que demostrar que
    longitud (duplica (x:xs)) = 2 * longitud (x:xs)
 En efecto,
    longitud (duplica (x:xs))
    = longitud (x:x:duplica xs)       [por duplica.2]
    = 1 + longitud (x:duplica xs)     [por longitud.2]
    = 1 + 1 + longitud (duplica xs)   [por longitud.2]
    = 1 + 1 + 2*(longitud xs)         [por hip. de inducción]
```

```
    = 2 * (1 + longitud xs)           [por aritmética]
    = 2 * longitud (x:xs)             [por longitud.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las expresiones aritméticas 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))
--
-- 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
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    esFib :: Int -> Bool
-- tal que (esFib x) se verifica si existe un número n tal que x es el
-- n-ésimo término de la sucesión de Fibonacci. Por ejemplo,
--    esFib 89  ==  True
--    esFib 69  ==  False
-- ---------------------------------------------------------------------

esFib :: Int -> Bool
esFib n = n == head (dropWhile (<n) fibs)

-- fibs es la sucesión de Fibonacci. Por ejemplo,
```

```
--    take 10 fibs  ==  [0,1,1,2,3,5,8,13,21,34]
fibs :: [Int]
fibs = 0:1:[x+y | (x,y) <- zip fibs (tail fibs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4 El ejercicio 4 de la Olimpiada Matemáticas de 1993 es el
-- siguiente:
--    Demostrar que para todo número primo p distinto de 2 y de 5,
--    existen infinitos múltiplos de p de la forma 1111......1 (escrito
--    sólo con unos).
--
-- Definir la función
--    multiplosEspeciales :: Integer -> Int -> [Integer]
-- tal que (multiplosEspeciales p n) es una lista de n múltiplos p de la
-- forma 1111...1 (escrito sólo con unos), donde p es un número primo
-- distinto de 2 y 5. Por ejemplo,
--    multiplosEspeciales 7 2  ==  [111111,111111111111]
-- ---------------------------------------------------------------------

-- 1ª definición
multiplosEspeciales :: Integer -> Int -> [Integer]
multiplosEspeciales p n = take n [x | x <- unos, mod x p == 0]

-- unos es la lista de los números de la forma 111...1 (escrito sólo con
-- unos). Por ejemplo,
--    take 5 unos  ==  [1,11,111,1111,11111]
unos :: [Integer]
unos = 1 : [10*x+1 | x <- unos]

-- Otra definición no recursiva de unos es
unos2 :: [Integer]
unos2 = [div (10^n-1) 9 | n <- [1..]]

-- 2ª definición (sin usar unos)
multiplosEspeciales2 :: Integer -> Int -> [Integer]
multiplosEspeciales2 p n =
    [div (10^((p-1)*x)-1) 9 | x <- [1..fromIntegral n]]
```

1.6. Examen 6 (11 de Abril de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 6º examen de evaluación continua (11 de abril de 2011)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 1. Las expresiones aritméticas pueden representarse usando
-- el siguiente tipo de datos
--    data Expr = N Int | V Char | S Expr Expr | P Expr Expr
--              deriving Show
-- Por ejemplo, la expresión 2*(a+5) se representa por
--    P (N 2) (S (V 'a') (N 5))
--
-- Definir la función
--    valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- 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
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    ocurrencias :: Ord a => a -> Monticulo a -> Int
-- tal que (ocurrencias x m) es el número de veces que ocurre el
```

```haskell
-- elemento x en el montículo m. Por ejemplo,
--    ocurrencias 7 (foldr inserta vacio [6,1,7,8,7,5,7])  ==  3
-- ---------------------------------------------------------------------

ocurrencias :: Ord a => a -> Monticulo a -> Int
ocurrencias x m
    | esVacio m = 0
    | x < mm    = 0
    | x == mm   = 1 + ocurrencias x rm
    | otherwise = ocurrencias x rm
    where mm = menor m
          rm = resto m

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se consideran los tipos de los vectores y de las
-- matrices definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
--
-- Definir la función
--    diagonal :: Num a => Vector a -> Matriz a
-- tal que (diagonal v) es la matriz cuadrada cuya diagonal es el vector
-- v. Por ejemplo,
--    ghci> diagonal (array (1,3) [(1,7),(2,6),(3,5)])
--    array ((1,1),(3,3)) [((1,1),7),((1,2),0),((1,3),0),
--                         ((2,1),0),((2,2),6),((2,3),0),
--                         ((3,1),0),((3,2),0),((3,3),5)]
-- ---------------------------------------------------------------------

type Vector a = Array Int a
type Matriz a = Array (Int,Int) a

diagonal :: Num a => Vector a -> Matriz a
diagonal v =
    array ((1,1),(n,n))
          [((i,j),f i j) | i <- [1..n], j <- [1..n]]
    where n = snd (bounds v)
          f i j | i == j    = v!i
                | otherwise = 0
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. El enunciado del problema 652 de "Números y algo más" es
-- el siguiente
--    Si factorizamos los factoriales de un número en función de sus
--    divisores primos y sus potencias, ¿Cuál es el menor número N tal
--    que entre los factores primos y los exponentes de estos, N!
--    contiene los dígitos del cero al nueve?
--    Por ejemplo
--        6! = 2^4*3^2*5^1, le faltan los dígitos 0,6,7,8 y 9
--       12! = 2^10*3^5*5^2*7^1*11^1, le faltan los dígitos 4,6,8 y 9
--
-- Definir la función
--    digitosDeFactorizacion :: Integer -> [Integer]
-- tal que (digitosDeFactorizacion n) es el conjunto de los dígitos que
-- aparecen en la factorización de n. Por ejemplo,
--    digitosDeFactorizacion (factorial 6)   ==  [1,2,3,4,5]
--    digitosDeFactorizacion (factorial 12)  ==  [0,1,2,3,5,7]
-- Usando la función anterior, calcular la solución del problema.
-- ---------------------------------------------------------------------

digitosDeFactorizacion :: Integer -> [Integer]
digitosDeFactorizacion n =
   sort (nub (concat [digitos x | x <- numerosDeFactorizacion n]))

-- (digitos n) es la lista de los digitos del número n. Por ejemplo,
--    digitos 320274  ==  [3,2,0,2,7,4]
digitos :: Integer -> [Integer]
digitos n = [read [x] | x <- show n]

-- (numerosDeFactorizacion n) es el conjunto de los números en la
-- factorización de n. Por ejemplo,
--    numerosDeFactorizacion 60  ==  [1,2,3,5]
numerosDeFactorizacion :: Integer -> [Integer]
numerosDeFactorizacion n =
   sort (nub (aux (factorizacion n)))
   where aux [] = []
         aux ((x,y):zs) = x : y : aux zs

-- (factorización n) es la factorización de n. Por ejemplo,
--    factorizacion 300  ==  [(2,2),(3,1),(5,2)]
```

```
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n =
    [(head xs, fromIntegral (length xs)) | xs <- group (factorizacion' n)]

-- (factorizacion' n) es la lista de todos los factores primos de n; es
-- decir, es una lista de números primos cuyo producto es n. Por ejemplo,
--    factorizacion 300  ==  [2,2,3,5,5]
factorizacion' :: Integer -> [Integer]
factorizacion' n | n == 1    = []
                 | otherwise = x : factorizacion' (div n x)
                 where x = menorFactor n

-- (menorFactor n) es el menor factor primo de n. Por ejemplo,
--    menorFactor 15  ==  3
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 5  ==  120
factorial :: Integer -> Integer
factorial n = product [1..n]

-- Para calcular la solución, se define la constante
solucion =
    head [n | n <- [1..], digitosDeFactorizacion (factorial n) == [0..9]]

-- El cálculo de la solución es
--    ghci> solucion
--    49
```

1.7. Examen 7 (23 de Mayo de 2011)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 7º examen de evaluación continua (23 de mayo de 2011)
-- ---------------------------------------------------------------------

import Data.Array
import GrafoConVectorDeAdyacencia
import RecorridoEnAnchura
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Se considera que los puntos del plano se representan por
-- pares de números como se indica a continuación
--    type Punto = (Double,Double)
--
-- Definir la función
--    cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
-- tal que (cercanos ps qs) es un par de puntos, el primero de ps y el
-- segundo de qs, que son los más cercanos (es decir, no hay otro par
-- (p',q') con p' en ps y q' en qs tales que la distancia entre p' y q'
-- sea menor que la que hay entre p y q). Por ejemplo,
--    ghci> cercanos [(2,5),(3,6)] [(4,3),(1,0),(7,9)]
--    ((2.0,5.0),(4.0,3.0))
-- ---------------------------------------------------------------------

type Punto = (Double,Double)

cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
cercanos ps qs = (p,q)
    where (d,p,q) = minimum [(distancia p q, p, q) | p <- ps, q <-qs]
          distancia (x,y) (u,v) = sqrt ((x-u)^2+(y-v)^2)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las relaciones binarias pueden representarse mediante
-- conjuntos de pares de elementos.
--
-- Definir la función
--    simetrica :: Eq a => [(a,a)] -> Bool
-- tal que (simetrica r) se verifica si la relación binaria r es
-- simétrica. Por ejemplo,
--    simetrica [(1,3),(2,5),(3,1),(5,2)]  ==  True
--    simetrica [(1,3),(2,5),(3,1),(5,3)]  ==  False
-- ---------------------------------------------------------------------

simetrica :: Eq a => [(a,a)] -> Bool
simetrica [] = True
simetrica ((x,y):r)
    | x == y    = True
    | otherwise = elem (y,x) r && simetrica (borra (y,x) r)
```

```haskell
-- (borra x ys) es la lista obtenida borrando el elemento x en ys. Por
-- ejemplo,
--    borra 2 [3,2,5,7,2,3]  ==  [3,5,7,3]
borra :: Eq a => a -> [a] -> [a]
borra x ys = [y | y <- ys, y /= x]

-- ---------------------------------------------------------------------
-- 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 False (1,3) [(1,2,0),(3,2,0)])  ==  True
--    conexo (creaGrafo False (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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los tipos de los vectores y de las
-- matrices definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
-- y, como ejemplo, la matriz q definida por
--    q :: Matriz Int
--    q = array ((1,1),(2,2)) [((1,1),1),((1,2),1),((2,1),1),((2,2),0)]
--
-- 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,
--    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
```

```haskell
--    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?
-- ---------------------------------------------------------------------

type Vector a = Array Int a
type Matriz a = Array (Int,Int) a

q :: Matriz Int
q = array ((1,1),(2,2)) [((1,1),1),((1,2),1),((2,1),1),((2,2),0)]

potencia :: Num a => Matriz a -> Int -> Matriz a
potencia p 0 = identidad (numFilas p)
potencia p (n+1) = prodMatrices p (potencia p n)

-- (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)]
```

```haskell
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

-- (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 = numFilas p

-- (numFilas m) es el número de filas de la matriz m. Por ejemplo,
--    numFilas q  ==  2
numFilas :: Num a => Matriz a -> Int
numFilas = fst . snd . bounds

-- (numColumnas m) es el número de columnas de la matriz m. Por ejemplo,
--    numColumnas q  ==  2
numColumnas:: Num a => Matriz a -> Int
numColumnas = snd . snd . bounds

-- (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 = numFilas p
          n = numColumnas q

-- 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.
```

1.8. Examen 8 (24 de Junio de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 8º examen de evaluación continua (24 de junio de 2011)
```

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

import Test.QuickCheck
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    ordena :: (a -> a -> Bool) -> [a] -> [a]
-- tal que (ordena r xs) es la lista obtenida ordenando los elementos de
-- xs según la relación r. Por ejemplo,
--   ghci> ordena (\x y -> abs x < abs y) [-6,3,7,-9,11]
--   [3,-6,7,-9,11]
--   ghci> ordena (\x y -> length x < length y) [[2,1],[3],[],[1]]
--   [[],[3],[1],[2,1]]
-- ---------------------------------------------------------------------

ordena :: (a -> a -> Bool) -> [a] -> [a]
ordena _ [] = []
ordena r (x:xs) =
    (ordena r menores) ++ [x] ++ (ordena r mayores)
        where menores = [y | y <- xs, r y x]
              mayores = [y | y <- xs, not (r y x)]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se consideran el tipo de las matrices definido por
--    type Matriz a = Array (Int,Int) a
-- y, como ejemplo, la matriz q definida por
--    q :: Matriz Int
--    q = array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),3),((2,2),1)]
--
-- 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,
--    indicesMaximo q  ==  [(1,1),(2,1)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

q :: Matriz Int
```

```haskell
q = array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),3),((2,2),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 3. Los montículo se pueden representar mediante el
-- siguiente tipo de dato algebraico.
--    data Monticulo = Vacio
--                   | M Int Monticulo Monticulo
--                   deriving Show
-- Por ejemplo, el montículo
--         1
--        / \
--       /   \
--      5     4
--     / \   /
--    7   6 8
-- se representa por
--    m1, m2, m3 :: Monticulo
--    m1 = M 1 m2 m3
--    m2 = M 5 (M 7 Vacio Vacio) (M 6 Vacio Vacio)
--    m3 = M 4 (M 8 Vacio Vacio) Vacio
--
-- Definir las funciones
--    ramaDerecha :: Monticulo -> [Int]
--    rango :: Monticulo -> Int
-- tales que (ramaDerecha m) es la rama derecha del montículo m. Por ejemplo,
--    ramaDerecha m1  ==  [1,4]
--    ramaDerecha m2  ==  [5,6]
--    ramaDerecha m3  ==  [4]
-- y (rango m) es el rango del montículo m; es decir, la menor distancia
-- desde la raíz de m a un montículo vacío. Por ejemplo,
--    rango m1        ==  2
--    rango m2        ==  2
--    rango m3        ==  1
-- ---------------------------------------------------------------------

data Monticulo = Vacio
```

```haskell
              | M Int Monticulo Monticulo
              deriving Show

m1, m2, m3 :: Monticulo
m1 = M 1 m2 m3
m2 = M 5 (M 7 Vacio Vacio) (M 6 Vacio Vacio)
m3 = M 4 (M 8 Vacio Vacio) Vacio

ramaDerecha :: Monticulo -> [Int]
ramaDerecha Vacio = []
ramaDerecha (M v i d) = v : ramaDerecha d

rango :: Monticulo -> Int
rango Vacio = 0
rango (M _ i d) = 1 + min (rango i) (rango d)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Los polinomios pueden representarse mediante listas
-- dispersas. Por ejemplo, el polinomio x^5+3x^4-5x^2+x-7 se representa
-- por [1,3,0,-5,1,-7]. En dicha lista, obviando el cero, se producen
-- tres cambios de signo: del 3 al -5, del -5 al 1 y del 1 al
-- -7. Llamando C(p) al número de cambios de signo en la lista de
-- coeficientes del polinomio p(x), tendríamos entonces que en este caso
-- C(p)=3.
--
-- La regla de los signos de Descartes dice que el número de raíces
-- reales positivas de una ecuación polinómica con coeficientes reales
-- igualada a cero es, como mucho, igual al número de cambios de signo
-- que se produzcan entre sus coeficientes (obviando los ceros). Por
-- ejemplo, en el caso anterior la ecuación tendría como mucho tres
-- soluciones reales positivas, ya que C(p)=3.
--
-- Además, si la cota C(p) no se alcanza, entonces el número de raíces
-- positivas de la ecuación difiere de ella un múltiplo de dos. En el
-- ejemplo anterior esto significa que la ecuación puede tener tres
-- raíces positivas o tener solamente una, pero no podría ocurrir que
-- tuviera dos o que no tuviera ninguna.
--
-- Definir, por comprensión, la función
--    cambiosC :: [Int] -> [(Int,Int)]
```

```haskell
-- tal que (cambiosC xs) es la lista de los pares de elementos de xs con
-- signos distintos, obviando los ceros. Por ejemplo,
--    cambiosC [1,3,0,-5,1,-7]  ==  [(3,-5),(-5,1),(1,-7)]
-- ---------------------------------------------------------------------

cambiosC :: [Int] -> [(Int,Int)]
cambiosC xs = [(x,y) | (x,y) <- consecutivos (noCeros xs), x*y < 0]
    where consecutivos xs = zip xs (tail xs)

-- (noCeros xs) es la lista de los elementos de xs distintos de cero.
-- Por ejemplo,
--    noCeros [1,3,0,-5,1,-7]  ==  [1,3,-5,1,-7]
noCeros = filter (/=0)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por recursión, la función
--    cambiosR :: [Int] -> [(Int,Int)]
-- tal que (cambiosR xs) es la lista de los pares de elementos de xs con
-- signos distintos, obviando los ceros. Por ejemplo,
--    cambiosR [1,3,0,-5,1,-7]  ==  [(3,-5),(-5,1),(1,-7)]
-- ---------------------------------------------------------------------

cambiosR :: [Int] -> [(Int,Int)]
cambiosR xs = cambiosR' (noCeros xs)
    where cambiosR' (x:y:xs)
              | x*y < 0   = (x,y) : cambiosR' (y:xs)
              | otherwise = cambiosR' (y:xs)
          cambiosR' _ = []

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que las dos definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_cambios :: [Int] -> Bool
prop_cambios xs =
    cambiosC xs == cambiosR xs

-- La comprobación es
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Usando las anteriores definiciones y la regla de
-- Descartes, definir la función
--    nRaicesPositivas :: [Int] -> [Int]
-- tal que (nRaicesPositivas p) es la lista de los posibles números de
-- raíces positivas del polinomio p (representado mediante una lista
-- dispersa) según la regla de los signos de Descartes. Por ejemplo,
--    nRaicesPositivas [1,3,0,-5,1,-7]  ==  [3,1]
-- que significa que la ecuación x^5+3x^4-5x^2+x-7=0 puede tener 3 ó 1
-- raíz positiva.
-- ---------------------------------------------------------------------

nRaicesPositivas :: [Int] -> [Int]
nRaicesPositivas xs = [n,n-2..0]
    where n = length (cambiosC xs)

-- ---------------------------------------------------------------------
-- Nota: El ejercicio 4 está basado en el artículo de Gaussiano "La
-- regla de los signos de Descartes" http://bit.ly/iZXybH
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Se considera la siguiente enumeración de los pares de
-- números naturales
--    (0,0),
--    (0,1), (1,0),
--    (0,2), (1,1), (2,0),
--    (0,3), (1,2), (2,1), (3,0),
--    (0,4), (1,3), (2,2), (3,1), (4,0),
--    (0,5), (1,4), (2,3), (3,2), (4,1), (5,0), ...
--
-- Definir la función
--    siguiente :: (Int,Int) -> (Int,Int)
-- tal que (siguiente (x,y)) es el siguiente del término (x,y) en la
-- enumeración. Por ejemplo,
--    siguiente (2,0)  ==  (0,3)
--    siguiente (0,3)  ==  (1,2)
```

```
--    siguiente (1,2)  ==  (2,1)
-- ---------------------------------------------------------------------

siguiente :: (Int,Int) -> (Int,Int)
siguiente (x,0) = (0,x+1)
siguiente (x,y) = (x+1,y-1)

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la constante
--    enumeracion :: [(Int,Int)]
-- tal que enumeracion es la lista que representa la anterior
-- enumeracion de los pares de números naturales. Por ejemplo,
--    take 6 enumeracion == [(0,0),(0,1),(1,0),(0,2),(1,1),(2,0)]
--    enumeracion !! 9  ==  (3,0)
-- ---------------------------------------------------------------------

enumeracion :: [(Int,Int)]
enumeracion = iterate siguiente (0,0)

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir la función
--       posicion :: (Int,Int) -> Int
--    tal que (posicion p) es la posición del par p en la
--    enumeración. Por ejemplo,
--       posicion (3,0)  ==  9
--       posicion (1,2)  ==  7
-- ---------------------------------------------------------------------

posicion :: (Int,Int) -> Int
posicion (x,y) = length (takeWhile (/= (x,y)) enumeracion)

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Definir la propiedad
--    prop_posicion :: Int -> Bool
-- tal que (prop_posicion n) se verifica si para los n primeros términos
-- (x,y) de la enumeración se cumple que
--    posicion (x,y) == (x+y)*(x+y+1) `div` 2 + x
-- Comprobar si la propiedad se cumple para los 100 primeros elementos.
-- ---------------------------------------------------------------------
```

```haskell
prop_posicion :: Int -> Bool
prop_posicion n =
    and [posicion (x,y) == (x+y)*(x+y+1) `div` 2 + x |
         (x,y) <- take n enumeracion]

-- La comprobación es
--    ghci> prop_posicion 100
--    True
```

1.9. Examen 9 (8 de Julio de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- Examen de la 1ª convocatoria (8 de julio de 2011)
-- ---------------------------------------------------------------------

import Data.Array
import Data.Char
import Test.QuickCheck
import GrafoConVectorDeAdyacencia

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. El enunciado de un problema de las olimpiadas rusas de
-- matemáticas es el siguiente:
--    Si escribimos todos los números enteros empezando por el uno, uno
--    al lado del otro (o sea, 1234567891011121314...), ¿qué dígito
--    ocupa la posición 206788?
-- En los distintos apartados de este ejercicios resolveremos el
-- problema.
--
-- Definir la constante
--    cadenaDeNaturales :: String
-- tal que cadenaDeNaturales es la cadena obtenida escribiendo todos los
-- números enteros empezando por el uno. Por ejemplo,
--    take 19 cadenaDeNaturales  ==  "1234567891011121314"
-- ---------------------------------------------------------------------

cadenaDeNaturales :: String
cadenaDeNaturales = concat [show n | n <- [1..]]

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

```haskell
-- Ejercicio 1.2. Definir la función
--    digito :: Int -> Int
-- tal que (digito n) es el dígito que ocupa la posición n en la cadena
-- de los naturales (el número de las posiciones empieza por 1). Por
-- ejemplo,
--    digito 10  ==  1
--    digito 11  ==  0
-- ---------------------------------------------------------------------

digito :: Int -> Int
digito n = digitToInt (cadenaDeNaturales !! (n-1))

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Calcular el dígito que ocupa la posición 206788 en la
-- cadena de los naturales.
-- ---------------------------------------------------------------------

-- El cálculo es
--   ghci> digito 206788
--   7

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. El problema número 15 de los desafíos matemáticos
-- de El Pais parte de la observación de que todos los números naturales
-- tienen al menos un múltiplo no nulo que está formado solamente por
-- ceros y unos. Por ejemplo, 1x10=10, 2x5=10, 3x37=111, 4x25=100,
-- 5x2=10, 6x185=1110; 7x143=1001; 8X125=1000; 9x12345679=111111111, ...
-- y así para cualquier número natural.
--
-- Definir la constante
--    numerosCon1y0 :: [Integer]
-- tal que numerosCon1y0 es la lista de los números cuyos dígitos son 1
-- ó 0. Por ejemplo,
--    ghci> take 15 numerosCon1y0
--    [1,10,11,100,101,110,111,1000,1001,1010,1011,1100,1101,1110,1111]
-- ---------------------------------------------------------------------

numerosCon1y0 :: [Integer]
numerosCon1y0 = 1 : concat [[10*x,10*x+1] | x <- numerosCon1y0]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    multiplosCon1y0 :: Integer -> [Integer]
-- tal que (multiplosCon1y0 n) es la lista de los múltiplos de n cuyos
-- dígitos son 1 ó 0. Por ejemplo,
--    take 4 (multiplosCon1y0 3)  ==  [111,1011,1101,1110]
-- ---------------------------------------------------------------------

multiplosCon1y0 :: Integer -> [Integer]
multiplosCon1y0 n =
    [x | x <- numerosCon1y0, x `rem` n == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que todo número natural,
-- mayor que 0, tiene múltiplos cuyos dígitos son 1 ó 0.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_existe_multiplosCon1y0 :: Integer -> Property
prop_existe_multiplosCon1y0 n =
    n > 0 ==> multiplosCon1y0 n /= []

-- La comprobación es
--    ghci> quickCheck prop_existe_multiplosCon1y0
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una matriz permutación es una matriz cuadrada con
-- todos sus elementos iguales a 0, excepto uno cualquiera por cada fila
-- y columna, el cual debe ser igual a 1.
--
-- En este ejercicio se usará el tipo de las matrices definido por
--    type Matriz a = Array (Int,Int) a
-- y los siguientes ejemplos de matrices
--    q1, q2, q3 :: Matriz Int
--    q1 = array ((1,1),(2,2)) [((1,1),1),((1,2),0),((2,1),0),((2,2),1)]
--    q2 = array ((1,1),(2,2)) [((1,1),0),((1,2),1),((2,1),0),((2,2),1)]
--    q3 = array ((1,1),(2,2)) [((1,1),3),((1,2),0),((2,1),0),((2,2),1)]
--
-- Definir la función
```

```
--    esMatrizPermutacion :: Num a => Matriz a -> Bool
-- tal que (esMatrizPermutacion p) se verifica si p es una matriz
-- permutación. Por ejemplo.
--    esMatrizPermutacion q1  ==  True
--    esMatrizPermutacion q2  ==  False
--    esMatrizPermutacion q3  ==  False
-- ------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

q1, q2, q3 :: Matriz Int
q1 = array ((1,1),(2,2)) [((1,1),1),((1,2),0),((2,1),0),((2,2),1)]
q2 = array ((1,1),(2,2)) [((1,1),0),((1,2),1),((2,1),0),((2,2),1)]
q3 = array ((1,1),(2,2)) [((1,1),3),((1,2),0),((2,1),0),((2,2),1)]

esMatrizPermutacion :: (Num a, Eq a) => Matriz a -> Bool
esMatrizPermutacion p =
    and [esListaUnitaria [p!(i,j) | i <- [1..n]] | j <- [1..n]] &&
    and [esListaUnitaria [p!(i,j) | j <- [1..n]] | i <- [1..n]]
    where ((1,1),(n,_)) = bounds p

-- (esListaUnitaria xs) se verifica si xs tiene un 1 y los restantes
-- elementos son 0. Por ejemplo,
--    esListaUnitaria [0,1,0,0]  ==  True
--    esListaUnitaria [0,1,0,1]  ==  False
--    esListaUnitaria [0,2,0,0]  ==  False
esListaUnitaria :: (Num a, Eq a) => [a] -> Bool
esListaUnitaria xs =
    [x | x <- xs, x /= 0] == [1]

-- ------------------------------------------------------------------
--  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 False (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 | D 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 5.1. La expansión decimal de un número racional puede
-- representarse mediante una lista cuyo primer elemento es la parte
-- entera y el resto está formado por los dígitos de su parte decimal.
--
-- Definir la función
--    expansionDec :: Integer -> Integer -> [Integer]
```

```haskell
-- tal que (expansionDec x y) es la expansión decimal de x/y. Por
-- ejemplo,
--    take 10 (expansionDec 1 4)    ==  [0,2,5]
--    take 10 (expansionDec 1 7)    ==  [0,1,4,2,8,5,7,1,4,2]
--    take 12 (expansionDec 90 7)   ==  [12,8,5,7,1,4,2,8,5,7,1,4]
--    take 12 (expansionDec 23 14)  ==  [1,6,4,2,8,5,7,1,4,2,8,5]
-- ---------------------------------------------------------------------

expansionDec :: Integer -> Integer -> [Integer]
expansionDec x y
    | r == 0    = [q]
    | otherwise = q : expansionDec (r*10) y
    where (q,r) = quotRem x y

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. La parte decimal de las expansiones decimales se puede
-- dividir en la parte pura y la parte periódica (que es la que se
-- repite). Por ejemplo, puesto que la expansión de 23/14 es
--    [1,6,4,2,8,5,7,1,4,2,8,5,...
-- su parte entera es 1, su parte decimal pura es [6] y su parte decimal
-- periódica es [4,2,8,5,7,1].
--
-- Definir la función
--    formaDecExpDec :: [Integer] -> (Integer,[Integer],[Integer])
-- tal que (formaDecExpDec xs) es la forma decimal de la expresión
-- decimal xs; es decir, la terna formada por la parte entera, la parte
-- decimal pura y la parte decimal periódica. Por ejemplo,
--    formaDecExpDec [3,1,4]               ==  (3,[1,4],[])
--    formaDecExpDec [3,1,4,6,7,5,6,7,5]   ==  (3,[1,4],[6,7,5])
--    formaDecExpDec (expansionDec 23 14)  ==  (1,[6],[4,2,8,5,7,1])
-- ---------------------------------------------------------------------

formaDecExpDec :: [Integer] -> (Integer,[Integer],[Integer])
formaDecExpDec (x:xs) = (x,ys,zs)
    where (ys,zs) = decimales xs

-- (decimales xs) es el par formado por la parte decimal pura y la parte
-- decimal periódica de la lista de decimales xs. Por ejemplo,
--    decimales [3,1,4]            ==  ([3,1,4],[])
--    decimales [3,1,6,7,5,6,7,5]  ==  ([3,1],[6,7,5])
```

```haskell
decimales :: [Integer] -> ([Integer],[Integer])
decimales xs = decimales' xs []
    where decimales' [] ys = (reverse ys, [])
          decimales' (x:xs) ys
              | x `elem` ys = splitAt k ys'
              | otherwise   = decimales' xs (x:ys)
              where ys' = reverse ys
                    k   = posicion x ys'

-- (posicion x ys) es la primera posición de x en la lista ys. Por
-- ejemplo,
--    posicion 2 [0,2,3,2,5]  ==  1
posicion :: Eq a => a -> [a] -> Int
posicion x ys = head [n | (n,y) <- zip [0..] ys, x == y]

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir la función
--    formaDec :: Integer -> Integer -> (Integer,[Integer],[Integer])
-- tal que (formaDec x y) es la forma decimal de x/y; es decir, la terna
-- formada por la parte entera, la parte decimal pura y la parte decimal
-- periódica. Por ejemplo,
--    formaDec 1 4    ==  (0,[2,5],[])
--    formaDec 23 14  ==  (1,[6],[4,2,8,5,7,1])
-- ---------------------------------------------------------------------

formaDec :: Integer -> Integer -> (Integer,[Integer],[Integer])
formaDec x y =
    formaDecExpDec (expansionDec x y)
```

1.10. Examen 10 (16 de Septiembre de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 2ª convocatoria (16 de septiembre de 2011)
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
```

```
--    subsucesiones :: [Integer] -> [[Integer]]
-- tal que (subsucesiones xs) es la lista de las subsucesiones
-- crecientes de elementos consecutivos de xs. Por ejemplo,
--    subsucesiones [1,0,1,2,3,0,4,5]  == [[1],[0,1,2,3],[0,4,5]]
--    subsucesiones [5,6,1,3,2,7]      == [[5,6],[1,3],[2,7]]
--    subsucesiones [2,3,3,4,5]        == [[2,3],[3,4,5]]
--    subsucesiones [7,6,5,4]          == [[7],[6],[5],[4]]
-- ---------------------------------------------------------------------

subsucesiones :: [Integer] -> [[Integer]]
subsucesiones []  = []
subsucesiones [x] = [[x]]
subsucesiones (x:y:zs)
    | x < y     = (x:us):vss
    | otherwise = [x]:p
    where p@(us:vss) = subsucesiones (y:zs)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    menor :: Ord a => [[a]] -> a
-- tal que (menor xss) es el menor elemento común a todas las listas de
-- xss, donde las listas de xss están ordenadas (de menor a mayor) y
-- pueden ser infinitas. Por ejemplo,
--    menor [[3,4,5]]                          ==  3
--    menor [[1,2,3,4,5,6,7],[0.5,3/2,4,19]]   ==  4.0
--    menor [[0..],[4,6..],[2,3,5,7,11,13,28]]  ==  28
-- ---------------------------------------------------------------------

menor :: Ord a => [[a]] -> a
menor (xs:xss) =
    head [x | x <- xs, all (x `pertenece`) xss]

-- (pertenece x ys) se verifica si x pertenece a la lista ys, donde ys
-- es una lista ordenada de menor a mayor y, posiblemente, infinita. Por
-- ejemplo,
--    pertenece 6 [0,2..]  ==  True
--    pertenece 7 [0,2..]  ==  False
pertenece :: Ord a => a -> [a] -> Bool
pertenece x [] = False
pertenece x (y:ys) | x < y  = False
```

```
                  | x == y = True
                  | x > y  = pertenece x ys

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un conjunto A está cerrado respecto de una función  f si
-- para todo elemento x de A se tiene que f(x) pertenece a A. La
-- clausura de un conjunto B respecto de una función f es el menor
-- conjunto A que contiene a B y es cerrado respecto de f. Por ejemplo,
-- la clausura de {0,1,2] respecto del opuesto es {0,1,2,-1,-2}.
--
-- Definir la función
--    clausura :: Eq a => (a -> a) -> [a] -> [a]
-- tal que (clausura f xs) es la clausura de xs respecto de f. Por
-- ejemplo,
--    clausura (\x -> -x) [0,1,2]         ==  [0,1,2,-1,-2]
--    clausura (\x -> (x+1) `mod` 5) [0]  ==  [0,1,2,3,4]
-- ---------------------------------------------------------------------

clausura :: Eq a => (a -> a) -> [a] -> [a]
clausura f xs = clausura' f xs xs
    where clausura' f xs ys | null zs = ys
                            | otherwise = clausura' f zs (ys++zs)
              where zs = nuevosSucesores f xs ys

nuevosSucesores :: Eq a => (a -> a) -> [a] -> [a] -> [a]
nuevosSucesores f xs ys = nub [f x | x <- xs] \\ ys

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El problema del laberinto numérico consiste en, dados
-- un par de números, encontrar la longitud del camino más corto entre
-- ellos usando sólo las siguientes operaciones:
--    * multiplicar por 2,
--    * dividir por 2 (sólo para los pares) y
--    * sumar 2.
-- Por ejemplo,
--    longitudCaminoMinimo 3 12  ==  2
--    longitudCaminoMinimo 12 3  ==  2
--    longitudCaminoMinimo 9 2   ==  8
--    longitudCaminoMinimo 2 9   ==  5
-- Unos caminos mínimos correspondientes a los ejemplos anteriores son
```

```haskell
-- [3,6,12], [12,6,3], [9,18,20,10,12,6,8,4,2] y [2,4,8,16,18,9].
--
-- Definir la función
--    orbita :: Int -> [Int] -> [Int]
-- tal que (orbita n xs) es el conjunto de números que se pueden obtener
-- aplicando como máximo n veces las operaciones a los elementos de
-- xs. Por ejemplo,
--    orbita 0 [12]  ==  [12]
--    orbita 1 [12]  ==  [6,12,14,24]
--    orbita 2 [12]  ==  [3,6,7,8,12,14,16,24,26,28,48]
-- ---------------------------------------------------------------------

orbita :: Int -> [Int] -> [Int]
orbita 0 xs = sort xs
orbita n xs = sort (nub (ys ++ concat [sucesores x | x <- ys]))
    where ys = orbita (n-1) xs
          sucesores x | odd x     = [2*x, x+2]
                      | otherwise = [2*x, x `div` 2, x+2]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    longitudCaminoMinimo :: Int -> Int -> Int
-- tal que (longitudCaminoMinimo x y) es la longitud del camino mínimo
-- desde x hasta y en el laberinto numérico.
--    longitudCaminoMinimo 3 12  ==  2
--    longitudCaminoMinimo 12 3  ==  2
--    longitudCaminoMinimo 9 2   ==  8
--    longitudCaminoMinimo 2 9   ==  5
-- ---------------------------------------------------------------------

longitudCaminoMinimo :: Int -> Int -> Int
longitudCaminoMinimo x y =
    head [n | n <- [1..], y `elem` orbita n [x]]

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. En este ejercicio se estudia las relaciones entre los
-- valores de polinomios y los de sus correspondientes expresiones
-- aritméticas.
--
-- Las expresiones aritméticas construidas con una variables, los
```

```haskell
-- números enteros y las operaciones de sumar y multiplicar se pueden
-- representar mediante el tipo de datos Exp definido por
--    data Exp = Var | Const Int | Sum Exp Exp | Mul Exp  Exp
--               deriving Show
-- Por ejemplo, la expresión 3+5x^2 se puede representar por
--    exp1 :: Exp
--    exp1 = Sum (Const 3) (Mul Var (Mul Var (Const 5)))
--
-- Definir la función
--    valorE :: Exp -> 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
-- ---------------------------------------------------------------------

data Exp = Var | Const Int | Sum Exp Exp | Mul Exp  Exp
           deriving Show

exp1 :: Exp
exp1 = Sum (Const 3) (Mul Var (Mul Var (Const 5)))

valorE :: Exp -> Int -> Int
valorE Var         n = n
valorE (Const a)   n = a
valorE (Sum e1 e2) n = valorE e1 n + valorE e2 n
valorE (Mul e1 e2) n = valorE e1 n * valorE e2 n

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Los polinomios se pueden representar por la lista de
-- sus coeficientes. Por ejemplo, el polinomio 3+5x^2 se puede
-- representar por [3,0,5].
--
-- Definir la función
--    expresion :: [Int] -> Exp
-- tal que (expresion p) es una expresión aritmética equivalente al
-- polinomio p. Por ejemplo,
--    ghci> expresion [3,0,5]
--    Sum (Const 3) (Mul Var (Sum (Const 0) (Mul Var (Const 5))))
-- ---------------------------------------------------------------------
```

```
expresion :: [Int] -> Exp
expresion [a]   = Const a
expresion (a:p) = Sum (Const a) (Mul Var (expresion p))

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir la función
--    valorP :: [Int] -> Int -> Int
-- tal que (valorP p n) es el valor del polinomio p cuando se sustituye
-- su variable por n. Por ejemplo,
--    valorP [3,0,5] 2  ==  23
-- ---------------------------------------------------------------------

valorP :: [Int] -> Int -> Int
valorP [a] _ = a
valorP (a:p) n = a + n * valorP p n

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Comprobar con QuickCheck que, para todo polinomio p y
-- todo entero n,
--    valorP p n == valorE (expresion p) n
-- ---------------------------------------------------------------------

-- La propiedad es
prop_valor :: [Int] -> Int -> Property
prop_valor p n =
    not (null p) ==>
    valorP p n == valorE (expresion p) n

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

1.11. Examen 11 (22 de Noviembre de 2011)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 3ª convocatoria (22 de noviembre de 2011)
-- ---------------------------------------------------------------------

import Data.Array
import Test.QuickCheck
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    barajaC :: [a] -> [a] -> [a]
-- tal que (barajaC xs ys) es la lista obtenida intercalando los
-- elementos de las listas xs e ys. Por ejemplo,
--    barajaC [1,6,2] [3,7]       ==  [1,3,6,7]
--    barajaC [1,6,2] [3,7,4,9,0] ==  [1,3,6,7,2,4]
-- ---------------------------------------------------------------------

barajaC :: [a] -> [a] -> [a]
barajaC xs ys = concat [[x,y] | (x,y) <- zip xs ys]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    barajaR :: [a] -> [a] -> [a]
-- tal que (barajaR xs ys) es la lista obtenida intercalando los
-- elementos de las listas xs e ys. Por ejemplo,
--    barajaR [1,6,2] [3,7]       ==  [1,3,6,7]
--    barajaR [1,6,2] [3,7,4,9,0] ==  [1,3,6,7,2,4]
-- ---------------------------------------------------------------------

barajaR :: [a] -> [a] -> [a]
barajaR (x:xs) (y:ys) = x : y : barajaR xs ys
barajaR _      _      = []

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que la longitud de
-- (barajaR xs y1) es el doble del mínimo de las longitudes de xs es ys.
-- ---------------------------------------------------------------------

prop_baraja :: [Int] -> [Int] -> Bool
prop_baraja xs ys =
    length (barajaC xs ys) == 2 * min (length xs) (length ys)

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

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

```haskell
-- Ejercicio 2.1. Un número n es refactorizable si el número de los
-- divisores de n es un divisor de n.
--
-- Definir la constante
--    refactorizables :: [Int]
-- tal que refactorizables es la lista de los números
-- refactorizables. Por ejemplo,
--    take 10 refactorizables  ==  1,2,8,9,12,18,24,36,40,56]
-- ---------------------------------------------------------------------

refactorizables :: [Int]
refactorizables =
    [n | n <- [1..], length (divisores n) `divide` n]

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

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--    divisores 36  ==  [1,2,3,4,6,9,12,18,36]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `rem` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Un número n es redescompible si el número de
-- descomposiciones de n como producto de dos factores distintos divide
-- a n.
--
-- Definir la función
--    redescompible :: Int -> Bool
-- tal que (redescompible x) se verifica si x es
-- redescompible. Por ejemplo,
--    redescompible 56  ==  True
--    redescompible 57  ==  False
-- ---------------------------------------------------------------------
```

```haskell
redescompible :: Int -> Bool
redescompible n = nDescomposiciones n `divide` n

nDescomposiciones :: Int -> Int
nDescomposiciones n =
    2 * length [(x,y) | x <- [1..n], y <- [x+1..n], x*y == n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función
--    prop_refactorizable :: Int -> Bool
-- tal que (prop_refactorizable n) se verifica si para todo x
-- entre los n primeros números refactorizables se tiene que x es
-- redescompible syss x no es un cuadrado. Por ejemplo,
--    prop_refactorizable 10  ==  True
-- ---------------------------------------------------------------------

prop_refactorizable :: Int -> Bool
prop_refactorizable n =
    and [(nDescomposiciones x `divide` x) == not (esCuadrado x)
         | x <- take n refactorizables]

-- (esCuadrado x) se verifica si  x es un cuadrado perfecto; es decir,
-- si existe un y tal que y^2 es igual a x. Por ejemplo,
--    esCuadrado 16  ==  True
--    esCuadrado 17  ==  False
esCuadrado :: Int -> Bool
esCuadrado x = y^2 == x
    where y = round (sqrt (fromIntegral x))

-- Otra solución, menos eficiente, es
esCuadrado' :: Int -> Bool
esCuadrado' x =
    [y | y <- [1..x], y^2 == x] /= []

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

```haskell
-- 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 = V
--             | N Int ABB ABB
--             deriving (Eq, Show)
-- Por ejemplo, la definición del ABB anteriore es
--    ej :: ABB
--    ej = N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
-- 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 ej
--    N 3 (N 2 V V) (N 6 (N 4 V (N 5 V V)) (N 8 V V))
--    ghci>  inserta 1 ej
--    N 3 (N 2 (N 1 V V) V) (N 6 (N 4 V V) (N 8 V V))
--    ghci>  inserta 2 ej
--    N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
-- ---------------------------------------------------------------------

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

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

inserta :: Int -> ABB -> ABB
inserta v' V = N v' V V
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)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los tipos de los vectores y de las
-- matrices definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
--
-- 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,  si m1 y m2 son las matrices definidas por
--    m1, m2 :: Matriz Int
--    m1 = array ((1,1),(3,3)) [((1,1),7),((1,2),0),((1,3),4),
--                              ((2,1),0),((2,2),6),((2,3),0),
--                              ((3,1),0),((3,2),0),((3,3),5)]
--    m2 = array ((1,1),(3,3)) [((1,1),0),((1,2),0),((1,3),4),
--                              ((2,1),0),((2,2),6),((2,3),0),
--                              ((3,1),0),((3,2),0),((3,3),0)]
-- entonces
--    antidiagonal m1  ==  False
--    antidiagonal m2  ==  True
-- ---------------------------------------------------------------------

type Vector a = Array Int a
type Matriz a = Array (Int,Int) a

m1, m2 :: Matriz Int
m1 = array ((1,1),(3,3)) [((1,1),7),((1,2),0),((1,3),4),
                          ((2,1),0),((2,2),6),((2,3),0),
                          ((3,1),0),((3,2),0),((3,3),5)]
m2 = array ((1,1),(3,3)) [((1,1),0),((1,2),0),((1,3),4),
                          ((2,1),0),((2,2),6),((2,3),0),
                          ((3,1),0),((3,2),0),((3,3),0)]

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) = snd (bounds p)

nula :: (Num a, Eq a) => [a] -> Bool
nula xs = xs == [0 | x <- xs]
```

2

Exámenes del grupo 2

María J. Hidalgo

2.1. Examen 1 (29 de Octubre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 1º examen de evaluación continua (29 de octubre de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. 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 2.1. Definir la función puntoMedio tal que
-- (puntoMedio p1 p2) es el punto medio entre los puntos p1 y p2. Por
-- ejemplo,
--    puntoMedio (0,2) (0,6)  ==  (0.0,4.0)
--    puntoMedio (-1,2) (7,6) ==  (3.0,4.0)
-- ---------------------------------------------------------------------

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con quickCheck que el punto medio entre P y
-- Q equidista de ambos puntos.
-- ---------------------------------------------------------------------

-- El primer intento es
prop_puntoMedio (x1,y1) (x2,y2) =
    distancia (x1,y1) p == distancia (x2,y2) p
    where p = puntoMedio (x1,y1) (x2,y2)

-- (distancia p q) es la distancia del punto p al q. Por ejemplo,
--    distancia (0,0) (3,4)  ==  5.0
distancia (x1,y1) (x2,y2) = sqrt((x1-x2)^2+(y1-y2)^2)

-- La comprobación es
--    ghci> quickCheck prop_puntoMedio
--    *** Failed! Falsifiable (after 13 tests and 5 shrinks):
--    (10.0,-9.69156092012789)
--    (6.0,27.0)

-- Falla, debido a los errores de redondeo. Hay que expresarla en
-- términos de la función ~=.

-- (x ~= y) se verifica si x es aproximadamente igual que y; es decir,
-- el valor absoluto de su diferencia es menor que 0.0001.
x ~= y = abs(x-y) < 0.0001

-- El segundo intento es
prop_puntoMedio2 (x1,y1) (x2,y2) =
    distancia (x1,y1) p ~= distancia (x2,y2) p
    where p = puntoMedio (x1,y1) (x2,y2)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función ciclo tal que (ciclo xs) es
-- permutación de xs obtenida pasando su último elemento a la primera
```

```
-- posición y desplazando los otros elementosPor ejemplo,
--    ciclo [2,5,7,9]         ==  [9,2,5,7]
--    ciclo ["yo","tu","el"]  ==  ["el","yo","tu"]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función numeroMayor tal que
-- (numeroMayor x y) es el mayor número de dos cifras que puede puede
-- construirse con los dígitos x e y. Por ejemplo,
--    numeroMayor 2 5  ==  52
--    numeroMayor 5 2  ==  52
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función numeroMenor tal que tal que
-- (numeroMenor x y) es el menor número de dos cifras que puede puede
-- construirse con los dígitos x e y. Por ejemplo,
--    numeroMenor 2 5  ==  25
--    numeroMenor 5 2  ==  25
-- ---------------------------------------------------------------------

numeroMenor x y = 10*b + a
    where a = max x y
          b = min x y

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que el menor número que puede
-- construirse con dos dígitos es menor o igual que el mayor.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorMayor x y =
  numeroMenor x y <= numeroMayor x y
```

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

2.2. Examen 2 (26 de Noviembre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 2º examen de evaluación continua (26 de noviembre de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    sustituyeImpar :: [Int] -> [Int]
-- tal que (sustituyeImpar x) es la lista obtenida sustituyendo cada
-- número impar de xs por el siguiente número par. Por ejemplo,
--    sustituyeImpar [3,2,5,7,4]  ==  [4,2,6,8,4]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickChek que para cualquier
-- lista de números enteros xs, todos los elementos de la lista
-- (sustituyeImpar xs) son números pares.
-- ---------------------------------------------------------------------

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.1 El número e se puede definir como la suma de la serie
--    1/0! + 1/1! + 1/2! + 1/3! +...
--
-- Definir la función aproxE tal que (aproxE n) es la aproximación de e
-- que se obtiene sumando los términos de la serie hasta 1/n!. Por
-- ejemplo,
--    aproxE 10    ==  2.718281801146385
--    aproxE 100   ==  2.7182818284590455
--    aproxE 1000  ==  2.7182818284590455
-- ---------------------------------------------------------------------

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

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 5  ==  120
factorial n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la constante e como 2.71828459.
-- ---------------------------------------------------------------------

e = 2.71828459

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función errorE tal que (errorE x) es el
-- menor número de términos de la serie anterior necesarios para obtener
-- e con un error menor que x. Por ejemplo,
--    errorE 0.001   == 6.0
--    errorE 0.00001 == 8.0
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Un número natural n se denomina abundante si es menor
-- que la suma de sus divisores propios.
--
-- Definir una función
--    numeroAbundante:: Int -> Bool
```

```haskell
-- tal que (numeroAbundante n) se verifica si n es un número
-- abundante. Por ejemplo,
--    numeroAbundante 5   ==  False
--    numeroAbundante 12  ==  True
--    numeroAbundante 28  ==  False
--    numeroAbundante 30  ==  True
-- ---------------------------------------------------------------------

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

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--    divisores 24  ==  [1,2,3,4,6,8,12]
divisores:: Int -> [Int]
divisores n = [m | m <- [1..n-1], n `mod` m == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    numerosAbundantesMenores :: Int -> [Int]
-- tal que (numerosAbundantesMenores n) es la lista de números
-- abundantes menores o iguales que n. Por ejemplo,
--    numerosAbundantesMenores 50  ==  [12,18,20,24,30,36,40,42,48]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    todosPares :: Int -> Bool
-- tal que (todosPares n) se verifica si todos los números abundantes
-- menores o iguales que n son pares. Comprobar  el valor de dicha
-- función para n = 10, 100 y 1000.
-- ---------------------------------------------------------------------

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

-- La comprobación es
--    ghci> todosPares 10
```

```
--    True
--    ghci> todosPares 100
--    True
--    ghci> todosPares 1000
--    False

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir la constante
--    primerAbundanteImpar :: Int
-- cuyo valor es el primer número natural abundante impar.
-- ---------------------------------------------------------------------

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

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

2.3. Examen 3 (17 de Diciembre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 3º examen de evaluación continua (17 de diciembre de 2010)
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    grafoReducido_1:: (Eq a, Eq b) => (a->b)->(a-> Bool) -> [a] -> [(a,b)]
-- tal que (grafoReducido f p xs) es la lista (sin repeticiones) de los
-- pares formados por los elementos de xs que verifican el predicado p y
-- sus imágenes. Por ejemplo,
--    grafoReducido (^2) even [1..9]  ==  [(2,4),(4,16),(6,36),(8,64)]
--    grafoReducido (+4) even (replicate 40 1) == []
--    grafoReducido (*5) even (replicate 40 2) == [(2,10)]
-- ---------------------------------------------------------------------

-- 1ª definición
```

```haskell
grafoReducido1:: (Eq a, Eq b) => (a->b)->(a-> Bool) -> [a] -> [(a,b)]
grafoReducido1 f p xs = nub (map (\x -> (x,f x)) (filter p xs))

-- 2ª definición
grafoReducido2:: (Eq a, Eq b) => (a->b)->(a-> Bool) -> [a] -> [(a,b)]
grafoReducido2 f p xs = zip as (map f as)
    where as = filter p (nub xs)

-- 3ª definición
grafoReducido3:: (Eq a, Eq b) => (a->b)->(a-> Bool) -> [a] -> [(a,b)]
grafoReducido3 f p xs = nub [(x,f x) | x <- xs, p x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Un número natural n se denomina semiperfecto si es la
-- suma de algunos de sus divisores propios. Por ejemplo, 18 es
-- semiperfecto ya que sus divisores son 1, 2, 3, 6, 9 y se cumple que
-- 3+6+9=18.
--
-- Definir la función
--    esSemiPerfecto:: Int -> Bool
-- tal que (esSemiPerfecto n) se verifica si n es semiperfecto. Por
-- ejemplo,
--    esSemiPerfecto 18 == True
--    esSemiPerfecto 9  == False
--    esSemiPerfecto 24 == True
-- ---------------------------------------------------------------------

-- 1ª solución:
esSemiPerfecto:: Int -> Bool
esSemiPerfecto n = any p (sublistas (divisores n))
    where p xs = sum xs == n

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

-- (sublistas xs) es la lista de las sublistas de xs. por ejemplo,
--    sublistas [3,2,5]  ==  [[],[5],[2],[2,5],[3],[3,5],[3,2],[3,2,5]]
sublistas :: [a] -> [[a]]
```

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

-- 2ª solución:
esSemiPerfecto2:: Int -> Bool
esSemiPerfecto2 n = or [sum xs == n  | xs <- sublistas (divisores n)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la constante
--    primerSemiPerfecto :: Int
-- tal que su valor es el primer número semiperfecto.
-- ---------------------------------------------------------------------

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

-- Su cálculo es
--    ghci> primerSemiPerfecto
--    6

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función
--    semiPerfecto :: Int -> Int
-- tal que (semiPerfecto n) es el n-ésimo número semiperfecto. Por
-- ejemplo,
--    semiPerfecto 1   == 6
--    semiPerfecto 4   == 20
--    semiPerfecto 100 == 414
-- ---------------------------------------------------------------------

semiPerfecto :: Int -> Int
semiPerfecto n = [n| n <- [1..], esSemiPerfecto n] !! (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir mediante plegado la función
--    producto :: Num a => [a] -> a
-- tal que (producto xs) es el producto de los elementos de la lista
-- xs. Por ejemplo,
--    producto [2,1,-3,4,5,-6] == 720
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir mediante plegado la función
--    productoPred :: Num a => (a -> Bool) -> [a] -> a
-- tal que (productoPred p xs) es el producto de los elementos de la
-- lista xs que verifican el predicado p. Por ejemplo,
--    productoPred even [2,1,-3,4,5,-6] == -48
-- ---------------------------------------------------------------------

productoPred :: Num a => (a -> Bool) -> [a] -> a
productoPred p = foldr f 1
    where f x y | p x       = x*y
                | otherwise = y

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función la función
--    productoPos :: (Num a, Ord a) => [a] -> a
-- tal que (productoPos xs) es el producto de los elementos
-- estrictamente positivos de la lista xs. Por ejemplo,
--    productoPos [2,1,-3,4,5,-6] == 40
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las relaciones finitas se pueden representar mediante
-- listas de pares. Por ejemplo,
--    r1, r2, r3 :: [(Int, Int)]
--    r1 = [(1,3), (2,6), (8,9), (2,7)]
--    r2 = [(1,3), (2,6), (8,9), (3,7)]
--    r3 = [(1,3), (2,6), (8,9), (3,6)]
--
-- Definir la función
--    esFuncion :: [(Int,Int)] -> Bool
-- tal que (esFuncion r) se verifica si la relación r es una función (es
```

```haskell
-- decir, a cada elemento del dominio de la relación r le corresponde un
-- único elemento). Por ejemplo,
--    esFuncion r1 == False
--    esFuncion r2 == True
--    esFuncion r3 == True
-- ---------------------------------------------------------------------

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

-- 1ª definición:
esFuncion :: [(Int,Int)] -> Bool
esFuncion r = and [length (imagenes x r) == 1 | x <- dominio r]

-- (dominio r) es el dominio de la relación r. Por ejemplo,
--    dominio r1  ==  [1,2,8]
dominio :: [(Int, Int)] -> [Int]
dominio r = nub [x | (x,_) <- r]

-- (imagenes x r) es la lista de las imágenes de x en la relación r. Por
-- ejemplo,
--    imagenes 2 r1  ==  [6,7]
imagenes :: Int -> [(Int, Int)] -> [Int]
imagenes x r = nub [y | (z,y) <- r, z == x]

-- 2ª definición:
esFuncion2 :: (Eq a, Eq b) => [(a, b)] -> Bool
esFuncion2 r = [fst x | x <- nub r] == nub [fst x | x <- nub r]

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

```haskell
--    colas []        == []
--    colas [1,2]     == [[1,2],[2]]
--    colas [4,1,2,5] == [[4,1,2,5],[1,2,5],[2,5],[5]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Se denomina cabeza de una lista xs a una sublista no
-- vacía de xs formada por el primer elemento y los siguientes hasta uno
-- dado. Por ejemplo, [1,2,3] es una cabeza de [1,2,3,4,5].
--
-- Definir, por recursión, la función
--    cabezasR :: [a] -> [[a]]
-- tal que (cabezasR xs) es la lista de las cabezas de la lista xs. Por
-- ejemplo,
--    cabezasR []          == []
--    cabezasR [1,4]       == [[1],[1,4]]
--    cabezasR [1,4,5,2,3] == [[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-- ---------------------------------------------------------------------

cabezasR :: [a] -> [[a]]
cabezasR []     = []
cabezasR (x:xs) = [x] : [x:ys | ys <- cabezasR xs]

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir, por plegado, la función
--    cabezasP :: [a] -> [[a]]
-- tal que (cabezasP xs) es la lista de las cabezas de la lista xs. Por
-- ejemplo,
--    cabezasP []          == []
--    cabezasP [1,4]       == [[1],[1,4]]
--    cabezasP [1,4,5,2,3] == [[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-- ---------------------------------------------------------------------

cabezasP :: [a] -> [[a]]
cabezasP = foldr (\x ys -> [x] : [x:y | y <- ys]) []
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir, por composición, la función
--    cabezasC :: [a] -> [[a]]
-- tal que (cabezasC xs) es la lista de las cabezas de la lista xs. Por
-- ejemplo,
--    cabezasC []          == []
--    cabezasC [1,4]       == [[1],[1,4]]
--    cabezasC [1,4,5,2,3] == [[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-- ---------------------------------------------------------------------

cabezasC :: [a] -> [[a]]
cabezasC = reverse . map reverse . colas . reverse
```

2.4. Examen 4 (11 de Febrero de 2011)

El examen es común con el del grupo 1 (ver página 14).

2.5. Examen 5 (14 de Marzo de 2011)

El examen es común con el del grupo 1 (ver página 16).

2.6. Examen 6 (15 de abril de 2011)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 6º examen de evaluación continua (15 de abril de 2011)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import Test.QuickCheck
import Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    mayor :: Ord a => Monticulo a -> a
-- tal que (mayor m) es el mayor elemento del montículo m. Por ejemplo,
--    mayor (foldr inserta vacio [6,8,4,1])  ==  8
-- ---------------------------------------------------------------------
```

```haskell
mayor :: Ord a => Monticulo a -> a
mayor m | esVacio m = error "mayor: monticulo vacio"
        | otherwise = aux m (menor m)
        where aux m k | esVacio m = k
                      | otherwise = aux (resto m) (max k (menor m))

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    minMax :: Ord a => Monticulo a -> Maybe (a,a)
-- tal que (minMax m) es un par con el menor y el mayor elemento de m
-- si el montículo no es vacío. Por ejemplo,
--    minMax (foldr inserta vacio [6,1,4,8])  ==  Just (1,8)
--    minMax (foldr inserta vacio [6,8,4,1])  ==  Just (1,8)
--    minMax (foldr inserta vacio [7,5])      ==  Just (5,7)
-- ---------------------------------------------------------------------

minMax :: (Ord a) => Monticulo a -> Maybe (a,a)
minMax m | esVacio m = Nothing
         | otherwise = Just (menor m, mayor m)


-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Consideremos el siguiente tipo de dato
--    data Arbol a = H a | N (Arbol a) a (Arbol a)
-- y el siguiente ejemplo,
--    ejArbol :: Arbol Int
--    ejArbol = N (N (H 1) 3 (H 4)) 5 (N (H 6) 7 (H 9))
--
-- Definir la función
--     arbolMonticulo:: Ord t => Arbol t -> Monticulo t
-- tal que (arbolMonticulo a) es el montículo formado por los elementos
-- del árbol a. Por ejemplo,
--    ghci> arbolMonticulo ejArbol
--    M 1 2 (M 4 1 (M 6 2 (M 9 1 Vacio Vacio) (M 7 1 Vacio Vacio)) Vacio)
--          (M 3 1 (M 5 1 Vacio Vacio) Vacio)
-- ---------------------------------------------------------------------

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

```haskell
ejArbol :: Arbol Int
ejArbol = N (N (H 1) 3 (H 4)) 5 (N (H 6) 7 (H 9))

arbolMonticulo:: Ord t => Arbol t -> Monticulo t
arbolMonticulo = lista2Monticulo . arbol2Lista

-- (arbol2Lista a) es la lista de los valores del árbol a. Por ejemplo,
--    arbol2Lista ejArbol  ==  [5,3,1,4,7,6,9]
arbol2Lista :: Arbol t -> [t]
arbol2Lista (H x)     = [x]
arbol2Lista (N i x d) = x : (arbol2Lista i ++ arbol2Lista d)

-- (lista2Monticulo xs) es el montículo correspondiente a la lista
-- xs. Por ejemplo,
--    ghci> lista2Monticulo [5,3,4,7]
--    M 3 2 (M 4 1 (M 7 1 Vacio Vacio) Vacio) (M 5 1 Vacio Vacio)
lista2Monticulo :: Ord t => [t] -> Monticulo t
lista2Monticulo = foldr inserta vacio

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    minArbol :: Ord t => Arbol t -> t
-- tal que (minArbol a) es el menor elemento de a. Por ejemplo,
--    minArbol ejArbol == 1
-- ---------------------------------------------------------------------

minArbol :: Ord t => Arbol t -> t
minArbol = menor . arbolMonticulo

-- ---------------------------------------------------------------------
-- Ejercicio 3.1.  Consideremos los tipos de los vectores y las matrices
-- definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
-- y los siguientes ejemplos
--    p1, p2, p3 :: Matriz Double
--    p1 = listArray ((1,1),(3,3)) [1.0,2,3,1,2,4,1,2,5]
--    p2 = listArray ((1,1),(3,3)) [1.0,2,3,1,3,4,1,2,5]
--    p3 = listArray ((1,1),(3,3)) [1.0,2,1,0,4,7,0,0,5]
--
```

```haskell
-- Definir la función
--    esTriangularS :: Num a => Matriz a -> Bool
-- tal que (esTriangularS p) se verifica si p es una matriz triangular
-- superior. Por ejemplo,
--    esTriangularS p1 == False
--    esTriangularS p3 == True
-- ---------------------------------------------------------------------

type Vector a = Array Int a

type Matriz a = Array (Int,Int) a

p1, p2, p3:: Matriz Double
p1 = listArray ((1,1),(3,3)) [1.0,2,3,1,2,4,1,2,5]
p2 = listArray ((1,1),(3,3)) [1.0,2,3,1,3,4,1,2,5]
p3 = listArray ((1,1),(3,3)) [1.0,2,1,0,4,7,0,0,5]

esTriangularS:: Num 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 3.2. 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. Por ejemplo,
--    ghci> submatriz 2 3 (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
--    array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),5),((2,2),4)]
--    ghci> submatriz 2 3 (listArray ((1,1),(3,3)) [1..9])
--    array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),7),((2,2),8)]
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 4.1. El número 22940075 tiene una curiosa propiedad. Si lo
-- factorizamos, obtenemos 22940075 = 5^2×229×4007. Reordenando y
-- concatenando los factores primos (5, 229, 40079 podemos obtener el
-- número original: 22940075.
--
-- Diremos que un número es especial si tiene esta propiedad.
--
-- Definir la función
--    esEspecial :: Integer -> Bool
-- tal que (esEspecial n) se verifica si n es un número especial. Por
-- ejemplo,
--    esEspecial 22940075  ==  True
--    esEspecial 22940076  ==  False
-- ---------------------------------------------------------------------

esEspecial :: Integer -> Bool
esEspecial n =
    sort (concat (map cifras (nub (factorizacion n)))) == sort (cifras n)

-- (factorizacion n) es la lista de los factores de n. Por ejemplo,
--    factorizacion 22940075  ==  [5,5,229,4007]
```

```haskell
factorizacion :: Integer -> [Integer]
factorizacion n | n == 1    = []
                | otherwise = x : factorizacion (div n x)
    where x = menorFactor n

-- (menorFactor n) es el menor factor primo de n. Por ejemplo,
--    menorFactor 22940075  ==  5
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 22940075  ==  [2,2,9,4,0,0,7,5]
cifras :: Integer -> [Integer]
cifras n = [read [x] | x <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que todos los números primos
-- son especiales.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_Especial :: Integer -> Property
prop_Especial n =
    esPrimo m ==> esEspecial m
    where m = abs n

-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 9  ==  False
esPrimo :: Integer -> Bool
esPrimo x = [y | y <- [1..x], x `rem` y == 0] == [1,x]
```

2.7. Examen 7 (27 de mayo de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 7º examen de evaluación continua (27 de mayo de 2011)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
```

```haskell
import Test.QuickCheck
-- import GrafoConMatrizDeAdyacencia
import GrafoConVectorDeAdyacencia

-- ---------------------------------------------------------------------
-- Ejercicio 1. En los distintos apartados de este ejercicio
-- consideraremos relaciones binarias, representadas mediante una lista
-- de pares. Para ello, definimos el tipo de las relaciones binarias
-- sobre el tipo a.
--    type RB a = [(a,a)]
-- Usaremos los siguientes ejemplos de relaciones
--    r1, r2, r3 :: RB Int
--    r1 = [(1,3),(3,1), (1,1), (3,3)]
--    r2 = [(1,3),(3,1)]
--    r3 = [(1,2),(1,4),(3,3),(2,1),(4,2)]
--
--  Definir la función
--    universo :: Eq a => RB a -> [a]
-- tal que (universo r) es la lista de elementos de la relación r. Por
-- ejemplo,
--     universo r1 == [1,3]
--     universo r3 == [1,2,3,4]
-- ---------------------------------------------------------------------

type RB a = [(a,a)]

r1, r2, r3 :: RB Int
r1 = [(1,3),(3,1), (1,1), (3,3)]
r2 = [(1,3),(3,1)]
r3 = [(1,2),(1,4),(3,3),(2,1),(4,2)]

-- 1ª definición:
universo :: Eq a => RB a -> [a]
universo r = nub (l1 ++ l2)
    where l1 = map fst r
          l2 = map snd r

-- 2ª definición:
universo2 :: Eq a => RB a -> [a]
universo2 r = nub (concat [[x,y] | (x,y) <- r])
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    reflexiva:: RB Int -> Bool
-- tal que (reflexiva r) se verifica si r es una relación reflexiva en
-- su universo. Por ejemplo,
--    reflexiva r1 == True
--    reflexiva r2 == False
-- ---------------------------------------------------------------------

reflexiva:: RB Int -> Bool
reflexiva r = and [(x,x) `elem` r | x <- universo r]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Dadas dos relaciones binarias R y S, la composición es
-- la relación R o S = {(a,c) | existe b tal que aRb y bRc}.
--
-- Definir la función
--    compRB:: RB Int -> RB Int -> RB Int
-- tal que (compRB r1 r2) es la composición de las relaciones r1 y r2.
-- Por ejemplo,
--   compRB r1 r3 == [(1,3),(3,2),(3,4),(1,2),(1,4),(3,3)]
--   compRB r3 r1 == [(3,1),(3,3),(2,3),(2,1)]
-- ---------------------------------------------------------------------

-- 1ª definición:
compRB:: RB Int -> RB Int -> RB Int
compRB r s = [(x,z) | (x,y) <- r, (y',z) <- s, y == y']

-- 2ª definición:
compRB2:: RB Int -> RB Int -> RB Int
compRB2 [] _        = []
compRB2 ((x,y):r) s = compPar (x,y) s ++ compRB2 r s

-- (compPar p r) es la relación obtenida componiendo el par p con la
-- relación binaria r. Por ejemplo,
--    compPar (5,1) r1  ==  [(5,3),(5,1)]
compPar:: (Int,Int) -> RB Int -> RB Int
compPar _ []            = []
compPar (x,y) ((z,t):r) | y == z    = (x,t) : compPar (x,y) r
```

```
                        | otherwise = compPar (x,y) r

-- 3ª definición:
compRB3:: RB Int -> RB Int -> RB Int
compRB3 r1 r2 = [(x,z) | x <- universo r1, z <- universo r2,
                         interRelacionados x z r1 r2]

-- (interRelacionados x z r s) se verifica si existe un y tal que (x,y)
-- está en r e (y,z) está en s. Por ejemplo.
--    interRelacionados 3 4 r1 r3  ==  True
interRelacionados :: Int -> Int -> RB Int -> RB Int -> Bool
interRelacionados x z r s =
    not (null [y | y<-universo r, (x,y) p `elem` r, (y,z) `elem` s])

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
--    transitiva:: RB Int -> Bool
-- tal que (transitiva r) se verifica si r es una relación
-- transitiva. Por ejemplo,
--    transitiva r1 == True
--    transitiva r2 == False
-- ---------------------------------------------------------------------

-- 1ª solución:
transitiva :: RB Int -> Bool
transitiva r = and [(x,z) `elem` r | (x,y) <- r, (y',z) <- r, y == y']

-- 2ª solución:
transitiva2 :: RB Int -> Bool
transitiva2 [] = True
transitiva2 r  = and [trans par r | par <- r]
    where trans (x,y) r = and [(x,v) `elem` r | (u,v) <- r, u == y ]

-- 3ª solución (usando la composición de relaciones):
transitiva3 :: RB Int -> Bool
transitiva3 r = contenida r (compRB r r)
    where contenida [] _      = True
          contenida (x:xs) ys = elem x ys && contenida  xs ys

-- 4ª solución:
```

```haskell
transitiva4 :: RB Int -> Bool
transitiva4 = not . noTransitiva

-- (noTransitiva r) se verifica si r no es transitiva; es decir, si
-- existe un (x,y), (y,z) en r tales que (x,z) no está en r.
noTransitiva :: RB Int -> Bool
noTransitiva r =
    not (null [(x,y,z) | (x,y,z) <- ls,
                         (x,y) `elem` r , (y,z) `elem` r,
                         (x,z) `notElem` r])
    where l = universo r
          ls = [(x,y,z) | x <- l, y <- l, z <- l, x/=y, y /= z]

-- ---------------------------------------------------------------------
-- 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
--    g :: Grafo Int Int
--    g = 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
-- ---------------------------------------------------------------------

g :: Grafo Int Int
g = 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
```

```haskell
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 3.1. Se dice que una matriz es booleana si sus elementos
-- son los valores booleanos: True, False.
--
-- Definir la función
--    sumaB :: Bool -> Bool -> Bool
-- tal que (sumaB x y) es falso si y sólo si ambos argumentos son
-- falsos.
-- ---------------------------------------------------------------------
```

```haskell
sumaB :: Bool -> Bool -> Bool
sumaB = (||)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    prodB :: Bool -> Bool -> Bool
-- tal que (prodB x y) es verdadero si y sólo si ambos argumentos son
-- verdaderos.
-- ---------------------------------------------------------------------

prodB :: Bool -> Bool -> Bool
prodB = (&&)

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. En los siguientes apartados usaremos los tipos
-- definidos a continuación:
-- * 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
-- En los ejemplos se usarán las siguientes matrices:
--    m1, m2 :: Matriz Bool
--    m1 = array ((1,1),(3,3)) [((1,1),True), ((1,2),False),((1,3),True),
--                              ((2,1),False),((2,2),False),((2,3),False),
--                              ((3,1),True), ((3,2),False),((3,3),True)]
--    m2 = array ((1,1),(3,3)) [((1,1),False),((1,2),False),((1,3),True),
--                              ((2,1),False),((2,2),False),((2,3),False),
--                              ((3,1),True), ((3,2),False),((3,3),False)]
--
-- También se usan las siguientes funciones definidas en las relaciones
-- de ejercicios.
--    numFilas :: Matriz a -> Int
--    numFilas = fst . snd . bounds
--
--    numColumnas:: Matriz a -> Int
--    numColumnas = snd . snd . bounds
--
--    filaMat :: Int -> Matriz a -> Vector a
```

```
--    filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
--        where n = numColumnas p
--
--    columnaMat :: Int -> Matriz a -> Vector a
--    columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]
--        where m = numFilas p
--
-- Definir la función
--    prodMatricesB :: Matriz Bool -> Matriz Bool -> Matriz Bool
-- tal que (prodMatricesB p q) es el producto de las matrices booleanas
-- p y q, usando la suma y el producto de booleanos, definidos
-- previamente. Por ejemplo,
--    ghci> prodMatricesB m1 m2
--    array ((1,1),(3,3)) [((1,1),True), ((1,2),False),((1,3),True),
--                         ((2,1),False),((2,2),False),((2,3),False),
--                         ((3,1),True), ((3,2),False),((3,3),True)]
-- ----------------------------------------------------------------------

type Vector a = Array Int a

type Matriz a = Array (Int,Int) a

m1, m2 :: Matriz Bool
m1 = array ((1,1),(3,3)) [((1,1),True), ((1,2),False),((1,3),True),
                          ((2,1),False),((2,2),False),((2,3),False),
                          ((3,1),True), ((3,2),False),((3,3),True)]
m2 = array ((1,1),(3,3)) [((1,1),False),((1,2),False),((1,3),True),
                          ((2,1),False),((2,2),False),((2,3),False),
                          ((3,1),True), ((3,2),False),((3,3),False)]

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

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

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

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

prodMatricesB:: Matriz Bool -> Matriz Bool -> Matriz Bool
prodMatricesB p q =
    array ((1,1),(m,n))
          [((i,j), prodEscalarB (filaMat i p) (columnaMat j q)) |
           i <- [1..m], j <- [1..n]]
    where m = numFilas p
          n = numColumnas q

-- (prodEscalarB v1 v2) es el producto escalar booleano de los vectores
-- v1 y v2.
prodEscalarB :: Vector Bool -> Vector Bool -> Bool
prodEscalarB v1 v2 =
    sumB [prodB i j | (i,j) <- zip (elems v1) (elems v2)]
    where sumB = foldr sumaB False

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Se considera la siguiente relación de orden entre
-- matrices: p es menor o igual que q si para toda posición (i,j), el
-- elemento de p en (i,j) es menor o igual que el elemento de q en la
-- posición (i,j). Definir la función
--    menorMatricesB :: Ord a => Matriz a -> Matriz a -> Bool
-- tal que (menorMatricesB p q) se verifica si p es menor o igual que
-- q.
--    menorMatricesB m1 m2  ==  False
--    menorMatricesB m2 m1  ==  True
-- ---------------------------------------------------------------------

menorMatricesB :: Ord a => Matriz a -> Matriz a -> Bool
menorMatricesB p q =
    and [p!(i,j) <= q!(i,j) | i <- [1..m], j <-[1..n]]
        where m = numFilas p
              n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Dada una relación r sobre un conjunto de números
-- naturales mayores que 0, la matriz asociada a r es una matriz
```

```
-- booleana p, tal que p_ij = True si y sólo si i está relacionado con j
-- mediante la relación r. Definir la función
--    matrizRB:: RB Int -> Matriz Bool
-- tal que (matrizRB r) es la matriz booleana asociada a r. Por ejemplo,
--    ghci> matrizRB r1
--    array ((1,1),(3,3)) [((1,1),True),((1,2),False),((1,3),True),
--                         ((2,1),False),((2,2),False),((2,3),False),
--                         ((3,1),True),((3,2),False),((3,3),True)]
--    ghci> matrizRB r2
--    array ((1,1),(3,3)) [((1,1),False),((1,2),False),((1,3),True),
--                         ((2,1),False),((2,2),False),((2,3),False),
--                         ((3,1),True),((3,2),False),((3,3),False)]
--
-- Nota: Construir una matriz booleana cuadrada, de dimensión nxn,
-- siendo n el máximo de los elementos del universo de r.
-- ----------------------------------------------------------------------

matrizRB:: RB Int -> Matriz Bool
matrizRB r = array ((1,1),(n,n)) [((i,j),f (i,j)) | i <- [1..n],j<-[1..n]]
    where n       = maximum (universo r)
          f (i,j) = (i,j) `elem` r

-- ----------------------------------------------------------------------
-- Ejercicio 3.5. Se verifica la siguiente propiedad: r es una relación
-- transitiva si y sólo si M^2 <= M, siendo M la matriz booleana
-- asoociada a r, y M^2 el resultado de multiplicar M por M mediante
-- el producto booleano. Definir la función
--    transitivaB :: RB Int -> Bool
-- tal que (transitivaB r) se verifica si r es una relación
-- transitiva. Por ejemplo,
--    transitivaB r1 == True
--    transitivaB r2 == False
-- ----------------------------------------------------------------------

transitivaB :: RB Int -> Bool
transitivaB r =  menorMatricesB q p
    where p = matrizRB r
          q = prodMatricesB p p
```

2.8. Examen 8 (24 de Junio de 2011)

El examen es común con el del grupo 1 (ver página 27).

2.9. Examen 9 (8 de Julio de 2011)

El examen es común con el del grupo 1 (ver página 34).

2.10. Examen 10 (16 de Septiembre de 2011)

El examen es común con el del grupo 1 (ver página 40).

2.11. Examen 11 (22 de Noviembre de 2011)

El examen es común con el del grupo 1 (ver página 51).

Apéndice A

Resumen de funciones predefinidas de Haskell

1. `x + y` es la suma de x e y.
2. `x - y` es la resta de x e y.
3. `x / y` es el cociente de x entre y.
4. `x ^ y` es x elevado a y.
5. `x == y` se verifica si x es igual a y.
6. `x /= y` se verifica si x es distinto de y.
7. `x < y` se verifica si x es menor que y.
8. `x <= y` se verifica si x es menor o igual que y.
9. `x > y` se verifica si x es mayor que y.
10. `x >= y` se verifica si x es mayor o igual que y.
11. `x && y` es la conjunción de x e y.
12. `x || y` es la disyunción de x e y.
13. `x:ys` es la lista obtenida añadiendo x al principio de ys.
14. `xs ++ ys` es la concatenación de xs e ys.
15. `xs !! n` es el elemento n–ésimo de xs.
16. `f . g` es la composición de f y g.
17. `abs x` es el valor absoluto de x.
18. `and xs` es la conjunción de la lista de booleanos xs.
19. `ceiling x` es el menor entero no menor que x.
20. `chr n` es el carácter cuyo código ASCII es n.
21. `concat xss` es la concatenación de la lista de listas xss.
22. `const x y` es x.

23. `curry f` es la versión curryficada de la función `f`.
24. `div x y` es la división entera de `x` entre `y`.
25. `drop n xs` borra los `n` primeros elementos de `xs`.
26. `dropWhile p xs` borra el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
27. `elem x ys` se verifica si `x` pertenece a `ys`.
28. `even x` se verifica si `x` es par.
29. `filter p xs` es la lista de elementos de la lista `xs` que verifican el predicado `p`.
30. `flip f x y` es `f y x`.
31. `floor x` es el mayor entero no mayor que `x`.
32. `foldl f e xs` pliega `xs` de izquierda a derecha usando el operador f y el valor inicial `e`.
33. `foldr f e xs` pliega `xs` de derecha a izquierda usando el operador f y el valor inicial `e`.
34. `fromIntegral x` transforma el número entero `x` al tipo numérico correspondiente.
35. `fst p` es el primer elemento del par `p`.
36. `gcd x y` es el máximo común divisor de de `x` e `y`.
37. `head xs` es el primer elemento de la lista `xs`.
38. `init xs` es la lista obtenida eliminando el último elemento de `xs`.
39. `iterate f x` es la lista `[x, f(x), f(f(x)), ...]`.
40. `last xs` es el último elemento de la lista `xs`.
41. `length xs` es el número de elementos de la lista `xs`.
42. `map f xs` es la lista obtenida aplicado `f` a cada elemento de `xs`.
43. `max x y` es el máximo de `x` e `y`.
44. `maximum xs` es el máximo elemento de la lista `xs`.
45. `min x y` es el mínimo de `x` e `y`.
46. `minimum xs` es el mínimo elemento de la lista `xs`.
47. `mod x y` es el resto de `x` entre `y`.
48. `not x` es la negación lógica del booleano `x`.
49. `noElem x ys` se verifica si `x` no pertenece a `ys`.
50. `null xs` se verifica si `xs` es la lista vacía.
51. `odd x` se verifica si `x` es impar.
52. `or xs` es la disyunción de la lista de booleanos `xs`.
53. `ord c` es el código ASCII del carácter `c`.

54. `product xs` es el producto de la lista de números `xs`.
55. `read c` es la expresión representada por la cadena `c`.
56. `rem x y` es el resto de `x` entre `y`.
57. `repeat x` es la lista infinita `[x, x, x, ...]`.
58. `replicate n x` es la lista formada por `n` veces el elemento `x`.
59. `reverse xs` es la inversa de la lista `xs`.
60. `round x` es el redondeo de `x` al entero más cercano.
61. `scanr f e xs` es la lista de los resultados de plegar `xs` por la derecha con `f` y `e`.
62. `show x` es la representación de `x` como cadena.
63. `signum x` es 1 si `x` es positivo, 0 si `x` es cero y -1 si `x` es negativo.
64. `snd p` es el segundo elemento del par `p`.
65. `splitAt n xs` es `(take n xs, drop n xs)`.
66. `sqrt x` es la raíz cuadrada de `x`.
67. `sum xs` es la suma de la lista numérica `xs`.
68. `tail xs` es la lista obtenida eliminando el primer elemento de `xs`.
69. `take n xs` es la lista de los `n` primeros elementos de `xs`.
70. `takeWhile p xs` es el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
71. `uncurry f` es la versión cartesiana de la función `f`.
72. `until p f x` aplica `f` a `x` hasta que se verifique `p`.
73. `zip xs ys` es la lista de pares formado por los correspondientes elementos de `xs` e `ys`.
74. `zipWith f xs ys` se obtiene aplicando `f` a los correspondientes elementos de `xs` e `ys`.

A.1. Resumen de funciones sobre TAD en Haskell

A.1.1. Polinomios

1. `polCero` es el polinomio cero.
2. `(esPolCero p)` se verifica si `p` es el polinomio cero.
3. `(consPol n b p)` es el polinomio $bx^n + p$.
4. `(grado p)` es el grado del polinomio `p`.

5. `(coefLider p)` es el coeficiente líder del polinomio `p`.
6. `(restoPol p)` es el resto del polinomio `p`.

A.1.2. Vectores y matrices (`Data.Array`)

1. `(range m n)` es la lista de los índices del `m` al `n`.
2. `(index (m,n) i)` es el ordinal del índice `i` en `(m,n)`.
3. `(inRange (m,n) i)` se verifica si el índice `i` está dentro del rango limitado por `m` y `n`.
4. `(rangeSize (m,n))` es el número de elementos en el rango limitado por `m` y `n`.
5. `(array (1,n) [(i, f i) | i <- [1..n])` es el vector de dimensión `n` cuyo elemento `i`–ésimo es `f i`.
6. `(array ((1,1),(m,n)) [((i,j), f i j) | i <- [1..m], j <- [1..n]])` es la matriz de dimensión `m.n` cuyo elemento `(i,j)`–ésimo es `f i j`.
7. `(array (m,n) ivs)` es la tabla de índices en el rango limitado por `m` y `n` definida por la lista de asociación `ivs` (cuyos elementos son pares de la forma (índice, valor)).
8. `(t ! i)` es el valor del índice `i` en la tabla `t`.
9. `(bounds t)` es el rango de la tabla `t`.
10. `(indices t)` es la lista de los índices de la tabla `t`.
11. `(elems t)` es la lista de los elementos de la tabla `t`.
12. `(assocs t)` es la lista de asociaciones de la tabla `t`.
13. `(t // ivs)` es la tabla `t` asignándole a los índices de la lista de asociación `ivs` sus correspondientes valores.
14. `(listArray (m,n) vs)` es la tabla cuyo rango es `(m,n)` y cuya lista de valores es `vs`.
15. `(accumArray f v (m,n) ivs)` es la tabla de rango `(m,n)` tal que el valor del índice `i` se obtiene acumulando la aplicación de la función `f` al valor inicial `v` y a los valores de la lista de asociación `ivs` cuyo índice es `i`.

A.1.3. Tablas

1. `(tabla ivs)` es la tabla correspondiente a la lista de asociación `ivs` (que es una lista de pares formados por los índices y los valores).
2. `(valor t i)` es el valor del índice `i` en la tabla `t`.
3. `(modifica (i,v) t)` es la tabla obtenida modificando en la tabla `t` el valor de `i` por `v`.

A.1.4. Grafos

1. `(creaGrafo d cs as)` es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso).
2. `(dirigido g)` se verifica si `g` es dirigido.
3. `(nodos g)` es la lista de todos los nodos del grafo `g`.
4. `(aristas g)` es la lista de las aristas del grafo `g`.
5. `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`.
6. `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`.
7. `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo `g`.

Apéndice B

Método de Pólya para la resolución de problemas

B.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?

- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejercutar tu plan de la solución, comprueba cada uno de los pasos
- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya “Cómo plantear y resolver problemas” (Ed. Trillas, 1978) p. 19

B.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?
- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.
- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson *How to program it*, basado en G. Polya *Cómo plantear y resolver problemas*.

Bibliografía

[1] J. A. Alonso and M. J. Hidalgo. Piensa en Haskell (Ejercicios de programación funcional con Haskell). Technical report, Univ. de Sevilla, 2012.

[2] R. Bird. *Introducción a la programación funcional con Haskell*. Prentice–Hall, 1999.

[3] H. C. Cunningham. Notes on functional programming with Haskell. Technical report, University of Mississippi, 2010.

[4] H. Daumé. Yet another Haskell tutorial. Technical report, University of Utah, 2006.

[5] A. Davie. *An introduction to functional programming systems using Haskell*. Cambridge University Press, 1992.

[6] K. Doets and J. van Eijck. *The Haskell road to logic, maths and programming*. King's College Publications, 2004.

[7] J. Fokker. Programación funcional. Technical report, Universidad de Utrech, 1996.

[8] P. Hudak. *The Haskell school of expression: Learning functional programming through multimedia*. Cambridge University Press, 2000.

[9] P. Hudak. The Haskell school of music (From signals to symphonies). Technical report, Yale University, 2012.

[10] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[11] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real world Haskell*. O'Reilly, 2008.

[12] G. Pólya. *Cómo plantear y resolver problemas*. Editorial Trillas, 1965.

[13] F. Rabhi and G. Lapalme. *Algorithms: A functional programming approach*. Addison–Wesley, 1999.

[14] B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. Gallardo. *Razonando con Haskell (Un curso sobre programación funcional)*. Thompson, 2004.

[15] S. Thompson. *Haskell: The craft of functional programming*. Addison–Wesley, third edition, 2011.