# Exámenes de "Programación funcional con Haskell"

Vol. 3 (Curso 2011–12)

José A. Alonso Jiménez

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

Sevilla, 10 de diciembre de 2012

# Índice general

# 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 2011–12) 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 2011–12) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2011–12) [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 tercer volumen de la serie de recopilaciones de exámenes de programación funcional con Haskell. Los volúmenes anteriores son

- Exámenes de "Programación funcional con Haskell". Vol. 1 (Curso 2009–10) [5]

[1] https://www.cs.us.es/~jalonso/cursos/i1m-11/temas/2011-12-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-11/ejercicios/ejercicios-I1M-2011.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol3
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

José A. Alonso
Sevilla, 10 de diciembre de 2012

[6] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol2

# 1

# Exámenes del grupo 1

José A. Alonso y Agustín Riscos

## 1.1. Examen 1 (26 de Octubre de 2011)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (26 de octubre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función numeroDeRaices tal que
-- (numeroDeRaices a b c) es el número de raíces reales de la ecuación
-- a*x^2 + b*x + c = 0. Por ejemplo,
--    numeroDeRaices 2 0 3    ==  0
--    numeroDeRaices 4 4 1    ==  1
--    numeroDeRaices 5 23 12  ==  2
-- ---------------------------------------------------------------------

numeroDeRaices a b c | d < 0     = 0
                     | d == 0    = 1
                     | otherwise = 2
               where d = b^2-4*a*c

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las dimensiones de los rectángulos puede representarse
-- por pares; por ejemplo, (5,3) representa a un rectángulo de base 5 y
-- altura 3. Definir la función mayorRectangulo tal que
-- (mayorRectangulo r1 r2) es el rectángulo de mayor área ente r1 y r2.
-- Por ejemplo,
```

```haskell
--    mayorRectangulo (4,6) (3,7)  ==  (4,6)
--    mayorRectangulo (4,6) (3,8)  ==  (4,6)
--    mayorRectangulo (4,6) (3,9)  ==  (3,9)
-- ---------------------------------------------------------------------

mayorRectanglo (a,b) (c,d) | a*b >= c*d = (a,b)
                           | otherwise = (c,d)

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

interior xs = tail (init xs)
```

## 1.2. Examen 2 (30 de Noviembre de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (30 de noviembre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [Problema 357 del Project Euler] Un número natural n
-- es especial si para todo divisor d de n, d+n/d es primo. Definir la
-- función
--    especial :: Integer -> Bool
-- tal que (especial x) se verifica si x es especial. Por ejemplo,
--    especial 30  ==  True
--    especial 20  ==  False
-- ---------------------------------------------------------------------

especial :: Integer -> Bool
especial x = and [esPrimo (d + x `div` d) | d <- divisores x]

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

```haskell
-- (esPrimo x) se verifica si x es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 8  ==  False
esPrimo :: Integer -> Bool
esPrimo x = divisores x == [1,x]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    sumaEspeciales :: Integer -> Integer
-- tal que (sumaEspeciales n) es la suma de los números especiales
-- menores o iguales que n. Por ejemplo,
--    sumaEspeciales 100  ==  401
-- ---------------------------------------------------------------------

-- Por comprensión
sumaEspeciales :: Integer -> Integer
sumaEspeciales n = sum [x | x <- [1..n], especial x]

-- Por recursión
sumaEspecialesR :: Integer -> Integer
sumaEspecialesR 0 = 0
sumaEspecialesR n | especial n = n + sumaEspecialesR (n-1)
                  | otherwise  = sumaEspecialesR (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    refinada :: [Float] -> [Float]
-- tal que (refinada xs) es la lista obtenida intercalando entre cada
-- dos elementos consecutivos de xs su media aritmética. Por ejemplo,
--    refinada [2,7,1,8]  ==  [2.0,4.5,7.0,4.0,1.0,4.5,8.0]
--    refinada [2]        ==  [2.0]
--    refinada []         ==  []
-- ---------------------------------------------------------------------

refinada :: [Float] -> [Float]
refinada (x:y:zs) = x : (x+y)/2 : refinada (y:zs)
refinada xs       = xs

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

```haskell
-- Ejercicio 3.1. En este ejercicio vamos a comprobar que la ecuación
-- diofántica
--    1/x_1 + 1/x_2 + ... + 1/x_n = 1
-- tiene solución; es decir, que para todo n >= 1 se puede construir una
-- lista de números enteros de longitud n tal que la suma de sus
-- inversos es 1. Para ello, basta observar que si
--    [x_1, x_2, ..., x_n]
-- es una solución, entonces
--    [2, 2*x_1, 2*x_2, ..., 2*x_n]
-- también lo es. Definir la función solucion tal que (solucion n) es la
-- solución de longitud n construida mediante el método anterior. Por
-- ejemplo,
--    solucion 1  ==  [1]
--    solucion 2  ==  [2,2]
--    solucion 3  ==  [2,4,4]
--    solucion 4  ==  [2,4,8,8]
--    solucion 5  ==  [2,4,8,16,16]
-- ---------------------------------------------------------------------

solucion 1 = [1]
solucion n = 2 : [2*x | x <- solucion (n-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función esSolucion tal que (esSolucion xs)
-- se verifica si la suma de los inversos de xs es 1. Por ejemplo,
--    esSolucion [4,2,4]       ==  True
--    esSolucion [2,3,4]       ==  False
--    esSolucion (solucion 5)  ==  True
-- ---------------------------------------------------------------------

esSolucion xs = sum [1/x | x <- xs] == 1
```

## 1.3. Examen 3 (25 de Enero de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 3º examen de evaluación continua (25 de enero de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [2 puntos] Un número es muy compuesto si tiene más
```

```haskell
-- divisores que sus anteriores. Por ejemplo, 12 es muy compuesto porque
-- tiene 6 divisores (1, 2, 3, 4, 6, 12) y todos los números del 1 al 11
-- tienen menos de 6 divisores.
--
-- Definir la función
--    esMuyCompuesto :: Int -> Bool
-- tal que (esMuyCompuesto x) se verifica si x es un número muy
-- compuesto. Por ejemplo,
--    esMuyCompuesto 24  ==  True
--    esMuyCompuesto 25  ==  False
-- Calcular  el menor número muy compuesto de 4 cifras.
-- ---------------------------------------------------------------------

esMuyCompuesto :: Int -> Bool
esMuyCompuesto x =
    and [numeroDivisores y < n | y <- [1..x-1]]
    where n = numeroDivisores x

-- (numeroDivisores x) es el número de divisores de x. Por ejemplo,
--    numeroDivisores 24  ==  8
numeroDivisores :: Int -> Int
numeroDivisores = length . divisores

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

-- Los primeros números muy compuestos son
--    ghci> take 14 [x | x <- [1..], esMuyCompuesto x]
--    [1,2,4,6,12,24,36,48,60,120,180,240,360,720]

-- El cálculo del menor número muy compuesto de 4 cifras es
--    ghci> head [x | x <- [1000..], esMuyCompuesto x]
--    1260

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. [1 punto] Definir la función
--    muyCompuesto :: Int -> Int
-- tal que (muyCompuesto n) es el n-ésimo número muy compuesto. Por
```

```haskell
-- ejemplo,
--    muyCompuesto 10  ==  180
-- ----------------------------------------------------------------------

muyCompuesto :: Int -> Int
muyCompuesto n =
    [x | x <- [1..], esMuyCompuesto x] !! n

-- ----------------------------------------------------------------------
-- Ejercicio 2.1. [2 puntos] [Problema 37 del proyecto Euler] Un número
-- primo es truncable si los números que se obtienen eliminado cifras,
-- de derecha a izquierda, son primos. Por ejemplo, 599 es un primo
-- truncable porque 599, 59 y 5 son primos; en cambio, 577 es un primo
-- no truncable porque 57 no es primo.
--
-- Definir la función
--    primoTruncable :: Int -> Bool
-- tal que (primoTruncable x) se verifica si x es un primo
-- truncable. Por ejemplo,
--    primoTruncable 599  ==  True
--    primoTruncable 577  ==  False
-- ----------------------------------------------------------------------

primoTruncable :: Int -> Bool
primoTruncable x
    | x < 10    = primo x
    | otherwise = primo x && primoTruncable (x `div` 10)

-- (primo x) se verifica si x es primo.
primo :: Int -> Bool
primo x = x == head (dropWhile (<x) primos)

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

-- ----------------------------------------------------------------------
-- Ejercicio 2.2. [1.5 puntos] Definir la función
```

```
--    sumaPrimosTruncables :: Int -> Int
-- tal que (sumaPrimosTruncables n) es la suma de los n primeros primos
-- truncables. Por ejemplo,
--    sumaPrimosTruncables 10  ==  249
-- Calcular la suma de los 20 primos truncables.
-- ---------------------------------------------------------------------

sumaPrimosTruncables :: Int -> Int
sumaPrimosTruncables n =
    sum (take n [x | x <- primos, primoTruncable x])

-- El cálculo es
--    ghci> sumaPrimosTruncables 20
--    2551

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. [2 puntos] Los números enteros se pueden ordenar como
-- sigue
--    0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5, -6, 6, -7, 7, ...
-- Definir la constante
--    enteros :: [Int]
-- tal que enteros es la lista de los enteros con la ordenación
-- anterior. Por ejemplo,
--    take 10 enteros  ==  [0,-1,1,-2,2,-3,3,-4,4,-5]
-- ---------------------------------------------------------------------

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

-- Otra definicición, por iteración, es
enteros1 :: [Int]
enteros1 = iterate siguiente 0
    where siguiente x | x >= 0    = -x-1
                      | otherwise = -x

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. [1.5 puntos] Definir la función
--    posicion :: Int -> Int
-- tal que (posicion x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
```

```haskell
--    posicion 2  ==  4
-- ---------------------------------------------------------------------

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

-- Definición por recursión
posicion1 :: Int -> Int
posicion1 x = aux enteros 0
    where aux (y:ys) n | x == y    = n
                       | otherwise = aux ys (n+1)

-- Definición por comprensión
posicion2 :: Int -> Int
posicion2 x = head [n | (n,y) <- zip [0..] enteros, y == x]

-- Definición directa
posicion3 :: Int -> Int
posicion3 x | x >= 0    = 2*x
            | otherwise = 2*(-x)-1
```

## 1.4. Examen 4 (29 de Febrero de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 4º examen de evaluación continua (29 de febrero de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] En el enunciado de uno de los problemas de
-- las Olimpiadas matemáticas de Brasil se define el primitivo de un
-- número como sigue:
--    Dado un número natural N, multiplicamos todos sus dígitos,
--    repetimos este procedimiento hasta que quede un solo dígito al
--    cual llamamos primitivo de N. Por ejemplo para 327: 3x2x7 = 42 y
--    4x2 = 8. Por lo tanto, el primitivo de 327 es 8.
--
-- Definir la función
--    primitivo :: Integer -> Integer
-- tal que (primitivo n) es el primitivo de n. Por ejemplo.
--    primitivo 327  ==  8
```

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

primitivo :: Integer -> Integer
primitivo n | n < 10    = n
            | otherwise = primitivo (producto n)

-- (producto n) es el producto de las cifras de n. Por ejemplo,
--    producto 327  ==  42
producto :: Integer -> Integer
producto = product . cifras

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Definir la función
--    sumas :: Int -> [Int] -> [Int]
-- tal que (sumas n xs) es la lista de los números que se pueden obtener
-- como suma de n, o menos, elementos de xs. Por ejemplo,
--    sumas 0 [2,5]    ==  [0]
--    sumas 1 [2,5]    ==  [2,5,0]
--    sumas 2 [2,5]    ==  [4,7,2,10,5,0]
--    sumas 3 [2,5]    ==  [6,9,4,12,7,2,15,10,5,0]
--    sumas 2 [2,3,5]  ==  [4,5,7,2,6,8,3,10,5,0]
-- ---------------------------------------------------------------------

sumas :: Int -> [Int] -> [Int]
sumas 0 _  = [0]
sumas _ [] = [0]
sumas n (x:xs) = [x+y | y <- sumas (n-1) (x:xs)] ++ sumas n xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Los árboles binarios se pueden representar
-- mediante el siguiente tipo de datos
--    data Arbol = H
--               | N Int Arbol Arbol
-- Por ejemplo, el árbol
--            9
```

```
--           / \
--          /   \
--         3     7
--        / \   / \
--       /   \ H   H
--      2     4
--     / \   / \
--    H   H H   H
-- se representa por
--    N 9 (N 3 (N 2 H H) (N 4 H H)) (N 7 H H)
-- Definir la función
--    ramaIzquierda :: Arbol -> [Int]
-- tal que (ramaIzquierda a) es la lista de los valores de los nodos de
-- la rama izquierda del árbol a. Por ejemplo,
--    ghci> ramaIzquierda (N 9 (N 3 (N 2 H H) (N 4 H H)) (N 7 H H))
--    [9,3,2]
-- ------------------------------------------------------------------

data Arbol = H
           | N Int Arbol Arbol

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

-- ------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] Un primo permutable es un número primo tal
-- que todos los números obtenidos permutando sus cifras son primos. Por
-- ejemplo, 337 es un primo permutable ya que 337, 373 y 733 son
-- primos.
--
-- Definir la función
--    primoPermutable :: Integer -> Bool
-- tal que (primoPermutable x) se verifica si x es un primo
-- permutable. Por ejemplo,
--    primoPermutable 17  ==  True
--    primoPermutable 19  ==  False
-- ------------------------------------------------------------------

primoPermutable :: Integer -> Bool
```

```haskell
primoPermutable x = and [primo y | y <- permutacionesN x]

-- (permutacionesN x) es la lista de los números obtenidos permutando
-- las cifras de x. Por ejemplo,
permutacionesN :: Integer -> [Integer]
permutacionesN x = [read ys | ys <- permutaciones (show x)]

-- (intercala x ys) es la lista de las listas obtenidas intercalando x
-- entre los elementos de ys. Por ejemplo,
--    intercala 1 [2,3]  ==  [[1,2,3],[2,1,3],[2,3,1]]
intercala :: a -> [a] -> [[a]]
intercala x [] = [[x]]
intercala x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala x ys]

-- (permutaciones xs) es la lista de las permutaciones de la lista
-- xs. Por ejemplo,
--    permutaciones "bc"   ==  ["bc","cb"]
--    permutaciones "abc"  ==  ["abc","bac","bca","acb","cab","cba"]
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat [intercala x ys | ys <- permutaciones xs]

-- (primo x) se verifica si x es primo.
primo :: Integer -> Bool
primo x = x == head (dropWhile (<x) primos)

-- primos es la lista de los números primos.
primos :: [Integer ]
primos = criba [2..]
    where criba :: [Integer] -> [Integer]
          criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]
```

## 1.5. Examen 5 (21 de Marzo de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 5º examen de evaluación continua (21 de marzo de 2012)
-- ---------------------------------------------------------------------

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

```haskell
-- Ejercicio 1. [2.5 puntos] Dos números son equivalentes si la media de
-- sus cifras son iguales. Por ejemplo, 3205 y 41 son equvalentes ya que
-- (3+2+0+5)/4 = (4+1)/2. Definir la función
--    equivalentes :: Int -> Int -> Bool
-- tal que (equivalentes x y) se verifica si los números x e y son
-- equivalentes. Por ejemplo,
--    equivalentes 3205 41  ==  True
--    equivalentes 3205 25  ==  False
-- ---------------------------------------------------------------------

equivalentes :: Int -> Int -> Bool
equivalentes x y = media (cifras x) == media (cifras y)

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 3205  ==  [3,2,0,5]
cifras :: Int -> [Int]
cifras n = [read [y] | y <- show n]

-- (media xs) es la media de la lista xs. Por ejemplo,
--    media [3,2,0,5]  ==  2.5
media :: [Int] -> Float
media xs = (fromIntegral (sum xs)) / (fromIntegral (length xs))

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Definir la función
--    relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
--    relacionados (<) [2,3,7,9]                ==  True
--    relacionados (<) [2,3,1,9]                ==  False
--    relacionados equivalentes [3205,50,5014]  ==  True
-- ---------------------------------------------------------------------

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

-- Una definición alternativa es
relacionados' :: (a -> a -> Bool) -> [a] -> Bool
relacionados' r xs = and [r x y | (x,y) <- zip xs (tail xs)]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Definir la función
--    primosEquivalentes :: Int -> [[Int]]
-- tal que (primosEquivalentes n) es la lista de las sucesiones de n
-- números primos consecutivos equivalentes. Por ejemplo,
--    take 2 (primosEquivalentes 2)  ==  [[523,541],[1069,1087]]
--    head (primosEquivalentes 3)    ==  [22193,22229,22247]
-- ---------------------------------------------------------------------

primosEquivalentes :: Int -> [[Int]]
primosEquivalentes n = aux primos
    where aux (x:xs) | relacionados equivalentes ys = ys : aux xs
                     | otherwise                    = aux xs
                     where ys = take n (x:xs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] Los polinomios pueden representarse
-- de forma dispersa o densa. Por ejemplo, el polinomio
-- 6x^4-5x^2+4x-7 se puede representar de forma dispersa por
-- [6,0,-5,4,-7] y de forma densa por [(4,6),(2,-5),(1,4),(0,-7)].
-- Definir la función
--    densa :: [Int] -> [(Int,Int)]
-- tal que (densa xs) es la representación densa del polinomio cuya
-- representación dispersa es xs. Por ejemplo,
--   densa [6,0,-5,4,-7]  ==  [(4,6),(2,-5),(1,4),(0,-7)]
--   densa [6,0,0,3,0,4]  ==  [(5,6),(2,3),(0,4)]
-- ---------------------------------------------------------------------

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

## 1.6. Examen 6 ( 2 de Mayo de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 6º examen de evaluación continua (2 de mayo de 2012)
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número x es especial si el número de ocurrencia de
-- cada dígito d de x en x^2 es el doble del número de ocurrencia de d
-- en x. Por ejemplo, 72576 es especial porque tiene un 2, un 5, un 6 y
-- dos 7 y su cuadrado es 5267275776 que tiene exactamente dos 2, dos 5,
-- dos 6 y cuatro 7.
--
-- Definir la función
--    especial :: Integer -> Bool
-- tal que (especial x) se verifica si x es un número especial. Por
-- ejemplo,
--    especial 72576  ==  True
--    especial 12     ==  False
-- Calcular el menor número especial mayor que 72576.
-- ---------------------------------------------------------------------

especial :: Integer -> Bool
especial x =
    sort (ys ++ ys) == sort (show (x^2))
    where ys = show x

-- EL cálculo es
--    ghci> head [x | x <- [72577..], especial x]
--    406512

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    posiciones :: Eq a => a -> Array (Int,Int) a -> [(Int,Int)]
-- tal que (posiciones x p) es la lista de las posiciones de la matriz p
-- cuyo valor es x. Por ejemplo,
--    ghci> let p = listArray ((1,1),(2,3)) [1,2,3,2,4,6]
```

```
--    ghci> p
--    array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
--                         ((2,1),2),((2,2),4),((2,3),6)]
--    ghci> posiciones 2 p
--    [(1,2),(2,1)]
--    ghci> posiciones 6 p
--    [(2,3)]
--    ghci> posiciones 7 p
--    []
-- ---------------------------------------------------------------------

posiciones :: Eq a => a -> Array (Int,Int) a -> [(Int,Int)]
posiciones x p = [(i,j) | (i,j) <- indices p, p!(i,j) == x]

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. [Basado en el problema 341 del proyecto Euler]. La
-- sucesión de Golomb {G(n)} es una sucesión auto descriptiva: es la
-- única sucesión no decreciente de números naturales tal que el número
-- n aparece G(n) veces en la sucesión. Los valores de G(n) para los
-- primeros números son los siguientes:
--    n       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
```

```
--    G(n)    1 2 2 3 3 4 4 4 5  5  5  6  6  6  6 ...
-- En los apartados de este ejercicio se definirá una función para
-- calcular los términos de la sucesión de Golomb.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    golomb :: Int -> Int
-- tal que (golomb n) es el n-ésimo término de la sucesión de Golomb.
-- Por ejemplo,
--    golomb 5  ==  3
--    golomb 9  ==  5
-- Indicación: Se puede usar la función sucGolomb del apartado 2.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    sucGolomb :: [Int]
-- tal que sucGolomb es la lista de los términos de la sucesión de
-- Golomb. Por ejemplo,
--    take 15 sucGolomb  ==  [1,2,2,3,3,4,4,4,5,5,5,6,6,6,6]
-- Indicación: Se puede usar la función subSucGolomb del apartado 3.
-- ---------------------------------------------------------------------

sucGolomb :: [Int]
sucGolomb = subSucGolomb 1

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    subSucGolomb :: Int -> [Int]
-- tal que (subSucGolomb x) es la lista de los términos de la sucesión
-- de Golomb a partir de la primera ocurrencia de x. Por ejemplo,
--    take 10 (subSucGolomb 4)  ==  [4,4,4,5,5,5,6,6,6,6]
-- Indicación: Se puede usar la función golomb del apartado 1.
-- ---------------------------------------------------------------------

subSucGolomb :: Int -> [Int]
```

```haskell
subSucGolomb 1 = 1 : subSucGolomb 2
subSucGolomb 2 = [2,2] ++ subSucGolomb 3
subSucGolomb x = replicate (golomb x) x ++ subSucGolomb (x+1)

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

## 1.7. Examen 7 (25 de Junio de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 7º examen de evaluación continua (25 de junio de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Definir la función
--    ceros :: Int -> Int
-- tal que (ceros n) es el número de ceros en los que termina el número
-- n. Por ejemplo,
--    ceros 30500  ==  2
--    ceros 30501  ==  0
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
ceros :: Int -> Int
ceros n | n `rem` 10 == 0 = 1 + ceros (n `div`10)
        | otherwise       = 0

-- 2ª definición (sin recursión):
ceros2 :: Int -> Int
ceros2 n = length (takeWhile (=='0') (reverse (show n)))

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    superpar :: Int -> Bool
-- tal que (superpar n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
```

```haskell
--    superpar 426  ==  True
--    superpar 456  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión)
superpar :: Int -> Bool
superpar n | n < 10    = even n
           | otherwise = even n && superpar (n `div` 10)

-- 2ª definición (por comprensión):
superpar2 :: Int -> Bool
superpar2 n = and [even d | d <- digitos n]

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

-- 3ª definición (por recursión sobre los dígitos):
superpar3 :: Int -> Bool
superpar3 n = sonPares (digitos n)
    where sonPares []     = True
          sonPares (d:ds) = even d && sonPares ds

-- la función sonPares se puede definir por plegado:
superpar3' :: Int -> Bool
superpar3' n = sonPares (digitos n)
    where sonPares ds = foldr ((&&) . even) True ds

-- 4ª definición (con all):
superpar4 :: Int -> Bool
superpar4 n = all even (digitos n)

-- 5ª definición (con filter):
superpar5 :: Int -> Bool
superpar5 n = filter even (digitos n) == digitos n

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    potenciaFunc :: Int -> (a -> a) -> a -> a
-- tal que (potenciaFunc n f x) es el resultado de aplicar n veces la
-- función f a x. Por ejemplo,
```

```haskell
--    potenciaFunc 3 (*10) 5  ==  5000
--    potenciaFunc 4 (+10) 5  ==  45
-- ---------------------------------------------------------------------

potenciaFunc :: Int -> (a -> a) -> a -> a
potenciaFunc 0 _ x = x
potenciaFunc n f x = potenciaFunc (n-1) f (f x)

-- 2ª definición (con iterate):
potenciaFunc2 :: Int -> (a -> a) -> a -> a
potenciaFunc2 n f x = last (take (n+1) (iterate f x))

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Las expresiones aritméticas con una variable
-- (denotada por X) se pueden representar mediante el siguiente tipo
--    data Expr = Num Int
--              | Suma Expr Expr
--              | X
-- Por ejemplo, la expresión "X+(13+X)" se representa por
-- "Suma X (Suma (Num 13) X)".
--
-- Definir la función
--    numVars :: Expr -> Int
-- tal que (numVars e) es el número de variables en la expresión e. Por
-- ejemplo,
--    numVars (Num 3)                    ==  0
--    numVars X                          ==  1
--    numVars (Suma X (Suma (Num 13) X))  ==  2
-- ---------------------------------------------------------------------

data Expr = Num Int
          | Suma Expr Expr
          | X

numVars :: Expr -> Int
numVars (Num n)    = 0
numVars (Suma a b) = numVars a + numVars b
numVars X          = 1

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

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

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

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

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

## 1.8. Examen 8 (29 de Junio de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- Examen de la 1ª convocatoria (29 de junio de 2012)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Definir la función
--    paresOrdenados :: [a] -> [(a,a)]
-- tal que (paresOrdenados xs) es la lista de todos los pares de
-- elementos (x,y) de xs, tales que x ocurren en xs antes que y. Por
-- ejemplo,
--    paresOrdenados [3,2,5,4] == [(3,2),(3,5),(3,4),(2,5),(2,4),(5,4)]
--    paresOrdenados [3,2,5,3] == [(3,2),(3,5),(3,3),(2,5),(2,3),(5,3)]
-- ---------------------------------------------------------------------

-- 1ª definición:
paresOrdenados :: [a] -> [(a,a)]
paresOrdenados []     = []
paresOrdenados (x:xs) = [(x,y) | y <- xs] ++ paresOrdenados xs

-- 2ª definición:
paresOrdenados2 :: [a] -> [(a,a)]
paresOrdenados2 [] = []
paresOrdenados2 (x:xs) =
    foldr (\y ac -> (x,y):ac) (paresOrdenados2 xs) xs

-- 3ª definición (con repeat):
paresOrdenados3 :: [a] -> [(a,a)]
paresOrdenados3 []     = []
paresOrdenados3 (x:xs) = zip (repeat x) xs ++ paresOrdenados3 xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    sumaDeDos :: Int -> [Int] -> Maybe (Int,Int)
-- tal que (sumaDeDos x ys) decide si x puede expresarse como suma de
-- dos elementos de ys y, en su caso, devuelve un par de elementos de ys
```

```haskell
-- cuya suma es x. Por ejemplo,
--    sumaDeDos 9 [7,4,6,2,5]  ==  Just (7,2)
--    sumaDeDos 5 [7,4,6,2,5]  ==  Nothing
-- ---------------------------------------------------------------------

sumaDeDos :: Int -> [Int] -> Maybe (Int,Int)
sumaDeDos _ []  =  Nothing
sumaDeDos _ [_] =  Nothing
sumaDeDos y (x:xs) | y-x `elem` xs = Just (x,y-x)
                   | otherwise     = sumaDeDos y xs

-- 2ª definición (usando paresOrdenados):
sumaDeDos2 :: Int -> [Int] -> Maybe (Int,Int)
sumaDeDos2 x xs
    | null ys   = Nothing
    | otherwise = Just (head ys)
    where ys = [(a,b) | (a,b) <- paresOrdenados xs , a+b == x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    esProductoDeDosPrimos :: Int -> Bool
-- tal que (esProductoDeDosPrimos n) se verifica si n es el producto de
-- dos primos distintos. Por ejemplo,
--    esProductoDeDosPrimos 6  ==  True
--    esProductoDeDosPrimos 9  ==  False
-- ---------------------------------------------------------------------

esProductoDeDosPrimos :: Int -> Bool
esProductoDeDosPrimos n =
    [x | x <- primosN,
         mod n x == 0,
         div n x /= x,
         div n x `elem` primosN] /= []
    where primosN = takeWhile (<=n) primos

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] La expresiones aritméticas se pueden
-- representar mediante el siguiente tipo
--    data Expr = V Char
--              | N Int
--              | S Expr Expr
--              | P Expr Expr
--              deriving Show
-- por ejemplo, representa la expresión "z*(3+x)" se representa por
-- (P (V 'z') (S (N 3) (V 'x'))).
--
-- Definir la función
--    sustitucion :: Expr -> [(Char, Int)] -> Expr
-- tal que (sustitucion e s) es la expresión obtenida sustituyendo las
-- variables de la expresión e según se indica en la sustitución s. Por
-- ejemplo,
--    ghci> sustitucion (P (V 'z') (S (N 3) (V 'x'))) [('x',7),('z',9)]
--    P (N 9) (S (N 3) (N 7))
--    ghci> sustitucion (P (V 'z') (S (N 3) (V 'y'))) [('x',7),('z',9)]
--    P (N 9) (S (N 3) (V 'y'))
-- ---------------------------------------------------------------------

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

sustitucion :: Expr -> [(Char, Int)] -> Expr
sustitucion e [] = e
sustitucion (V c) ((d,n):ps) | c == d = N n
                             | otherwise = sustitucion (V c) ps
sustitucion (N n) _ = N n
sustitucion (S e1 e2) ps = S (sustitucion e1 ps) (sustitucion e2 ps)
sustitucion (P e1 e2) ps = P (sustitucion e1 ps) (sustitucion e2 ps)

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] (Problema 345 del proyecto Euler) Las
-- matrices puede representarse mediante tablas cuyos índices son pares
```

```haskell
-- de números naturales:
--    type Matriz = Array (Int,Int) Int
-- Definir la función
--    maximaSuma :: Matriz -> Int
-- tal que (maximaSuma p) es el máximo de las sumas de las listas de
-- elementos de la matriz p tales que cada elemento pertenece sólo a una
-- fila y a una columna. Por ejemplo,
--    ghci> maximaSuma (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
--    17
-- ya que las selecciones, y sus sumas, de la matriz
--    |1 2 3|
--    |8 4 9|
--    |5 6 7|
-- son
--    [1,4,7] --> 12
--    [1,9,6] --> 16
--    [2,8,7] --> 17
--    [2,9,5] --> 16
--    [3,8,6] --> 17
--    [3,4,5] --> 12
-- Hay dos selecciones con máxima suma: [2,8,7] y [3,8,6].
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

maximaSuma :: Matriz -> Int
maximaSuma p = maximum [sum xs | xs <- selecciones p]

-- (selecciones p) es la lista de las selecciones en las que cada
-- elemento pertenece a un única fila y a una única columna de la matriz
-- p. Por ejemplo,
--    ghci> selecciones (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
--    [[1,4,7],[2,8,7],[3,4,5],[2,9,5],[3,8,6],[1,9,6]]
selecciones :: Matriz -> [[Int]]
selecciones p =
    [[p!(i,j) | (i,j) <- ijs] |
     ijs <- [zip [1..n] xs | xs <- permutations [1..n]]]
    where (_,(m,n)) = bounds p

-- Nota: En la anterior definición se ha usado la función pernutations
```

```haskell
-- de Data.List. También se puede definir mediante
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat [intercala x ys | ys <- permutaciones xs]

-- (intercala x ys) es la lista de las listas obtenidas intercalando x
-- entre los elementos de ys. Por ejemplo,
--    intercala 1 [2,3]  ==  [[1,2,3],[2,1,3],[2,3,1]]
intercala :: a -> [a] -> [[a]]
intercala x [] = [[x]]
intercala x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala x ys]

-- 2ª solución (mediante submatrices):
maximaSuma2 :: Matriz -> Int
maximaSuma2 p
    | (m,n) == (1,1) = p!(1,1)
    | otherwise = maximum [p!(1,j)
                  + maximaSuma2 (submatriz 1 j p) | j <- [1..n]]
    where (m,n) = dimension p

-- (dimension p) es la dimensión de la matriz p.
dimension :: Matriz -> (Int,Int)
dimension = snd . bounds

-- (submatriz i j p) es la matriz obtenida a partir de la p eliminando
-- la fila i y la columna j. Por ejemplo,
--    ghci> submatriz 2 3 (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
--    array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),5),((2,2),6)]
submatriz :: Int -> Int -> Matriz -> Matriz
submatriz i j p =
    array ((1,1), (m-1,n -1))
          [((k,l), p ! f k l) | k <- [1..m-1], l <- [1.. n-1]]
    where (m,n) = dimension p
          f k l | k < i  && l < j  = (k,l)
                | k >= i && l < j  = (k+1,l)
                | k < i  && l >= j = (k,l+1)
                | otherwise        = (k+1,l+1)
```

## 1.9. Examen 9 ( 9 de Septiembre de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la convocatoria de Septiembre (10 de septiembre de 2012)
-- ---------------------------------------------------------------------

import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [1.7 puntos] El enunciado de uno de los problemas de la
-- IMO de 1966 es
--    Calcular el número de maneras de obtener 500 como suma de números
--    naturales consecutivos.
-- Definir la función
--    sucesionesConSuma :: Int -> [(Int,Int)]
-- tal que (sucesionesConSuma n) es la lista de las sucesiones de
-- números naturales consecutivos con suma n. Por ejemplo,
--    sucesionesConSuma 15  == [(1,5),(4,6),(7,8),(15,15)]
-- ya que 15 = 1+2+3+4+5 = 4+5+6 = 7+8 = 15.
--
-- Calcular la solución del problema usando sucesionesConSuma.
-- ---------------------------------------------------------------------

sucesionesConSuma :: Int -> [(Int,Int)]
sucesionesConSuma n =
    [(x,y) | y <- [1..n], x <- [1..y], sum [x..y] == n]

-- La solución del problema es
--    ghci> length (sucesionesConSuma 500)
--    4

-- Otra definción, usando la fórmula de la suma es
sucesionesConSuma2 :: Int -> [(Int,Int)]
sucesionesConSuma2 n =
    [(x,y) | y <- [1..n], x <- [1..y], (x+y)*(y-x+1) == 2*n]

-- La 2ª definición es más eficiente
--    ghci> :set +s
--    ghci> sucesionesConSuma 500
--    [(8,32),(59,66),(98,102),(500,500)]
```

```
--    (1.47 secs, 1452551760 bytes)
--    ghci> sucesionesConSuma2 500
--    [(8,32),(59,66),(98,102),(500,500)]
--    (0.31 secs, 31791148 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2 [1.7 puntos] Definir la función
--    inversiones :: Ord a => [a] -> [(a,Int)]
-- tal que (inversiones xs) es la lista de pares formados por los
-- elementos x de xs junto con el número de elementos de xs que aparecen
-- a la derecha de x y son mayores que x. Por ejemplo,
--    inversiones [7,4,8,9,6]  == [(7,2),(4,3),(8,1),(9,0),(6,0)]
-- ---------------------------------------------------------------------

inversiones :: Ord a => [a] -> [(a,Int)]
inversiones []     = []
inversiones (x:xs) = (x,length (filter (>x) xs)) : inversiones xs

-- ---------------------------------------------------------------------
-- Ejercicio 3 [1.7 puntos] Se considera el siguiente procedimiento de
-- reducción de listas: Se busca un par de elementos consecutivos
-- iguales pero con signos opuestos, se eliminan dichos elementos y se
-- continúa el proceso hasta que no se encuentren pares de elementos
-- consecutivos iguales pero con signos opuestos. Por ejemplo, la
-- reducción de [-2,1,-1,2,3,4,-3] es
--    [-2,1,-1,2,3,4,-3]    (se elimina el par (1,-1))
--    -> [-2,2,3,4,-3]      (se elimina el par (-2,2))
--    -> [3,4,-3]           (el par (3,-3) no son consecutivos)
-- Definir la función
--    reducida :: [Int] -> [Int]
-- tal que (reducida xs) es la lista obtenida aplicando a xs el proceso
-- de eliminación de pares de elementos consecutivos opuestos. Por
-- ejemplo,
--    reducida [-2,1,-1,2,3,4,-3]          == [3,4,-3]
--    reducida [-2,1,-1,2,3,-4,4,-3]       == []
--    reducida [-2,1,-1,2,5,3,-4,4,-3]     == [5]
--    reducida [-2,1,-1,2,5,3,-4,4,-3,-5]  == []
-- ---------------------------------------------------------------------

paso :: [Int] -> [Int]
```

```haskell
paso [] = []
paso [x] = [x]
paso (x:y:zs) | x == -y   = paso zs
              | otherwise = x : paso (y:zs)

reducida :: [Int] -> [Int]
reducida xs | xs == ys  = xs
            | otherwise = reducida ys
            where ys = paso xs

reducida2 :: [Int] -> [Int]
reducida2 xs = aux xs []
    where aux [] ys                    = reverse ys
          aux (x:xs) (y:ys) | x == -y = aux xs ys
          aux (x:xs) ys                = aux xs (x:ys)

-- ---------------------------------------------------------------------
-- Ejercicio 4. [1.7 puntos] Las variaciones con repetición de una lista
-- xs se puede ordenar por su longitud y las de la misma longitud
-- lexicográficamente. Por ejemplo, las variaciones con repetición de
-- "ab" son
--    "","a","b","aa","ab","ba","bb","aaa","aab","aba","abb","baa",...
-- y las de "abc" son
--    "","a","b","c","aa","ab","ac","ba","bb","bc","ca","cb",...
-- Definir la función
--    posicion :: Eq a => [a] -> [a] -> Int
-- tal que (posicion xs ys) es posición de xs en la lista ordenada de
-- las variaciones con repetición de los elementos de ys. Por ejemplo,
--    posicion "ba" "ab"       == 5
--    posicion "ba" "abc"      == 7
--    posicion "abccba" "abc"  == 520
-- ---------------------------------------------------------------------

posicion :: Eq a => [a] -> [a] -> Int
posicion xs ys =
    length (takeWhile (/=xs) (variaciones ys))

variaciones :: [a] -> [[a]]
variaciones xs = concat aux
    where aux = [[]] : [[x:ys | x <- xs, ys <- yss] | yss <- aux]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5. [1.6 puntos] Un árbol ordenado es un árbol binario tal
-- que para cada nodo, los elementos de su subárbol izquierdo son
-- menores y los de su subárbol derecho son mayores. Por ejemplo,
--         5
--        / \
--       /   \
--      3     7
--     / \   / \
--    1   4 6   9
-- El tipo de los árboles binarios se define por
--    data Arbol = H Int
--               | N Int Arbol Arbol
-- con lo que el ejemplo anterior se define por
--    ejArbol = N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9))
--
-- Definir la función
--    ancestroMasProximo :: Int -> Int -> Int
-- tal que (ancestroMasProximo x y a) es el ancestro más próximo de los
-- nodos x e y en el árbol a. Por ejemplo,
--    ancestroMasProximo 4 1 ejArbol  == 3
--    ancestroMasProximo 4 6 ejArbol  == 5
-- ---------------------------------------------------------------------

data Arbol = H Int
           | N Int Arbol Arbol

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

ancestroMasProximo :: Int -> Int -> Arbol -> Int
ancestroMasProximo x y (N z i d)
    | x < z && y < z = ancestroMasProximo x y i
    | x > z && y > z = ancestroMasProximo x y d
    | otherwise      = z

-- ---------------------------------------------------------------------
-- Ejercicio 6. [1.6 puntos] Las matrices puede representarse mediante
-- tablas cuyos índices son pares de números naturales:
```

```
--    type Matriz = Array (Int,Int) Int
-- Definir la función
--    maximos :: Matriz -> [Int]
-- tal que (maximos p) es la lista de los máximos locales de la matriz
-- p; es decir de los elementos de p que son mayores que todos sus
-- vecinos. Por ejemplo,
--    ghci> maximos (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,0,2,5,4])
--    [9,7]
-- ya que los máximos locales de la matriz
--    |9 4 6 5|
--    |8 1 7 3|
--    |0 2 5 4|
-- son 9 y 7.
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

maximos :: Matriz -> [Int]
maximos p =
    [p!(i,j) | (i,j) <- indices p,
               and [p!(a,b) < p!(i,j) | (a,b) <- vecinos (i,j)]]
    where (_,(m,n)) = bounds p
          vecinos (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                                   b <- [max 1 (j-1)..min n (j+1)],
                                   (a,b) /= (i,j)]
```

## 1.10. Examen 10 (10 de Diciembre de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la convocatoria de Diciembre de 2012
-- ---------------------------------------------------------------------

import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir una función verificanP
--    verificanP :: Int -> Integer -> (Integer -> Bool) -> Bool
-- tal que (verificanP k n p) se cumple si los primeros k dígitos del
-- número n verifican la propiedad p y el (k+1)-ésimo no la verifica.
-- Por ejemplo,
```

```
--   verificanP 3 224119 even == True
--   verificanP 3 265119 even == False
--   verificanP 3 224619 even == False
-- ---------------------------------------------------------------------

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

verificanP :: Int -> Integer -> (Integer -> Bool) -> Bool
verificanP k n p = length (takeWhile p (digitos n)) == k

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--   primosPK :: (Integer -> Bool) -> Int -> [Integer]
-- tal que (primosPK p k) es la lista de los números primos cuyos
-- primeros k dígitos verifican la propiedad p y el (k+1)-ésimo no la
-- verifica. Por ejemplo,
--   take 10 (primosPK even 4)
--   [20021,20023,20029,20047,20063,20089,20201,20249,20261,20269]
-- ---------------------------------------------------------------------

primosPK :: (Integer -> Bool) -> Int -> [Integer]
primosPK p k = [n | n <- primos, verificanP k n p]

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    suma2 :: Int -> [Int] -> Maybe (Int,Int)
-- tal que (suma2 n xs) es un par de elementos (x,y) de la lista xs cuya
-- suma es n, si éstos existe. Por ejemplo,
--    suma2 27 [1..6] == Nothing
--    suma2 7 [1..6]  == Just (1,6)
-- ---------------------------------------------------------------------

suma2 :: Int -> [Int] -> Maybe (Int,Int)
```

```haskell
suma2 _ []                    =  Nothing
suma2 _ [_]                   =  Nothing
suma2 y (x:xs) | y-x `elem` xs = Just (x,y-x)
               | otherwise     = suma2 y xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideremos el tipo de los árboles binarios definido
-- por
--    data Arbol = H Int
--               | N Int Arbol Arbol
--               deriving Show
-- Por ejemplo,
--         5
--        / \
--       /   \
--      3     7
--     / \   / \
--    1   4 6   9
-- se representa por
--    ejArbol = N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9))
--
-- Definir la función
--    aplica :: (Int -> Int) -> Arbol -> Arbol
-- tal que (aplica f a) es el árbol obtenido aplicando la función f a
-- los elementos del árbol a. Por ejemplo,
--    ghci> aplica (+2) ejArbol
--    N 7 (N 5 (H 3) (H 6)) (N 9 (H 8) (H 11))
--    ghci> aplica (*5) ejArbol
--    N 25 (N 15 (H 5) (H 20)) (N 35 (H 30) (H 45))
-- ---------------------------------------------------------------------

data Arbol = H Int
           | N Int Arbol Arbol
           deriving Show

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

aplica :: (Int -> Int) -> Arbol -> Arbol
aplica f (H x)     = H (f x)
```

```haskell
aplica f (N x i d) = N (f x) (aplica f i) (aplica f d)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las matrices puede representarse mediante tablas cuyos
-- índices son pares de números naturales:
--    type Matriz = Array (Int,Int) Int
-- Definir la función
--    algunMenor :: Matriz -> [Int]
-- tal que (algunMenor p) es la lista de los elementos de p que tienen
-- algún vecino menor que él. Por ejemplo,
--    algunMenor (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,4,2,5,4])
--    [9,4,6,5,8,7,4,2,5,4]
-- pues sólo el 1 y el 3 no tienen ningún vecino menor en la matriz
--    |9 4 6 5|
--    |8 1 7 3|
--    |4 2 5 4|
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

algunMenor :: Matriz -> [Int]
algunMenor p =
    [p!(i,j) | (i,j) <- indices p,
               or [p!(a,b) < p!(i,j) | (a,b) <- vecinos (i,j)]]
    where (_,(m,n)) = bounds p
          vecinos (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                                   b <- [max 1 (j-1)..min n (j+1)],
                                   (a,b) /= (i,j)]
```

# 2

# Exámenes del grupo 2

María J. Hidalgo

## 2.1. Examen 1 (27 de Octubre de 2011)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 1º examen de evaluación continua (27 de octubre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Los puntos del plano se pueden representar mediante
-- pares de números reales.
--
-- Definir la función estanEnLinea tal que (estanEnLinea p1 p2) se
-- verifica si los puntos p1 y p2 están en la misma línea vertical u
-- horizontal. Por ejemplo,
--    estanEnLinea (1,3) (1,-6)  == True
--    estanEnLinea (1,3) (-1,-6) == False
--    estanEnLinea (1,3) (-1,3)  == True
-- ---------------------------------------------------------------------

estanEnLinea (x1,y1) (x2,y2) = x1 == x2 || y1 == y2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función pCardinales tal que
-- (pCardinales (x,y) d) es la lista formada por los cuatro puntos
-- situados al norte, sur este y oeste, a una distancia d de (x,y).
-- Por ejemplo,
--    pCardinales (0,0) 2  == [(0,2),(0,-2),(-2,0),(2,0)]
```

```haskell
--    pCardinales (-1,3) 4 == [(-1,7),(-1,-1),(-5,3),(3,3)]
-- ---------------------------------------------------------------------

pCardinales (x,y) d = [(x,y+d),(x,y-d),(x-d,y),(x+d,y)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función elementosCentrales tal que
-- (elementosCentrales xs) es la lista formada por el elemento central
-- si xs tiene un número impar de elementos, y los dos elementos
-- centrales si xs tiene un número par de elementos. Por ejemplo,
--    elementosCentrales [1..8] == [4,5]
--    elementosCentrales [1..7] == [4]
-- ---------------------------------------------------------------------

elementosCentrales xs
    | even n    = [xs!!(m-1), xs!!m]
    | otherwise = [xs !! m]
    where n = length xs
          m = n `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 4. Consideremos el problema geométrico siguiente: partir un
-- segmento en dos trozos, a y b, de forma que, al dividir la longitud
-- total entre el mayor (supongamos que es a), obtengamos el mismo
-- resultado que al dividir la longitud del mayor entre la del menor.
--
-- Definir la función esParAureo tal que (esParAureo a b)
-- se verifica si a y b forman un par con la característica anterior.
-- Por ejemplo,
--    esParAureo 3 5                == False
--    esParAureo 1 2                == False
--    esParAureo ((1+ (sqrt 5))/2) 1 == True
-- ---------------------------------------------------------------------

esParAureo a b = (a+b)/c == c/d
    where c = max a b
          d = min a b
```

## 2.2. Examen 2 ( 1 de Diciembre de 2011)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 2º examen de evaluación continua (1 de diciembre de 2011)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--     todosIgualesR:: Eq a => [a] -> Bool
-- tal que (todosIgualesR xs) se verifica si los elementos de la
-- lista xs son todos iguales. Por ejemplo,
--     todosIgualesR [1..5]    == False
--     todosIgualesR [2,2,2]   == True
--     todosIgualesR ["a","a"] == True
-- ---------------------------------------------------------------------

todosIgualesR:: Eq a => [a] -> Bool
todosIgualesR []  = True
todosIgualesR [_] = True
todosIgualesR (x:y:xs) = x == y && todosIgualesR (y:xs)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por comprensión, la función
--     todosIgualesC:: Eq a => [a] -> Bool
-- tal que (todosIgualesC xs) se verifica si los elementos de la
-- lista xs son todos iguales. Por ejemplo,
--     todosIgualesC [1..5]    == False
--     todosIgualesC [2,2,2]   == True
--     todosIgualesC ["a","a"] == True
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que ambas definiciones
-- coinciden.
```

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

-- La propiedad es
prop_todosIguales :: [Int] -> Bool
prop_todosIguales xs = todosIgualesR xs == todosIgualesC xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    intercalaCero :: [Int] -> [Int]
-- tal que (intercalaCero xs) es la lista que resulta de intercalar un 0
-- entre cada dos elementos consecutivos x e y, cuando x es mayor que
-- y. Por ejemplo,
--    intercalaCero [2,1,8,3,5,1,9] == [2,0,1,8,0,3,5,0,1,9]
--    intercalaCero [1..9]          == [1,2,3,4,5,6,7,8,9]
-- ---------------------------------------------------------------------

intercalaCero :: [Int] -> [Int]
intercalaCero []  = []
intercalaCero [x] = [x]
intercalaCero (x:y:xs) | x > y     = x : 0 : intercalaCero (y:xs)
                       | otherwise = x : intercalaCero (y:xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck la siguiente propiedad: para
-- cualquier lista de enteros xs, la longitud de la lista que resulta
-- de intercalar ceros es mayor o igual que la longitud de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_intercalaCero :: [Int] -> Bool
prop_intercalaCero xs =
    length (intercalaCero xs) >= length xs

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Una lista social es una lista de números enteros
-- x_1,...,x_n tales que para cada índice i se tiene que la suma de los
-- divisores  propios de x_i es x_(i+1), para i=1,...,n-1 y la suma de
-- los divisores propios de x_n es x_1. Por ejemplo,
-- [12496,14288,15472,14536,14264] es una lista social.
--
-- Definir la función
--    esListaSocial :: [Int] -> Bool
-- tal que (esListaSocial xs) se verifica si xs es una lista social.
-- Por ejemplo,
--    esListaSocial [12496, 14288, 15472, 14536, 14264] == True
--    esListaSocial [12, 142, 154]                      == False
-- ---------------------------------------------------------------------

esListaSocial :: [Int] -> Bool
esListaSocial xs =
    (and [asociados x y | (x,y) <- zip xs (tail xs)]) &&
    asociados (last xs) (head xs)
    where asociados :: Int -> Int -> Bool
          asociados x y = sum [k | k <- [1..x-1], rem x k == 0] == y

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. ¿Existen listas sociales de un único elemento? Si
-- crees que existen, busca una de ellas.
-- ---------------------------------------------------------------------

listasSocialesUnitarias :: [[Int]]
listasSocialesUnitarias = [[n] | n <- [1..], esListaSocial [n]]

-- El cálculo es
--    ghci> take 4 listasSocialesUnitarias
--    [[6],[28],[496],[8128]]

-- Se observa que [n] es una lista social syss n es un número perfecto.

-- ---------------------------------------------------------------------
-- Ejercicio 4. (Problema 358 del proyecto Euler) Un número x con n
-- cifras se denomina número circular si tiene la siguiente propiedad:
```

```haskell
-- si se multiplica por 1, 2, 3, 4, ..., n, todos los números que
-- resultan tienen exactamente las mismas cifras que x, pero en distinto
-- orden. Por ejemplo, el número 142857 es circular, ya que
--    142857 * 1 = 142857
--    142857 * 2 = 285714
--    142857 * 3 = 428571
--    142857 * 4 = 571428
--    142857 * 5 = 714285
--    142857 * 6 = 857142
--
-- Definir la función
--    esCircular :: Int -> Bool
-- tal que (esCircular x) se verifica si x es circular. Por ejemplo,
--    esCircular 142857 == True
--    esCircular 14285  == False
-- ---------------------------------------------------------------------

esCircular :: Int -> Bool
esCircular x = and [esPermutacionCifras y x | y <- ys]
  where n = numeroDeCifras x
        ys = [k*x | k <- [1..n]]

-- (numeroDeCifras x) es el número de cifras de x. Por ejemplo,
--    numeroDeCifras 142857  ==  6
numeroDeCifras :: Int -> Int
numeroDeCifras = length . cifras

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 142857  ==  [1,4,2,8,5,7]
cifras :: Int -> [Int]
cifras n = [read [x]| x <- show n]

-- (esPermutacion xs ys) se verifica si xs es una permutación de ys. Por
-- ejemplo,
--    esPermutacion [2,5,3] [3,5,2]    ==  True
--    esPermutacion [2,5,3] [2,3,5,2]  ==  False
esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     [] = True
esPermutacion []     _  = False
esPermutacion (x:xs) ys = elem x ys && esPermutacion xs (delete x ys)
```

```
-- (esPermutacion x y) se verifica si las cifras de x es una permutación
-- de las de y. Por ejemplo,
--    esPermutacionCifras 253 352  ==  True
--    esPermutacionCifras 253 2352  ==  False
esPermutacionCifras :: Int -> Int -> Bool
esPermutacionCifras x y =
   esPermutacion (cifras x) (cifras y)
```

## 2.3. Examen 3 (26 de Enero de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 3º examen de evaluación continua (26 de enero de 2012)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    sumatorio :: (Integer -> Integer) -> Integer -> Integer -> Integer
-- tal que (sumatorio f m n) es la suma de f(x) desde x=m hasta x=n. Por
-- ejemplo,
--    sumatorio (^2) 5 10     == 355
--    sumatorio abs (-5) 10   == 70
--    sumatorio (^2) 3 100000 == 333338333349995
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
sumatorioC :: (Integer -> Integer) -> Integer -> Integer -> Integer
sumatorioC f m n = sum [f x | x <- [m..n]]

-- 2ª definición (por recursión):
sumatorioR :: (Integer -> Integer) -> Integer -> Integer -> Integer
sumatorioR f m n = aux m 0
    where aux k ac | k > n     = ac
                   | otherwise = aux (k+1) (ac + f k)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
```

```haskell
--    sumaPred :: Num a => (a -> Bool) -> [a] -> a
-- tal que (sumaPred p xs) es la suma de los elementos de xs que
-- verifican el predicado p. Por ejemplo:
--    sumaPred even [1..1000]   == 250500
--    sumaPred even [1..100000] == 2500050000
-- ------------------------------------------------------------------

-- 1ª definición (por composición, usando funciones de orden superior):
sumaPred :: Num a => (a -> Bool) -> [a] -> a
sumaPred p = sum . filter p

-- 2ª definición (por recursión):
sumaPredR :: Num a => (a -> Bool) -> [a] -> a
sumaPredR _ [] = 0
sumaPredR p (x:xs) | p x       = x + sumaPredR p xs
                   | otherwise = sumaPredR p xs

-- 3ª definición (por plegado por la derecha, usando foldr):
sumaPredPD :: Num a => (a -> Bool) -> [a] -> a
sumaPredPD p = foldr f 0
    where f x y | p x       = x + y
                | otherwise = y

-- 4ª definición (por recursión final):
sumaPredRF :: Num a => (a -> Bool) -> [a] -> a
sumaPredRF p xs = aux xs 0
    where aux []     a = a
          aux (x:xs) a | p x       = aux xs (x+a)
                       | otherwise = aux xs a

-- 5ª definición (por plegado por la izquierda, usando foldl):
sumaPredPI p = foldl f 0
    where f x y | p y       = x + y
                | otherwise = x

-- ------------------------------------------------------------------
-- Ejercicio 2.1. Representamos una relación binaria sobre un conjunto
-- como un par formado por:
--   * una lista, que representa al conjunto, y
--   * una lista de pares, que forman la relación
```

```haskell
-- En los ejemplos usaremos las siguientes relaciones
--    r1, r2, r3,r4 :: ([Int],[(Int, Int)])
--    r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
--    r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
--    r3 = ([1..9],[(1,3), (2,6), (6,2), (3,1), (4,4)])
--    r4 = ([1..3],[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)])
--
-- Definir la función
--    reflexiva :: Eq a => ([a],[(a,a)]) -> Bool
-- tal que (reflexiva r) se verifica si r es una relación reflexiva. Por
-- ejemplo,
--    reflexiva r1 == False
--    reflexiva r4 == True
-- ---------------------------------------------------------------------

r1, r2, r3,r4 :: ([Int],[(Int, Int)])
r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
r3 = ([1..9],[(1,3), (2,6), (6,2), (3,1), (4,4)])
r4 = ([1..3],[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)])

reflexiva :: Eq a => ([a],[(a,a)]) -> Bool
reflexiva (us,ps) = and [(x,x) `elem` ps | x <- us]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    simetrica :: Eq a => ([a],[(a,a)]) -> Bool
-- tal que (simetrica r) se verifica si r es una relación simétrica. Por
-- ejemplo,
--    simetrica r1 == False
--    simetrica r3 == True
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
simetricaC :: Eq a => ([a],[(a,a)]) -> Bool
simetricaC (x,r) =
    null [(x,y) | (x,y) <- r, (y,x) `notElem` r]

-- 2ª definición (por recursión):
simetrica :: Eq a => ([a],[(a,a)]) -> Bool
```

```haskell
simetrica r = aux (snd r)
    where aux [] = True
          aux ((x,y):s) | x == y    = aux s
                        | otherwise = elem (y,x) s && aux (delete (y,x) s)

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Problema 347 del proyecto Euler) El mayor entero menor
-- o igual que 100 que sólo es divisible por los primos 2 y 3, y sólo
-- por ellos, es 96, pues 96 = 3*32 = 3*2^5.
--
-- Dados dos primos distintos p y q, sea M(p,q,n) el mayor entero menor
-- o igual que n sólo divisible por ambos p y q; o M(p,q,N)=0 si tal
-- entero no existe. Por ejemplo:
--    M(2,3,100)  = 96
--    M(3,5,100)  = 75 y no es 90 porque 90 es divisible por 2, 3 y 5
--                     y tampoco es 81 porque no es divisible por 5.
--    M(2,73,100) = 0 porque no existe un entero menor o igual que 100 que
--                    sea divisible por 2 y por 73.
--
-- Definir la función
--    mayorSoloDiv :: Int -> Int -> Int -> Int
-- tal que (mayorSoloDiv p q n) es M(p,q,n). Por ejemplo,
--    mayorSoloDiv 2 3 100  == 96
--    mayorSoloDiv 3 5 100  == 75
--    mayorSoloDiv 2 73 100 == 0
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

mayorSoloDiv :: Int -> Int -> Int -> Int
mayorSoloDiv p q n
    | null xs   = 0
    | otherwise = head xs
    where xs = [x | x <- [n,n-1..1], divisoresPrimos x == sort [p,q]]

-- (divisoresPrimos n) es la lista de los divisores primos de x. Por
-- ejemplo,
--    divisoresPrimos 180  ==  [2,3,5]
divisoresPrimos :: Int -> [Int]
```

```haskell
divisoresPrimos n = [x | x <- [1..n], rem n x == 0, esPrimo x]

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

-- 2ª solución:
-- ============

mayorSoloDiv2 :: Int -> Int -> Int -> Int
mayorSoloDiv2 p q n
    | null xs = 0
    | otherwise = head xs
    where xs = [x | x <- [n,n-1..1], soloDivisible p q x]

-- (soloDivisible p q x) se verifica si x es divisible por los primos p
-- y por q, y sólo por ellos. Por ejemplo,
--    soloDivisible 2 3 96  ==  True
--    soloDivisible 3 5 90  ==  False
--    soloDivisible 3 5 75  ==  True
soloDivisible :: Int -> Int -> Int -> Bool
soloDivisible p q x =
    mod x p == 0 && mod x q == 0 && aux x
    where aux x | x `elem` [p,q] = True
                | mod x p == 0   = aux (div x p)
                | mod x q == 0   = aux (div x q)
                | otherwise      = False

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Dado un número n, calculamos la suma de sus divisores
-- propios reiteradamente hasta que quede un número primo. Por ejemplo,
--
--    n  | divisores propios                    | suma de div. propios
-- ------+--------------------------------------+---------------------
--   30  | [1,2,3,5,6,10,15]                    |  42
--   42  | [1,2,3,6,7,14,21]                    |  54
--   54  | [1,2,3,6,9,18,27]                    |  66
--   66  | [1,2,3,6,11,22,33]                   |  78
```

```
--   78  | [1,2,3,6,13,26,39]                    |  90
--   90  | [1,2,3,5,6,9,10,15,18,30,45]          | 144
--  144  | [1,2,3,4,6,8,9,12,16,18,24,36,48,72]  | 259
--  259  | [1,7,37]                              |  45
--   45  | [1,3,5,9,15]                          |  33
--   33  | [1,3,11]                              |  15
--   15  | [1,3,5]                               |   9
--    9  | [1,3]                                 |   4
--    4  | [1,2]                                 |   3
--    3 (es primo)
--
-- Definir una función
--    sumaDivReiterada :: Int -> Int
-- tal que (sumaDivReiterada n) calcule reiteradamente la suma de los
-- divisores propios hasta que se llegue a un número primo. Por ejemplo,
--    sumaDivReiterada 30   == 3
--    sumaDivReiterada 52   == 3
--    sumaDivReiterada 5289 == 43
--    sumaDivReiterada 1024 == 7
-- ---------------------------------------------------------------------

sumaDivReiterada :: Int -> Int
sumaDivReiterada n
    | esPrimo n   = n
    | otherwise = sumaDivReiterada (sumaDivPropios n)

-- (sumaDivPropios n) es la suma de los divisores propios de n. Por
-- ejemplo,
--    sumaDivPropios 30  ==  42
sumaDivPropios :: Int -> Int
sumaDivPropios n = sum [k | k <- [1..n-1], rem n k == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. ¿Hay números naturales para los que la función
-- anterior no termina? Si crees que los hay, explica por qué y
-- encuentra los tres primeros  números para los que la función anterior
-- no terminaría. En caso contrario, justifica por qué termina siempre.
-- .....................................................................

-- Basta observar que si n es igual a la suma de sus divisores propios
```

```
-- (es decir, si n es un número perfecto), la función no termina porque
-- vuelve a hacer la suma reiterada de sí mismo otra vez. Luego, la
-- función no termina para los números perfectos.

-- Los números perfectos se definen por
esPerfecto :: Int -> Bool
esPerfecto n = sumaDivPropios n == n

-- Los 3 primeros números perfectos se calcula por
--    ghci> take 3 [n | n <- [1..], esPerfecto n]
--    [6,28,496]

-- Por tanto, los tres primeros números para los que el algoritmo no
-- termina son los 6, 28 y 496.

-- Se puede comprobar con
--    ghci> sumaDivReiterada 6
--      C-c C-cInterrupted.
```

## 2.4. Examen 4 ( 1 de Marzo de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 4º examen de evaluación continua (1 de marzo de 2011)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    verificaP :: (a -> Bool) -> [[a]] -> Bool
-- tal que (verificaP p xs) se verifica si cada elemento de la lista xss
-- contiene algún elemento que cumple el predicado p. Por ejemplo,
--    verificaP odd [[1,3,4,2], [4,5], [9]] == True
--    verificaP odd [[1,3,4,2], [4,8], [9]] == False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
verificaP :: (a -> Bool) -> [[a]] -> Bool
verificaP p xss = and [any p xs | xs <- xss]
```

```haskell
-- 2ª definición (por recursión):
verificaP2 :: (a -> Bool) -> [[a]] -> Bool
verificaP2 p []       = True
verificaP2 p (xs:xss) = any p xs && verificaP2 p xss

-- 3ª definición (por plegado):
verificaP3 :: (a -> Bool) -> [[a]] -> Bool
verificaP3 p = foldr ((&&) . any p) True

-- ---------------------------------------------------------------------
-- Ejercicio 2. 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
-- se representa por
--    N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8))
--
-- Definir la función
--    mapArbol :: (Int -> Int) -> Arbol -> Arbol
-- tal que (mapArbol f a) es el árbol que resulta de aplicarle f a los
-- nodos y las hojas de a. Por ejemplo,
--    ghci> mapArbol (^2) (N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8)))
--    N (N (H 1) 81 (H 16)) 25 (N (H 36) 49 (H 64))
-- ---------------------------------------------------------------------

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

mapArbol :: (Int -> Int) -> Arbol -> Arbol
mapArbol f (H x)     = H (f x)
```

```haskell
mapArbol f (N i x d) = N (mapArbol f i) (f x) (mapArbol f d)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    separaSegunP :: (a -> Bool) -> [a] -> [[a]]
-- tal que (separaSegunP p xs) es la lista obtenida separando los
-- elementos de xs en segmentos según que verifiquen o no el predicado
-- p. Por jemplo,
--    ghci> separaSegunP odd [1,2,3,4,5,6,7,8]
--    [[1],[2],[3],[4],[5],[6],[7],[8]]
--    ghci> separaSegunP odd [1,1,3,4,6,7,8,10]
--    [[1,1,3],[4,6],[7],[8,10]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Un número poligonal es un número que puede
-- recomponerse en un polígono regular.
-- Los números triangulares (1, 3, 6, 10, 15, ...) son enteros del tipo
--    1 + 2 + 3 + ... + n.
-- Los números cuadrados (1, 4, 9, 16, 25, ...) son enteros del tipo
--    1 + 3 + 5 + ... + (2n-1).
-- Los números pentagonales (1, 5, 12, 22, ...) son enteros del tipo
--    1 + 4 + 7 + ... + (3n-2).
-- Los números hexagonales (1, 6, 15, 28, ...) son enteros del tipo
--    1 + 5 + 9 + ... + (4n-3).
-- Y así sucesivamente.
--
-- Según Fermat, todo número natural se puede expresar como la suma de n
-- números poligonales de n lados. Gauss lo demostró para los
-- triangulares y Cauchy para todo tipo de polígonos.
--
-- Para este ejercicio, decimos que un número poligonal de razón n es
-- un número del tipo
--    1 + (1+n) + (1+2*n)+...
-- Es decir, los números triángulares son números poligonales de razón
```

```haskell
-- 1, los números cuadrados son números poligonales de razón 2, los
-- pentagonales de razón 3, etc.
--
-- Definir la constante
--    triangulares :: [Integer]
-- tal que es la lista de todos los números triángulares. Por ejemplo,
--    ghci> take 20 triangulares
--    [1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171,190,210]
-- ---------------------------------------------------------------------

triangulares :: [Integer]
triangulares = [sum [1..k] | k <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    esTriangular :: Integer -> Bool
-- tal que (esTriangular n) se verifica si n es un número es triangular.
-- Por ejemplo,
--    esTriangular 253  == True
--    esTriangular 234  == False
-- ---------------------------------------------------------------------

esTriangular :: Integer -> Bool
esTriangular x = x `elem` takeWhile (<=x) triangulares

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    poligonales:: Integer -> [Integer]
-- tal que (poligonales n) es la lista de los números poligonales de
-- razón n. Por ejemplo,
--    take 10 (poligonales 1) == [1,3,6,10,15,21,28,36,45,55]
--    take 10 (poligonales 3) == [1,5,12,22,35,51,70,92,117,145]
-- ---------------------------------------------------------------------

poligonales:: Integer -> [Integer]
poligonales n = [sum [1+j*n | j <- [0..k]] | k <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Definir la función
--    esPoligonalN :: Integer -> Integer -> Bool
```

```haskell
-- tal que (esPoligonalN x n) se verifica si x es poligonal de razón n.
-- Por ejemplo,
--    esPoligonalN 12 3  ==  True
--    esPoligonalN 12 1  ==  False
-- ---------------------------------------------------------------------

esPoligonalN :: Integer -> Integer -> Bool
esPoligonalN x n = x `elem` takeWhile (<= x) (poligonales n)

-- ---------------------------------------------------------------------
-- Ejercicio 4.5. Definir la función
--    esPoligonal :: Integer -> Bool
-- tal que (esPoligonalN x) se verifica si x es un número poligonal. Por
-- ejemplo,
--    esPoligonal 12  ==  True
-- ---------------------------------------------------------------------

esPoligonal :: Integer -> Bool
esPoligonal x = or [esPoligonalN x n | n <- [1..x]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.6. Calcular el primer número natural no poligonal.
-- ---------------------------------------------------------------------

primerNoPoligonal :: Integer
primerNoPoligonal = head [x | x <- [1..], not (esPoligonal x)]

-- El cálculo es
--    ghci> primerNoPoligonal
--    2

-- ---------------------------------------------------------------------
-- Ejercicio 4.7. Definir la función
--    descomposicionTriangular :: Integer -> (Integer, Integer, Integer)
-- tal que que (descomposicionTriangular n) es la descomposición de un
-- número natural en la suma de, a lo sumo 3 números triángulares. Por
-- ejemplo,
--    descomposicionTriangular 20  == (0,10,10)
--    descomposicionTriangular 206 == (1,15,190)
--    descomposicionTriangular 6   == (0,0,6)
```

```
--    descomposicionTriangular 679 == (1,300,378)
-- ---------------------------------------------------------------------

descomposicionTriangular :: Integer -> (Integer, Integer, Integer)
descomposicionTriangular n =
  head [(x,y,z) | x <- xs,
                  y <- x : dropWhile (<x) xs,
                  z <- y : dropWhile (<y) xs,
                  x+y+z == n]
    where xs = 0 : takeWhile (<=n) triangulares
```

## 2.5. Examen 5 (22 de Marzo de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 5º examen de evaluación continua (22 de marzo de 2012)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List
import PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    interseccionC :: Eq a => [[a]] -> [a]
-- tal que (interseccionC xss) es la lista con los elementos comunes a
-- todas las listas de xss. Por ejemplo,
--    interseccionC [[1,2],[3]] == []
--    interseccionC [[1,2],[3,2], [2,4,5,6,1]] == [2]
-- ---------------------------------------------------------------------

interseccionC :: Eq a => [[a]] -> [a]
interseccionC []   = []
interseccionC (xs:xss) = [x | x <- xs, and [x `elem` ys| ys <- xss]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recurción, la función
--    interseccionR :: Eq a => [[a]] -> [a]
-- tal que (interseccionR xss) es la lista con los elementos comunes a
-- todas las listas de xss. Por ejemplo,
--    interseccionR [[1,2],[3]] == []
```

```
--    interseccionR [[1,2],[3,2], [2,4,5,6,1]] == [2]
-- ---------------------------------------------------------------------

interseccionR :: Eq a => [[a]] -> [a]
interseccionR [xs]     = xs
interseccionR (xs:xss) = inter xs (interseccionR xss)
    where inter xs ys = [x | x <- xs, x `elem` ys]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    primerComun :: Ord a => [a] -> [a] -> a
-- tal que (primerComun xs ys) el primer elemento común de las listas xs
-- e ys (suponiendo que ambas son crecientes y, posiblemente,
-- infinitas). Por ejemplo,
--    primerComun [2,4..] [7,10..] == 10
-- ---------------------------------------------------------------------

primerComun :: Ord a => [a] -> [a] -> a
primerComun xs ys = head [x | x <- xs, x `elem` takeWhile (<=x) ys]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, utilizando la función anterior, la función
--    mcm :: Int -> Int -> Int
-- tal que (mcm x y) es el mínimo común múltiplo de x e y. Por ejemplo,
--    mcm 123 45  == 1845
--    mcm 123 450 == 18450
--    mcm 35 450  == 3150
-- ---------------------------------------------------------------------

mcm :: Int -> Int -> Int
mcm x y = primerComun [x*k | k <- [1..]] [y*k | k <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Consideremos el TAD de los polinomios visto en
-- clase. Como ejemplo, tomemos el polinomio x^3 + 3.0*x^2 - 1.0*x - 2.0,
-- definido por
--    ejPol :: Polinomio Float
--    ejPol = consPol 3 1
--                    (consPol 2 3
--                             (consPol 1 (-1)
```

```
--                                        (consPol 0 (-2) polCero)))
--
-- Definir la función
--    integral :: Polinomio Float -> Polinomio Float
-- tal que (integral p) es la integral del polinomio p. Por ejemplo,
--    integral ejPol == 0.25*x^4 + x^3 + -0.5*x^2 -2.0*x
-- ---------------------------------------------------------------------

ejPol :: Polinomio Float
ejPol = consPol 3 1
                (consPol 2 3
                         (consPol 1 (-1)
                                  (consPol 0 (-2) polCero)))
integral :: Polinomio Float -> Polinomio Float
integral p
    | esPolCero p = polCero
    | otherwise   = consPol (n+1) (b/fromIntegral (n+1)) (integral r)
    where n = grado p
          b = coefLider p
          r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    integralDef :: Polinomio Float -> Float-> Float -> Float
-- tal que (integralDef p a b) es el valor de la integral definida
-- de p entre a y b. Por ejemplo,
--    integralDef ejPol 1 4 == 113.25
-- ---------------------------------------------------------------------

integralDef :: Polinomio Float -> Float -> Float -> Float
integralDef p a b = valor q b - valor q a
    where q = integral p

-- ---------------------------------------------------------------------
-- Ejercicio 4. El método de la bisección para calcular un cero de una
-- función en el intervalo [a,b] se basa en el teorema de Bolzano:
--    "Si f(x) es una función continua en el intervalo [a, b], y si,
--    además, en los extremos del intervalo la función f(x) toma valores
--    de signo opuesto (f(a) * f(b) < 0), entonces existe al menos un
--    valor c en (a, b) para el que f(c) = 0".
```

```
--
-- La idea es tomar el punto medio del intervalo c = (a+b)/2 y
-- considerar los siguientes casos:
-- * Si f(c) ~= 0, hemos encontrado una aproximación del punto que
--   anula f en el intervalo con un error aceptable.
-- * Si f(c) tiene signo distinto de f(a), repetir el proceso en el
--   intervalo [a,c].
-- * Si no, repetir el proceso en el intervalo [c,b].
--
-- Definir la función
--    ceroBiseccionE :: (Float -> Float) -> Float -> Float -> Float -> Float
-- tal que (ceroBiseccionE f a b e) es una aproximación del punto
-- del intervalo [a,b] en el que se anula la función f, con un error
-- menor que e, aplicando el método de la bisección (se supone que
-- f(a)*f(b)<0). Por ejemplo,
--    let f1 x = 2 - x
--    let f2 x = x^2 - 3
--    ceroBiseccionE f1 0 3 0.0001     == 2.000061
--    ceroBiseccionE f2 0 2 0.0001     == 1.7320557
--    ceroBiseccionE f2 (-2) 2 0.00001 == -1.732048
--    ceroBiseccionE cos 0 2 0.0001    == 1.5708008
-- ---------------------------------------------------------------------

ceroBiseccionE :: (Float -> Float) -> Float -> Float -> Float -> Float
ceroBiseccionE f a b e = aux a b
    where aux c d | aceptable m     = m
                  | f c * f m  < 0  = aux c m
                  | otherwise       = aux m d
              where m = (c+d)/2
                    aceptable x = abs (f x) < e
```

## 2.6. Examen 6 ( 3 de Mayo de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 6º examen de evaluación continua (3 de mayo de 2012)
-- ---------------------------------------------------------------------

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

```haskell
import PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    verificaP :: (a -> Bool) -> [[a]] -> Bool
-- tal que (verificaP p xss) se cumple si cada lista de xss contiene
-- algún elemento que verifica el predicado p. Por ejemplo,
--    verificaP odd [[1,3,4,2], [4,5], [9]] == True
--    verificaP odd [[1,3,4,2], [4,8], [9]] == False
-- ---------------------------------------------------------------------

verificaP :: (a -> Bool) -> [[a]] -> Bool
verificaP p xss = and [any p xs | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    verificaTT :: (a -> Bool) -> [[a]] -> Bool
-- tal que (verificaTT p xss) se cumple si todos los elementos de todas
-- las listas de xss verifican el predicado p. Por ejemplo,
--    verificaTT odd [[1,3], [7,5], [9]]     == True
--    verificaTT odd [[1,3,4,2], [4,8], [9]] == False
-- ---------------------------------------------------------------------

verificaTT :: (a -> Bool) -> [[a]] -> Bool
verificaTT p xss = and [all p xs | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    verificaEE :: (a -> Bool) -> [[a]] -> Bool
-- tal que (verificaEE p xss) se cumple si algún elemento de alguna
-- lista de xss verifica el predicado p. Por ejemplo,
--    verificaEE odd [[1,3,4,2], [4,8], [9]] == True
--    verificaEE odd [[4,2], [4,8], [10]]    == False
-- ---------------------------------------------------------------------

verificaEE :: (a -> Bool) -> [[a]] -> Bool
verificaEE p xss = or [any p xs | xs <- xss]

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

```
--    verificaET :: (a -> Bool) -> [[a]] -> Bool
-- tal que (verificaET p xss) se cumple si todos los elementos de alguna
-- lista de xss verifican el predicado p. Por ejemplo,
--    verificaET odd [[1,3], [4,8], [10]] == True
--    verificaET odd [[4,2], [4,8], [10]] == False
-- ---------------------------------------------------------------------

verificaET :: (a -> Bool) -> [[a]] -> Bool
verificaET p xss = or [all p xs | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 2. (Problema 303 del proyecto Euler). Dado un número
-- natural n, se define  f(n) como el menor natural, múltiplo de n,
-- cuyos dígitos son todos menores o iguales que 2. Por ejemplo, f(2)=2,
-- f(3)=12, f(7)=21, f(42)=210, f(89)=1121222.
--
-- Definir la función
--    menorMultiplo1y2 :: Int -> Int
-- tal que (menorMultiplo1y2 n) es el menor múltiplo de n cuyos dígitos
-- son todos menores o iguales que 2. Por ejemplo,
--    menorMultiplo1y2 42  ==  210
-- ---------------------------------------------------------------------

menorMultiplo1y2 :: Int -> Int
menorMultiplo1y2 n =
    head [x | x <- [n,2*n..], all (<=2) (cifras x)]

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 325  ==  [3,2,5]
cifras :: Int -> [Int]
cifras n = [read [x]| x <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    raicesApol :: Fractional t => [(t, Int)] -> Polinomio t
-- tal que (raicesApol rs) es el polinomio correspondiente si rs es la
-- lista de raices, con sus respectivas multiplicidades, rs; es decir,
-- raicesApol [(r1,n1),...,(rk,nk)] es (x-r1)^n1...(x-rk)^nk. Por
-- ejemplo,
--   raicesApol [(2,1),(-1,3)] == x^4 + x^3 + -3.0*x^2 + -5.0*x + -2.0
```

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

raicesApol :: Fractional t => [(t, Int)] -> Polinomio t
raicesApol rs = multListaPol factores
    where factores = [potencia (creaFactor x) n | (x,n) <- rs]

-- (creaFactor a) es el polinomio x-a. Por ejemplo,
--    ghci> creaFactor 5
--    1.0*x + -5.0
creaFactor :: Fractional t => t -> Polinomio t
creaFactor a = creaPolDensa [(1,1),(0,-a)]

-- (creaPolDensa ps) es el polinomio cuya representación densa (mediante
-- pares con grados y coeficientes) es ps. Por ejemplo,
--    ghci> creaPolDensa [(3,5),(2,4),(0,7)]
--    5*x^3 + 4*x^2 + 7
creaPolDensa :: Num a => [(Int,a)] -> Polinomio a
creaPolDensa []         = polCero
creaPolDensa ((n,a):ps) = consPol n a (creaPolDensa ps)

-- (potencia p n) es la n-ésima potencia de P. Por ejemplo,
--    ghci> potencia (creaFactor 5) 2
--    x^2 + -10.0*x + 25.0
potencia :: Num a => Polinomio a -> Int -> Polinomio a
potencia p 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))

-- (multListaPol ps) es el producto de los polinomios de la lista
-- ps. Por ejemplo,
--    ghci> multListaPol [creaFactor 2, creaFactor 3, creaFactor 4]
--    x^3 + -9.0*x^2 + 26.0*x + -24.0
multListaPol :: Num t => [Polinomio t] -> Polinomio t
multListaPol []     = polUnidad
multListaPol (p:ps) = multPol p (multListaPol ps)

-- multListaPol se puede definir por plegado:
multListaPol' :: Num t => [Polinomio t] -> Polinomio t
multListaPol' = foldr multPol polUnidad

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

```haskell
-- Ejercicio 4.1. Consideremos el tipo de los vectores y las matrices
-- definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
--
-- Definir la función
--    esEscalar:: Num a => Matriz a -> Bool
-- tal que (esEscalar p) se verifica si p es una matriz es escalar; es
-- decir, diagonal con todos los elementos de la diagonal principal
-- iguales. Por ejemplo,
--    esEscalar (listArray ((1,1),(3,3)) [5,0,0,0,5,0,0,0,5])  ==  True
--    esEscalar (listArray ((1,1),(3,3)) [5,0,0,1,5,0,0,0,5])  ==  False
--    esEscalar (listArray ((1,1),(3,3)) [5,0,0,0,6,0,0,0,5])  ==  False
-- ----------------------------------------------------------------------

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

esEscalar:: Num a => Matriz a -> Bool
esEscalar p = esDiagonal p && todosIguales (elems (diagonalPral p))

-- (esDiagonal p) se verifica si la matriz p es diagonal. Por ejemplo.
--    esDiagonal (listArray ((1,1),(3,3)) [5,0,0,0,6,0,0,0,5])  ==  True
--    esDiagonal (listArray ((1,1),(3,3)) [5,0,0,1,5,0,0,0,5])  ==  False
esDiagonal:: Num a => Matriz a -> Bool
esDiagonal p = all (==0) [p!(i,j) | i<-[1..m],j<-[1..n], i/=j]
    where (m,n) = dimension p

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    todosIguales [5,5,5]  ==  True
--    todosIguales [5,6,5]  ==  False
todosIguales :: Eq a => [a] -> Bool
todosIguales (x:y:ys) = x == y && todosIguales (y:ys)
todosIguales _ = True

-- (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
--    ghci> diagonalPral (listArray ((1,1),(3,3)) [5,0,0,1,6,0,0,2,4])
--    array (1,3) [(1,5),(2,6),(3,4)]
```

```haskell
diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)

-- (numFilas p) es el número de filas de la matriz p. Por ejemplo,
--    numFilas (listArray ((1,1),(2,3)) [5,0,0,1,6,0])  ==  2
numFilas :: Num a => Matriz a -> Int
numFilas = fst . snd . bounds

-- (numColumnas p) es el número de columnas de la matriz p. Por ejemplo,
--    numColumnas (listArray ((1,1),(2,3)) [5,0,0,1,6,0])  ==  3
numColumnas:: Num a => Matriz a -> Int
numColumnas = snd . snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. 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
    | dimension p == (1,1) = p!(1,1)
    | otherwise =
        sum [((-1)^(i+1))*(p!(i,1))*determinante (submatriz i 1 p)
             | i <- [1..numFilas p]]

-- (dimension p) es la dimensión de la matriz p. Por ejemplo,
--    dimension (listArray ((1,1),(2,3)) [5,0,0,1,6,0])  ==  (2,3)
dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas 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) = dimension p
          f k l | k < i  && l < j  = (k,l)
                | k >= i && l < j  = (k+1,l)
                | k < i  && l >= j = (k,l+1)
                | otherwise        = (k+1,l+1)
```

## 2.7. Examen 7 (24 de Junio de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 7º examen de evaluación continua (24 de junio de 2012)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import Data.Ratio
import Test.QuickCheck
import PolOperaciones
import GrafoConVectorDeAdyacencia

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    duplicaElemento :: Eq a => a -> [a] -> [a]
-- tal que (duplicaElemento x ys) es la lista obtenida duplicando las
-- apariciones del elemento x en la lista ys. Por ejemplo,
--    duplicaElemento 7 [2,7,3,7,7,5]  ==  [2,7,7,3,7,7,7,7,5]
-- ---------------------------------------------------------------------

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    listaAcumulada :: Num t => [t] -> [t]
-- tal que (listaAcumulada xs) es la lista obtenida sumando de forma
-- acumulada los elementos de xs. Por ejemplo,
--    listaAcumulada [1..4] == [1,3,6,10]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
listaAcumulada :: Num t => [t] -> [t]
listaAcumulada xs = [sum (take n xs) | n <- [1..length xs]]

-- 2ª definición (por recursión):
listaAcumuladaR [] = []
listaAcumuladaR xs = listaAcumuladaR (init xs) ++ [sum xs]

-- 3ª definición (por recursión final)
listaAcumuladaRF [] = []
listaAcumuladaRF (x:xs) = reverse (aux xs [x])
    where aux [] ys = ys
          aux (x:xs) (y:ys) = aux xs (x+y:y:ys)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que el último elemento de
-- (listaAcumulada xs) coincide con la suma de los elemntos de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_listaAcumulada :: [Int] -> Property
prop_listaAcumulada xs =
    not (null xs) ==> last (listaAcumulada xs) == sum xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    menorP :: (Int -> Bool) -> Int
```

```haskell
-- tal que (menorP p) es el menor número natural que verifica el
-- predicado p. Por ejemplo,
--    menorP (>7)   ==  8
-- ---------------------------------------------------------------------

menorP :: (Int -> Bool) -> Int
menorP p = head [n | n <- [0..], p n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    menorMayorP :: Int -> (Int -> Bool) -> Int
-- tal que (menorMayorP m p) es el menor número natural mayor que m que
-- verifica el predicado p. Por ejemplo,
--    menorMayorP 7 (\x -> rem x 5 == 0)   ==  10
-- ---------------------------------------------------------------------

menorMayorP :: Int -> (Int -> Bool) -> Int
menorMayorP m p = head [n | n <- [m+1..], p n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    mayorMenorP :: Int -> (Int -> Bool) -> Int
-- tal que (mayorMenorP p) es el mayor entero menor que m que verifica
-- el predicado p. Por ejemplo,
--    mayorMenorP 17 (\x -> rem x 5 == 0)   ==  15
-- ---------------------------------------------------------------------

mayorMenorP :: Int -> (Int -> Bool) -> Int
mayorMenorP m p = head [n | n <- [m-1,m-2..], p n]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    polNumero :: Int -> Polinomio Int
-- tal que (polNumero n) es el polinomio cuyos coeficientes son las
-- cifras de n. Por ejemplo,
--   polNumero 5703 == 5x^3 + 7x^2 + 3
-- ---------------------------------------------------------------------

polNumero :: Int -> Polinomio Int
polNumero = creaPolDispersa . cifras
```

```haskell
-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 142857  ==  [1,4,2,8,5,7]
cifras :: Int -> [Int]
cifras n = [read [x]| x <- show n]

-- (creaPolDispersa xs) es el polinomio cuya representación dispersa es
-- xs. Por ejemplo,
--    creaPolDispersa [7,0,0,4,0,3]  ==  7*x^5 + 4*x^2 + 3
creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
creaPolDispersa []     = polCero
creaPolDispersa (x:xs) = consPol (length xs) x (creaPolDispersa xs)

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Los vectores se definen por
--    type Vector = Array Int Float
--
-- Un vector se denomina estocástico si todos sus elementos son mayores
-- o iguales que 0 y suman 1.
--
-- Definir la función
--    vectorEstocastico :: Vector -> Bool
-- tal que (vectorEstocastico v) se verifica si v es estocástico. Por
-- ejemplo,
--    vectorEstocastico (listArray (1,5) [0.1, 0.2, 0, 0, 0.7]) == True
--    vectorEstocastico (listArray (1,5) [0.1, 0.2, 0, 0, 0.9]) == False
-- ---------------------------------------------------------------------

type Vector = Array Int Float

vectorEstocastico :: Vector -> Bool
vectorEstocastico v = all (>=0) xs && sum xs == 1
    where xs = elems v

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Las matrices se definen por
--    type Matriz = Array (Int,Int) Float
--
-- Una matriz se demonina estocástica si sus columnas son vectores
-- estocásticos.
```

```
--
-- Definir la función
--    matrizEstocastica :: Matriz -> Bool
-- tal que (matrizEstocastico p) se verifica si p es estocástica. Por
-- ejemplo,
--    matrizEstocastica (listArray ((1,1),(2,2)) [0.1,0.2,0.9,0.8]) == True
--    matrizEstocastica (listArray ((1,1),(2,2)) [0.1,0.2,0.3,0.8]) == False
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Float

matrizEstocastica :: Matriz -> Bool
matrizEstocastica p = all vectorEstocastico (columnas p)

-- (columnas p) es la lista de las columnas de la matriz p. Por ejemplo,
--    ghci> columnas (listArray ((1,1),(2,3)) [1..6])
--    [array (1,2) [(1,1.0),(2,4.0)],
--     array (1,2) [(1,2.0),(2,5.0)],
--     array (1,2) [(1,3.0),(2,6.0)]]
--    ghci> columnas (listArray ((1,1),(3,2)) [1..6])
--    [array (1,3) [(1,1.0),(2,3.0),(3,5.0)],
--     array (1,3) [(1,2.0),(2,4.0),(3,6.0)]]
columnas :: Matriz -> [Vector]
columnas p =
    [array (1,m) [(i,p!(i,j)) | i <- [1..m]] | j <- [1..n]]
    where (_,(m,n)) = bounds p

-- ---------------------------------------------------------------------
-- Ejercicio 6. 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 número de nodos 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
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  ==  3
--    anchuraN g 2  ==  3
--    anchuraN g 3  ==  3
--    anchuraN g 4  ==  3
--    anchuraN g 5  ==  4
anchuraN :: Grafo Int Int -> Int -> Int
anchuraN g x = length (adyacentes g x)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Un número natural par es admisible si es una potencia de
-- 2 o sus distintos factores primos son primos consecutivos. Los
-- primeros números admisibles son 2, 4, 6, 8, 12, 16, 18, 24, 30, 32,
-- 36, 48,...
--
-- Definir la constante
--    admisibles :: [Integer]
-- que sea la lista de los números admisibles. Por ejemplo,
--    take 12 admisibles  ==  [2,4,6,8,12,16,18,24,30,32,36,48]
-- ---------------------------------------------------------------------

admisibles:: [Integer]
admisibles = [n | n <- [2,4..], esAdmisible n]

-- (esAdmisible n) se verifica si n es admisible. Por ejemplo,
--    esAdmisible 32  ==  True
```

```haskell
--    esAdmisible 48  ==  True
--    esAdmisible 15  ==  False
--    esAdmisible 10  ==  False
esAdmisible :: Integer -> Bool
esAdmisible n =
    even n &&
    (esPotenciaDeDos n || primosConsecutivos (nub (factorizacion n)))

-- (esPotenciaDeDos n) se verifica si n es una potencia de 2. Por ejemplo,
--    esPotenciaDeDos 4  ==  True
--    esPotenciaDeDos 5  ==  False
esPotenciaDeDos :: Integer -> Bool
esPotenciaDeDos 1 = True
esPotenciaDeDos n = even n && esPotenciaDeDos (n `div` 2)

-- (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 16  ==  2
--    menorFactor 17  == 17
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- (primosConsecutivos xs) se verifica si xs es una lista de primos
-- consecutivos. Por ejemplo,
--    primosConsecutivos [17,19,23]  ==  True
--    primosConsecutivos [17,19,29]  ==  False
--    primosConsecutivos [17,19,20]  ==  False
primosConsecutivos :: [Integer] -> Bool
primosConsecutivos [] = True
primosConsecutivos (x:xs) =
    take (1 + length xs) (dropWhile (<x) primos) == x:xs
```

```haskell
-- primos es la lista de los números primos. Por ejemplo,
--    take 10 primos  ==  [2,3,5,7,11,13,17,19,23,29]
primos :: [Integer]
primos = 2 : [n | n <- [3,5..], esPrimo n]

-- (esPrimo n) se verifica si n es primo.
esPrimo :: Integer-> Bool
esPrimo n = [x | x <- [1..n], rem n x == 0] == [1,n]
```

## 2.8. Examen 8 (29 de Junio de 2012)

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

## 2.9. Examen 9 ( 9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 32).

## 2.10. Examen 10 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 36).

# 3

# Exámenes del grupo 3

Antonia M. Chávez

## 3.1. Examen 1 (14 de Noviembre de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 1º examen de evaluación continua (14 de noviembre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función posicion tal que (posicion x ys) es
-- la primera posición de x en ys. Por ejemplo,
--    posicion 5 [3,4,5,6,5]  ==  2
-- ---------------------------------------------------------------------

posicion x xs = head [i |( c,i) <- zip xs [0..], c == x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función impares tal que (impares xs) es la
-- lista de los elementos de xs que ocupan las posiciones impares. Por
-- ejemplo,
--    impares [4,3,5,2,6,1] == [3,2,1]
--    impares [5]           == []
--    impares []            == []
-- ---------------------------------------------------------------------

impares xs = [x | x <- xs, odd (posicion x xs)]

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

```haskell
-- Ejercicio 2.2. Definir la función pares tal que (pares xs) es la
-- lista de los elementos de xs que ocupan las posiciones pares. Por
-- ejemplo,
--    pares [4,3,5,2,6,1] == [4,5,6]
--    pares [5]           == [5]
--    pares []            == []
-- ---------------------------------------------------------------------

pares xs = [x | x <- xs, even (posicion x xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función separaPorParidad tal que
-- (separaPorParidad xs) es el par cuyo primer elemento es la lista de
-- los elementos de xs que ocupan las posiciones pares y el segundo es
-- la lista de los que ocupan las posiciones impares. Por ejemplo,
--    separaPorParidad [7,5,6,4,3] == ([7,6,3],[5,4])
--    separaPorParidad [4,3,5]     == ([4,5],[3])
-- ---------------------------------------------------------------------

separaPorParidad xs = (pares xs, impares xs)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función eliminaElemento tal que
-- (eliminaElemento xs n) es la lista que resulta de eliminar el n-ésimo
-- elemento de la lista xs. Por ejemplo,
--    eliminaElemento [1,2,3,4,5] 0    == [2,3,4,5]
--    eliminaElemento [1,2,3,4,5] 2    == [1,2,4,5]
--    eliminaElemento [1,2,3,4,5] (-1) == [1,2,3,4,5]
--    eliminaElemento [1,2,3,4,5] 7    == [1,2,3,4,5]
-- ---------------------------------------------------------------------

-- 1ª definición:
eliminaElemento xs n = take n xs ++ drop (n+1)xs

-- 2ª definición:
eliminaElemento2 xs n = [x| x <- xs, posicion x xs /= n]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir por comprensión, usando la lista [1..10], las
-- siguientes listas
```

```
--    l1 = [2,4,6,8,10]
--    l2 = [[1],[3],[5],[7],[9]]
--    l3 = [(1,2),(2,3),(3,4),(4,5),(5,6)]
-- ---------------------------------------------------------------------

l1 = [x | x <- [1..10], even x]

l2 = [[x] | x <- [1..10], odd x]

l3 = [(x,y)| (x,y) <- zip [1..10] (tail [1 .. 10]), x <= 5]
```

## 3.2. Examen 2 (12 de Diciembre de 2011)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 2º examen de evaluación continua (12 de diciembre de 2011)
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión, la función
--    filtraAplicaR :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaR f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaR (4+) (< 3) [1..7] == [5,6]
-- ---------------------------------------------------------------------

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

```haskell
filtraAplicaR _ _ [] = []
filtraAplicaR f p (x:xs) | p x       = f x : filtraAplicaR f p xs
                         | otherwise = filtraAplicaR f p xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Define la función
--    masDeDos :: Eq a => a -> [a] -> Bool
-- tal que (masDeDos x ys) se verifica si x aparece más de dos veces en
-- ys. Por ejemplo,
--    masDeDos 1 [2,1,3,1,4,1,1] == True
--    masDeDos 1 [1,1,2,3]       == False
-- ---------------------------------------------------------------------

masDeDos :: Eq a => a -> [a] -> Bool
masDeDos x ys = ocurrencias x ys > 2

-- (ocurrencias x ys) es el número de ocurrencias de x en ys. Por
-- ejemplo,
--    ocurrencias 1 [2,1,3,1,4,1,1]  ==  4
--    ocurrencias 1 [1,1,2,3]        ==  2
ocurrencias :: Eq a => a -> [a] -> Int
ocurrencias x ys = length [y | y <- ys, y == x]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    sinMasDeDos :: Eq a => [a] -> [a]
-- tal que (sinMasDeDos xs) es la la lista que resulta de eliminar en xs los
-- elementos muy repetidos, dejando que aparezcan dos veces a lo
-- sumo. Por ejemplos,
--    sinMasDeDos [2,1,3,1,4,1,1] == [2,3,4,1,1]
--    sinMasDeDos [1,1,2,3,2,2,5] == [1,1,3,2,2,5]
-- ---------------------------------------------------------------------

sinMasDeDos :: Eq a => [a] -> [a]
sinMasDeDos [] = []
sinMasDeDos (y:ys) | masDeDos y (y:ys) = sinMasDeDos ys
                   | otherwise         = y : sinMasDeDos ys

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
```

```
--    sinRepetidos :: Eq a => [a] -> [a]
-- tal que (sinRepetidos xs) es la lista que resulta de quitar todos los
-- elementos repetidos de xs. Por ejemplo,
--    sinRepetidos [2,1,3,2,1,3,1]   ==  [2,3,1]
-- ---------------------------------------------------------------------

sinRepetidos :: Eq a => [a] -> [a]
sinRepetidos [] = []
sinRepetidos (x:xs) | x `elem` xs = sinRepetidos xs
                    | otherwise   = x : sinRepetidos xs

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    repetidos :: Eq a => [a] -> [a]
-- tal que (repetidos xs) es la lista de los elementos repetidos de
-- xs. Por ejemplo,
--    repetidos [1,3,2,1,2,3,4] == [1,3,2]
--    repetidos [1,2,3]         == []
-- ---------------------------------------------------------------------

repetidos :: Eq a => [a] -> [a]
repetidos [] = []
repetidos (x:xs) | x `elem` xs = x : repetidos xs
                 | otherwise   =  repetidos xs

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck que si una lista xs no tiene
-- elementos repetidos, entonces (sinMasDeDos xs) y (sinRepetidos xs)
-- son iguales.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_limpia :: [Int] -> Property
prop_limpia xs =
    null (repetidos xs) ==> sinMasDeDos xs == sinRepetidos xs

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir, por recursión, la función
--    seleccionaElementoPosicionR :: Eq a => a -> Int -> [[a]]-> [[a]]
-- (seleccionaElementoPosicionR x n xss) es la lista de elementos de xss
-- en las que x aparece en la posición n. Por ejemplo,
--    ghci> seleccionaElementoPosicionR 'a' 1 [”casa”,”perro”, ”bajo”]
--    [”casa”,”bajo”]
-- ---------------------------------------------------------------------

seleccionaElementoPosicionR :: Eq a => a -> Int -> [[a]]-> [[a]]
seleccionaElementoPosicionR x n []= []
seleccionaElementoPosicionR x n (xs:xss)
    | ocurreEn x n xs = xs : seleccionaElementoPosicionR x n xss
    | otherwise       = seleccionaElementoPosicionR x n xss

-- (ocurreEn x n ys) se verifica si x ocurre en ys en la posición n. Por
-- ejemplo,
--    ocurreEn 'a' 1 ”casa”  ==  True
--    ocurreEn 'a' 2 ”casa”  ==  False
--    ocurreEn 'a' 7 ”casa”  ==  False
ocurreEn :: Eq a => a -> Int -> [a] -> Bool
ocurreEn x n ys = 0 <= n && n < length ys && ys!!n == x

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Definir, por comprensión, la función
--    seleccionaElementoPosicionC :: Eq a => a -> Int -> [[a]]-> [[a]]
-- (seleccionaElementoPosicionC x n xss) es la lista de elementos de xss
-- en las que x aparece en la posición n. Por ejemplo,
--    ghci> seleccionaElementoPosicionC 'a' 1 [”casa”,”perro”, ”bajo”]
--    [”casa”,”bajo”]
-- ---------------------------------------------------------------------

seleccionaElementoPosicionC :: Eq a => a -> Int -> [[a]]-> [[a]]
seleccionaElementoPosicionC x n xss =
    [xs | xs <- xss,  ocurreEn x n xs]
```

## 3.3. Examen 7 (29 de Junio de 2012)

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

## 3.4. Examen 8 (9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 32).

## 3.5. Examen 9 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 36).

# 4

# Exámenes del grupo 4

José F. Quesada

## 4.1. Examen 1 ( 7 de Noviembre de 2011)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    conjuntosIguales :: Eq a => [a] -> [a] -> Bool
-- tal que (conjuntosIguales xs ys) se verifica si xs e ys contienen los
-- mismos elementos independientemente del orden y posibles
-- repeticiones. Por ejemplo,
--    conjuntosIguales [1,2,3] [2,3,1]               == True
--    conjuntosIguales "arroz" "zorra"               == True
--    conjuntosIguales [1,2,2,3,2,1] [1,3,3,2,1,3,2] == True
--    conjuntosIguales [1,2,2,1] [1,2,3,2,1]         == False
--    conjuntosIguales [(1,2)] [(2,1)]               == False
-- ---------------------------------------------------------------------

conjuntosIguales :: Eq a => [a] -> [a] -> Bool
conjuntosIguales xs ys =
    and [x `elem` ys | x <- xs] && and [y `elem` xs | y <- ys]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    puntoInterior :: (Float,Float) -> Float -> (Float,Float) -> Bool
```

```haskell
-- tal que (puntoInterior c r p) se verifica si p es un punto interior
-- del círculo de centro c y radio r. Por ejemplo,
--    puntoInterior (0,0) 1 (1,0)   == True
--    puntoInterior (0,0) 1 (1,1)   == False
--    puntoInterior (0,0) 2 (-1,-1) == True
-- ---------------------------------------------------------------------

puntoInterior :: (Float,Float) -> Float -> (Float,Float) -> Bool
puntoInterior (cx,cy) r (px,py) = distancia (cx,cy) (px,py) <= r

-- (distancia p1 p2) es la distancia del punto p1 al p2. Por ejemplo,
--    distancia (0,0) (3,4)  ==  5.0
distancia :: (Float,Float) -> (Float,Float) -> Float
distancia (x1,y1) (x2,y2) = sqrt ( (x2-x1)^2 + (y2-y1)^2)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    tripletes :: Int -> [(Int,Int,Int)]
-- tal que (tripletes n) es la lista de tripletes (tuplas de tres
-- elementos) con todas las combinaciones posibles de valores numéricos
-- entre 1 y n en cada posición del triplete, pero de forma que no haya
-- ningún valor repetido dentro de cada triplete. Por ejemplo,
--    tripletes 3 == [(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)]
--    tripletes 4 == [(1,2,3),(1,2,4),(1,3,2),(1,3,4),(1,4,2),(1,4,3),
--                    (2,1,3),(2,1,4),(2,3,1),(2,3,4),(2,4,1),(2,4,3),
--                    (3,1,2),(3,1,4),(3,2,1),(3,2,4),(3,4,1),(3,4,2),
--                    (4,1,2),(4,1,3),(4,2,1),(4,2,3),(4,3,1),(4,3,2)]
--    tripletes 2 == []
-- ---------------------------------------------------------------------

tripletes :: Int -> [(Int,Int,Int)]
tripletes n = [(x,y,z) | x <- [1..n],
                         y <- [1..n],
                         z <- [1..n],
                         x /= y,
                         x /= z,
                         y /= z]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Las bases de datos de alumnos matriculados por
```

```
-- provincia y por especialidad se pueden representar como sigue
--    matriculas :: [(String,String,Int)]
--    matriculas = [("Almeria","Matematicas",27),
--                  ("Sevilla","Informatica",325),
--                  ("Granada","Informatica",296),
--                  ("Huelva","Matematicas",41),
--                  ("Sevilla","Matematicas",122),
--                  ("Granada","Matematicas",131),
--                  ("Malaga","Informatica",314)]
-- Es decir, se indica que por ejemplo en Almería hay 27 alumnos
-- matriculados en Matemáticas.
--
-- Definir la función
--    totalAlumnos :: [(String,String,Int)] -> Int
-- tal que (totalAlumnos bd) es el total de alumnos matriculados,
-- incluyendo todas las provincias y todas las especialidades, en la
-- base de datos bd. Por ejemplo,
--    totalAlumnos matriculas == 1256
-- ---------------------------------------------------------------------

matriculas :: [(String,String,Int)]
matriculas = [("Almeria","Matematicas",27),
              ("Sevilla","Informatica",325),
              ("Granada","Informatica",296),
              ("Huelva","Matematicas",41),
              ("Sevilla","Matematicas",122),
              ("Granada","Matematicas",131),
              ("Malaga","Informatica",314)]

totalAlumnos :: [(String,String,Int)] -> Int
totalAlumnos bd = sum [ n | (_,_,n) <- bd]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    totalMateria :: [(String,String,Int)] -> String -> Int
-- tal que (totalMateria bd m) es el número de alumnos de la base de
-- datos bd matriculados en la materia m. Por ejemplo,
--    totalMateria matriculas "Informatica" == 935
--    totalMateria matriculas "Matematicas" == 321
--    totalMateria matriculas "Fisica"      == 0
```

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

totalMateria :: [(String,String,Int)] -> String -> Int
totalMateria bd m = sum [ n | (_,m',n) <- bd, m == m']
```

## 4.2. Examen 2 (30 de Noviembre de 2011)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (30 de noviembre de 2011)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número es tipo repunit si todos sus dígitos son
-- 1. Por ejemplo, el número 1111 es repunit.
--
-- Definir la función
--    menorRepunit :: Integer -> Integer
-- tal que (menorRepunit n) es el menor repunit que es múltiplo de
-- n. Por ejemplo,
--    menorRepunit 3  == 111
--    menorRepunit 7  == 111111
-- ---------------------------------------------------------------------

menorRepunit :: Integer -> Integer
menorRepunit n = head [x | x <- [n,n*2..], repunit x]

-- (repunit n) se verifica si n es un repunit. Por ejemplo,
--    repunit 1111  ==  True
--    repunit 1121  ==  False
repunit :: Integer -> Bool
repunit n = and [x == 1 | x <- cifras n]

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 325  ==  [3,2,5]
cifras :: Integer -> [Integer]
cifras n = [read [d] | d <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    maximos :: [Float -> Float] -> [Float] -> [Float]
```

```
-- tal que (maximos fs xs) es la lista de los máximos de aplicar
-- cada función de fs a los elementos de xs. Por ejemplo,
--    maximos [(/2),(2/)] [5,10]                 == [5.0,0.4]
--    maximos [(^2),(/2),abs,(1/)] [1,-2,3,-4,5] == [25.0,2.5,5.0,1.0]
-- ---------------------------------------------------------------------

-- 1ª definición:
maximos :: [Float -> Float] -> [Float] -> [Float]
maximos fs xs = [maximum [f x | x <- xs] | f <- fs]

-- 2ª definición:
maximos2 :: [Float -> Float] -> [Float] -> [Float]
maximos2 fs xs = map maximum [ map f xs | f <- fs]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    reduceCifras :: Integer -> Integer
-- tal que (reduceCifras n) es el resultado de la reducción recursiva de
-- sus cifras; es decir, a partir del número n, se debe calcular la suma
-- de las cifras de n (llamémosle c), pero si c es a su vez mayor que 9,
-- se debe volver a calcular la suma de cifras de c y así sucesivamente
-- hasta que el valor obtenido sea menor o igual que 9. Por ejemplo,
--    reduceCifras   5   ==  5
--    reduceCifras 123   ==  6
--    reduceCifras 190   ==  1
--    reduceCifras 3456  ==  9
-- ---------------------------------------------------------------------

reduceCifras :: Integer -> Integer
reduceCifras n | m <= 9    = m
               | otherwise = reduceCifras m
               where m = sum (cifras n)

-- (sumaCifras n) es la suma de las cifras de n. Por ejemplo,
--    sumaCifras 3456  ==  18
sumaCifras :: Integer -> Integer
sumaCifras n = sum (cifras n)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las bases de datos con el nombre, profesión, año de
```

```haskell
-- nacimiento y año de defunción de una serie de personas se puede
-- representar como sigue
--    personas :: [(String,String,Int,Int)]
--    personas = [("Cervantes","Literatura",1547,1616),
--                ("Velazquez","Pintura",1599,1660),
--                ("Picasso","Pintura",1881,1973),
--                ("Beethoven","Musica",1770,1823),
--                ("Poincare","Ciencia",1854,1912),
--                ("Quevedo","Literatura",1580,1654),
--                ("Goya","Pintura",1746,1828),
--                ("Einstein","Ciencia",1879,1955),
--                ("Mozart","Musica",1756,1791),
--                ("Botticelli","Pintura",1445,1510),
--                ("Borromini","Arquitectura",1599,1667),
--                ("Bach","Musica",1685,1750)]
-- Es decir, se indica que por ejemplo Mozart se dedicó a la Música y
-- vivió entre 1756 y 1791.
--
-- Definir la función
--    coetaneos :: [(String,String,Int,Int)] -> String -> [String]
-- tal que (coetaneos bd p) es la lista de nombres de personas que
-- fueron coetáneos con la persona p; es decir que al menos alguno
-- de los años vividos por ambos coincidan. Se considera que una persona
-- no es coetanea a sí misma. Por ejemplo,
--    coetaneos personas "Einstein"   == ["Picasso", "Poincare"]
--    coetaneos personas "Botticelli" == []
-- ----------------------------------------------------------------------

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

```
              ("Bach","Musica",1685,1750)]

-- 1ª solución:
coetaneos :: [(String,String,Int,Int)] -> String -> [String]
coetaneos bd p =
    [n | (n,_,fn,fd) <- bd,
         n /= p,
         not ((fd < fnp) || (fdp < fn))]
    where (fnp,fdp) = head [(fn,fd) | (n,_,fn,fd) <- bd, n == p]

-- 2ª solución:

coetaneos2 :: [(String,String,Int,Int)] -> String -> [String]
coetaneos2 bd p =
    [n | (n,_,fn,fd) <- bd,
         n /= p,
         not (null (inter [fn..fd] [fnp..fdp]))]
    where (fnp,fdp) = head [(fn,fd) | (n,_,fn,fd) <- bd, n == p]

-- (inter xs ys) es la interseción de xs e ys. Por ejemplo,
--    inter [2,5,3,6] [3,7,2]  ==  [2,3]
inter :: Eq a => [a] -> [a] -> [a]
inter xs ys = [x | x <- xs, x `elem` ys]
```

## 4.3. Examen 3 (16 de Enero de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 3º examen de evaluación continua (16 de enero de 2012)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Dada una lista de números enteros, definiremos el
-- mayor salto como el mayor valor de las diferencias (en valor
-- absoluto) entre números consecutivos de la lista. Por ejemplo, dada
-- la lista [2,5,-3] las distancias son
--    3 (valor absoluto de la resta 2 - 5) y
--    8 (valor absoluto de la resta de 5 y (-3))
```

```haskell
-- Por tanto, su mayor salto es 8. No está definido el mayor salto para
-- listas con menos de 2 elementos
--
-- Definir, por compresión, la función
--    mayorSaltoC :: [Integer] -> Integer
-- tal que (mayorSaltoC xs) es el mayor salto de la lista xs. Por
-- ejemplo,
--    mayorSaltoC [1,5]              == 4
--    mayorSaltoC [10,-10,1,4,20,-2] == 22
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    mayorSaltoR :: [Integer] -> Integer
-- tal que (mayorSaltoR xs) es el mayor salto de la lista xs. Por
-- ejemplo,
--    mayorSaltoR [1,5]              == 4
--    mayorSaltoR [10,-10,1,4,20,-2] == 22
-- ---------------------------------------------------------------------

mayorSaltoR :: [Integer] -> Integer
mayorSaltoR [x,y]    = abs (x-y)
mayorSaltoR (x:y:ys) = max (abs (x-y)) (mayorSaltoR (y:ys))

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que mayorSaltoC y mayorSaltoR
-- son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mayorSalto :: [Integer] -> Property
prop_mayorSalto xs =
    length xs > 1 ==> mayorSaltoC xs == mayorSaltoR xs

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    acumulada :: [Int] -> [Int]
-- que (acumulada xs) es la lista que tiene en cada posición i el valor
-- que resulta de sumar los elementos de la lista xs desde la posicion 0
-- hasta la i. Por ejemplo,
--    acumulada [2,5,1,4,3] == [2,7,8,12,15]
--    acumulada [1,-1,1,-1] == [1,0,1,0]
-- ---------------------------------------------------------------------

-- 1ª definición (pro comprensión):
acumulada :: [Int] -> [Int]
acumulada xs = [sum (take n xs) | n <- [1..length xs]]

-- 2ª definición (por recursión)
acumuladaR :: [Int] -> [Int]
acumuladaR [] = []
acumuladaR xs = acumuladaR (init xs) ++ [sum xs]

-- 3ª definición (por recursión final):
acumuladaRF :: [Int] -> [Int]
acumuladaRF [] = []
acumuladaRF (x:xs) = reverse (aux xs [x])
    where aux [] ys = ys
          aux (x:xs) (y:ys) = aux xs (x+y:y:ys)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Dada una lista de números reales, la lista de
-- porcentajes contendrá el porcentaje de cada elemento de la lista
-- original en relación con la suma total de elementos. Por ejemplo,
-- la lista de porcentajes de [1,2,3,4] es [10.0,20.0,30.0,40.0],
-- ya que 1 es el 10% de la suma (1+2+3+4 = 10), y así sucesivamente.
--
-- Definir, por recursión, la función
--    porcentajesR :: [Float] -> [Float]
-- tal que (porcentajesR xs) es la lista de porcentaje de xs. Por
-- ejemplo,
--    porcentajesR [1,2,3,4] == [10.0,20.0,30.0,40.0]
--    porcentajesR [1,7,8,4]  ==  [5.0,35.0,40.0,20.0]
```

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

porcentajesR :: [Float] -> [Float]
porcentajesR xs = aux xs (sum xs)
    where aux [] _     = []
          aux (x:xs) s = (x*100/s) : aux xs s

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por comprensión, la función
--    porcentajesC :: [Float] -> [Float]
-- tal que (porcentajesC xs) es la lista de porcentaje de xs. Por
-- ejemplo,
--    porcentajesC [1,2,3,4] == [10.0,20.0,30.0,40.0]
--    porcentajesC [1,7,8,4]  ==  [5.0,35.0,40.0,20.0]
-- ---------------------------------------------------------------------

porcentajesC :: [Float] -> [Float]
porcentajesC xs = [x*100/s | x <- xs]
    where s = sum xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, usando map, la función
--    porcentajesS :: [Float] -> [Float]
-- tal que (porcentajesS xs) es la lista de porcentaje de xs. Por
-- ejemplo,
--    porcentajesS [1,2,3,4] == [10.0,20.0,30.0,40.0]
--    porcentajesS [1,7,8,4]  ==  [5.0,35.0,40.0,20.0]
-- ---------------------------------------------------------------------

porcentajesS :: [Float] -> [Float]
porcentajesS xs = map (*(100/sum xs)) xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, por plegado, la función
--    porcentajesP :: [Float] -> [Float]
-- tal que (porcentajesP xs) es la lista de porcentaje de xs. Por
-- ejemplo,
--    porcentajesP [1,2,3,4] == [10.0,20.0,30.0,40.0]
--    porcentajesP [1,7,8,4]  ==  [5.0,35.0,40.0,20.0]
-- ---------------------------------------------------------------------
```

```haskell
porcentajesF :: [Float] -> [Float]
porcentajesF xs = foldr (\x y -> (x*100/s):y) [] xs
    where s = sum xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir la función
--    equivalentes :: [Float] -> [Float] -> Bool
-- tal que (equivalentes xs ys) se verifica si el valor absoluto
-- de las diferencias de los elementos de xs e ys (tomados
-- posicionalmente) son infereriores a 0.001. Por ejemplo,
--    equivalentes [1,2,3] [1,2,3]     == True
--    equivalentes [1,2,3] [0.999,2,3] == True
--    equivalentes [1,2,3] [0.998,2,3] == False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
equivalentes :: [Float] -> [Float] -> Bool
equivalentes xs ys =
    and [abs (x-y) <= 0.001 | (x,y) <- zip xs ys]

-- 2ª definición (por recursión)
equivalentes2 :: [Float] -> [Float] -> Bool
equivalentes2 [] []         = True
equivalentes2 _ []          = False
equivalentes2 [] _          = False
equivalentes2 (x:xs) (y:ys) = abs (x-y) <= 0.001 && equivalentes2 xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Comprobar con QuickCheck que si xs es una lista de
-- números mayores o iguales que 0 cuya suma es mayor que 0, entonces
-- las listas (porcentajesR xs), (porcentajesC xs), (porcentajesS xs) y
-- (porcentajesF xs) son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_porcentajes :: [Float] -> Property
prop_porcentajes xs =
    and [x >= 0 | x <- xs] && sum xs > 0 ==>
    equivalentes (porcentajesC xs) ys &&
```

```
    equivalentes (porcentajesS xs) ys &&
    equivalentes (porcentajesF xs) ys
    where ys = porcentajesR xs

-- La comprobación es
--    ghci> quickCheck prop_porcentajes
--    *** Gave up! Passed only 15 tests.

-- Otra forma de expresar la propiedad es
prop_porcentajes2 :: [Float] -> Property
prop_porcentajes2 xs =
    sum xs' > 0 ==>
    equivalentes (porcentajesC xs') ys &&
    equivalentes (porcentajesS xs') ys &&
    equivalentes (porcentajesF xs') ys
    where xs' = map abs xs
          ys = porcentajesR xs'

-- Su comprobación es
--    ghci> quickCheck prop_porcentajes2
--    +++ OK, passed 100 tests.
```

## 4.4. Examen 4 ( 7 de Marzo de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 4º examen de evaluación continua (7 de marzo de 2012)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    inicialesDistintos :: Eq a => [a] -> Int
-- tal que (inicialesDistintos xs) es el número de elementos que hay en
-- xs antes de que aparezca el primer repetido. Por ejemplo,
--    inicialesDistintos [1,2,3,4,5,3] == 2
--    inicialesDistintos [1,2,3]       == 3
--    inicialesDistintos "ahora"       == 0
--    inicialesDistintos "ahorA"       == 5
```

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

inicialesDistintos [] = 0
inicialesDistintos (x:xs)
    | x `elem` xs = 0
    | otherwise = 1 + inicialesDistintos xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Diremos que un número entero positivo es autodivisible
-- si es divisible por todas sus cifras diferentes de 0.  Por ejemplo,
-- el número 150 es autodivisible ya que es divisible por 1 y por 5 (el
-- 0 no se usará en dicha comprobación), mientras que el 123 aunque es
-- divisible por 1 y por 3, no lo es por 2, y por tanto no es
-- autodivisible.
--
-- Definir, por comprensión, la función
--    autodivisibleC :: Integer -> Bool
-- tal que (autodivisibleC n) se verifica si n es autodivisible. Por
-- ejemplo,
--    autodivisibleC 0       == True
--    autodivisibleC 25      == False
--    autodivisibleC 1234    == False
--    autodivisibleC 1234608 == True
-- ---------------------------------------------------------------------

autodivisibleC :: Integer -> Bool
autodivisibleC n = and [d == 0 || n `rem` d == 0 | d <- cifras n]

-- (cifra n) es la lista de las cifras de n. Por ejemplo,
--    cifras 325  ==  [3,2,5]
cifras :: Integer -> [Integer]
cifras n = [read [y] | y <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    autodivisibleR :: Integer -> Bool
-- tal que (autodivisibleR n) se verifica si n es autodivisible. Por
-- ejemplo,
--    autodivisibleR 0       == True
--    autodivisibleR 25      == False
```

```haskell
--    autodivisibleR 1234    == False
--    autodivisibleR 1234608 == True
-- ---------------------------------------------------------------------

autodivisibleR :: Integer -> Bool
autodivisibleR n = aux n (cifras n)
    where aux _ [] = True
          aux n (x:xs) | x == 0 || n `rem` x == 0 = aux n xs
                       | otherwise                = False

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que las definiciones
-- autodivisibleC y autodivisibleR son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_autodivisible :: Integer -> Property
prop_autodivisible n =
    n > 0 ==> autodivisibleC n == autodivisibleR n

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Definir la función
--    siguienteAutodivisible :: Integer -> Integer
-- tal que (siguienteAutodivisible n) es el menor número autodivisible
-- mayor o igual que n. Por ejemplo,
--    siguienteAutodivisible 1234 == 1236
--    siguienteAutodivisible 111  == 111
-- ---------------------------------------------------------------------

siguienteAutodivisible :: Integer -> Integer
siguienteAutodivisible n =
    head [x | x <- [n..], autodivisibleR x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios se pueden representar mediante el
-- siguiente tipo de datos
```

```
--    data Arbol = H
--               | N Int Arbol Arbol
-- donde H representa una hoja y N un nodo con un valor y dos ramas. Por
-- ejemplo, el árbol
--             5
--             /\
--            /  \
--           /    \
--          1      4
--         / \    / \
--        /   \  H   5
--       5     H    / \
--      / \         H   H
--     H   H
-- se representa por
--    arbol1 :: Arbol
--    arbol1 = N 5 (N 1 (N 5 H H) H) (N 4 H (N 5 H H))
--
-- Definir la función
--    cuentaArbol :: Arbol -> Int -> Int
-- tal que (cuentaArbol a x) es el número de veces aparece x en el árbol
-- a. Por ejemplo,
--    cuentaArbol arbol1 5    = 3
--    cuentaArbol arbol1 2    = 0
--    cuentaArbol (N 5 H H) 5 = 1
--    cuentaArbol H 5         = 0
-- ---------------------------------------------------------------------

data Arbol = H
           | N Int Arbol Arbol
           deriving Show

arbol1 :: Arbol
arbol1 = N 5 (N 1 (N 5 H H) H) (N 4 H (N 5 H H))

cuentaArbol :: Arbol -> Int -> Int
cuentaArbol H _ = 0
cuentaArbol (N n a1 a2) x
    | n == x    = 1 + c1 + c2
    | otherwise = c1 + c2
```

```
    where c1 = cuentaArbol a1 x
          c2 = cuentaArbol a2 x
```

## 4.5. Examen 5 (28 de Marzo de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 5º examen de evaluación continua (28 de marzo de 2012)
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck
import PolRepTDA

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La moda estadística se define como el valor (o los
-- valores) con una mayor frecuencia en una lista de datos.
--
-- Definir la función
--    moda :: [Int] -> [Int]
-- tal que (moda ns) es la lista de elementos de xs con mayor frecuencia
-- absoluta de aparición en xs. Por ejemplo,
--    moda [1,2,3,2,3,3,3,1,1,1] == [1,3]
--    moda [1,2,2,3,2]           == [2]
--    moda [1,2,3]               == [1,2,3]
--    moda []                    == []
-- ---------------------------------------------------------------------

moda :: [Int] -> [Int]
moda xs = nub [x | x <- xs, ocurrencias x xs == m]
    where m = maximum [ocurrencias x xs | x <-xs]

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que los elementos de
-- (moda xs) pertenecen a xs.
```

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

-- La propiedad es
prop_moda_pertenece :: [Int] -> Bool
prop_moda_pertenece xs = and [x `elem` xs | x <- moda xs]

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que para cualquier elemento
-- de xs que no pertenezca a (moda xs), la cantidad de veces que aparece
-- x en xs es estrictamente menor que la cantidad de veces que aparece
-- el valor de la moda (para cualquier valor de la lista de elementos de
-- la moda).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_modas_resto_menores :: [Int] -> Bool
prop_modas_resto_menores xs =
    and [ocurrencias x xs < ocurrencias m xs |
         x <- xs,
         x `notElem` ys,
         m <- ys]
    where ys = moda xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Representaremos un recorrido como una secuencia de
-- puntos en el espacio de dos dimensiones. Para ello utilizaremos la
-- siguiente definición
--    data Recorrido = Nodo Double Double Recorrido
--                   | Fin
--                     deriving Show
-- De esta forma, el recorrido que parte del punto (0,0) pasa por el
-- punto (1,2) y termina en el (2,4) se representará como
```

```haskell
--    rec0 :: Recorrido
--    rec0 = Nodo 0 0 (Nodo 1 2 (Nodo 2 4 Fin))
--    A continuación se muestran otros ejemplos definidos
--    rec1, rec2, rec3, rec4 :: Recorrido
--    rec1 = Nodo 0 0 (Nodo 1 1 Fin)
--    rec2 = Fin
--    rec3 = Nodo 1 (-1) (Nodo 2 3 (Nodo 5 (-2) (Nodo 1 0 Fin)))
--    rec4 = Nodo 0 0
--            (Nodo 0 2
--              (Nodo 2 0
--                (Nodo 0 0
--                  (Nodo 2 2
--                    (Nodo 2 0
--                      (Nodo 0 0 Fin))))))
--
-- Definir la función
--    distanciaRecorrido :: Recorrido -> Double
-- tal que (distanciaRecorrido ps) esla suma de las distancias de todos
-- los segmentos de un recorrido ps. Por ejemplo,
--    distanciaRecorrido rec0     ==  4.4721359549995
--    distanciaRecorrido rec1     ==  1.4142135623730951
--    distanciaRecorrido rec2     ==  0.0
-- ----------------------------------------------------------------------

data Recorrido = Nodo Double Double Recorrido
               | Fin
                 deriving Show

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    nodosDuplicados :: Recorrido -> Int
-- tal que (nodosDuplicados e) es el número de nodos por los que el
-- recorrido r pasa dos o más veces. Por ejemplo,
--    nodosDuplicados rec3 == 0
--    nodosDuplicados rec4 == 2
-- ---------------------------------------------------------------------

nodosDuplicados :: Recorrido -> Int
nodosDuplicados Fin = 0
nodosDuplicados (Nodo x y r)
    | existeNodo r x y = 1 + nodosDuplicados (eliminaNodo r x y)
    | otherwise        = nodosDuplicados r

-- (existeNodo r x y) se verifica si el nodo (x,y) está en el recorrido
-- r. Por ejemplo,
--    existeNodo rec3 2 3  ==  True
--    existeNodo rec3 3 2  ==  False
existeNodo :: Recorrido -> Double -> Double -> Bool
existeNodo Fin _ _ = False
existeNodo (Nodo x y r) x' y'
           | x == x' && y == y' = True
           | otherwise          = existeNodo r x' y'

-- (eliminaNodo r x y) es el recorrido obtenido eliminando en r las
-- ocurrencias del nodo (x,y). Por ejemplo,
```

```haskell
--    ghci> rec3
--    Nodo 1.0 (-1.0) (Nodo 2.0 3.0 (Nodo 5.0 (-2.0) (Nodo 1.0 0.0 Fin)))
--    ghci> eliminaNodo rec3 2 3
--    Nodo 1.0 (-1.0) (Nodo 5.0 (-2.0) (Nodo 1.0 0.0 Fin))
eliminaNodo :: Recorrido -> Double -> Double -> Recorrido
eliminaNodo Fin _ _ = Fin
eliminaNodo (Nodo x y r) x' y'
            | x == x' && y == y' = eliminaNodo r x' y'
            | otherwise          = Nodo x y (eliminaNodo r x' y')

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se dice que un polinomio es completo si todos los
-- coeficientes desde el término nulo hasta el término de mayor grado
-- son distintos de cero.
--
-- Para hacer este ejercicio se utilizará algunas de las
-- implementaciones del tipo abstracto de datos de polinomio definidas
-- en el tema 21 y los siguientes ejemplos,
--    pol1, pol2, pol3 :: Polinomio Int
--    pol1 = polCero
--    pol2 = consPol 5 2 (consPol 3 1 (consPol 0 (-1) polCero))
--    pol3 = consPol 3 1 (consPol 2 2 (consPol 1 3 (consPol 0 4 polCero)))
--
-- Definir la función
--    polinomioCompleto :: Num a => Polinomio a -> Bool
-- tal que (polinomioCompleto p) se verifica si p es un polinomio
-- completo. Por ejemplo,
--    polinomioCompleto pol1  == False
--    polinomioCompleto pol2  == False
--    polinomioCompleto pol3  == True
-- ---------------------------------------------------

pol1, pol2, pol3 :: Polinomio Int
pol1 = polCero
pol2 = consPol 5 2 (consPol 3 1 (consPol 0 (-1) polCero))
pol3 = consPol 3 1 (consPol 2 2 (consPol 1 3 (consPol 0 4 polCero)))

polinomioCompleto :: Num a => Polinomio a -> Bool
polinomioCompleto p = 0 `notElem` coeficientes p
```

```haskell
-- (coeficientes p) es la lista de los coeficientes de p. Por ejemplo,
--    coeficientes pol1  ==  [0]
--    coeficientes pol2  ==  [2,0,1,0,0,-1]
--    coeficientes pol3  ==  [1,2,3,4]
coeficientes :: Num a => Polinomio a -> [a]
coeficientes p = [coeficiente n p | n <- [g,g-1..0]]
    where g = grado p

-- (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--    pol2                ==  2*x^5 + x^3 + -1
--    coeficiente 5 pol2  ==  2
--    coeficiente 6 pol2  ==  0
--    coeficiente 4 pol2  ==  0
--    coeficiente 3 pol2  ==  1
coeficiente :: Num a => Int -> Polinomio a -> a
coeficiente k p | k > n     = 0
                | k == n    = c
                | otherwise = coeficiente k r
                where n = grado p
                      c = coefLider p
                      r = restoPol p
```

## 4.6. Examen 6 ( 9 de Mayo de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 6º examen de evaluación continua (7 de mayo de 2012)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import Test.QuickCheck
import PolRepTDA
import TablaConMatrices

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    aplicaT :: (Ix a, Num b) => Array a b -> (b -> c) -> Array a c
-- tal que (aplicaT t f) es la tabla obtenida aplicado la función f a
-- los elementos de la tabla t. Por ejemplo,
```

```
--    ghci> aplicaT (array (1,5) [(1,6),(2,3),(3,-1),(4,9),(5,20)]) (+1)
--    array (1,5) [(1,7),(2,4),(3,0),(4,10),(5,21)]
--    ghci> :{
--    *Main| aplicaT (array ((1,1),(2,3)) [((1,1),3),((1,2),-1),((1,3),0),
--    *Main|                               ((2,1),0),((2,2),0),((2,3),-1)])
--    *Main|         (*2)
--    *Main| :}
--    array ((1,1),(2,3)) [((1,1),6),((1,2),-2),((1,3),0),
--                         ((2,1),0),((2,2),0),((2,3),-2)]
-- ---------------------------------------------------------------------

aplicaT :: (Ix a, Num b) => Array a b -> (b -> c) -> Array a c
aplicaT t f = listArray (bounds t) [f e | e <- elems t]

-- ---------------------------------------------------------------------
-- Ejercicio 2. En este ejercicio se usará el TAD de polinomios (visto
-- en el tema 21) y el de tabla (visto en el tema 18). Para ello, se
-- importan la librerías PolRepTDA y TablaConMatrices.
--
-- Definir la función
--    polTabla :: Num a => Polinomio a -> Tabla Integer a
-- tal que (polTabla p) es la tabla con los grados y coeficientes de los
-- términos del polinomio p; es decir, en la tabla el valor del índice n
-- se corresponderá con el coeficiente del grado n del mismo
-- polinomio. Por ejemplo,
--    ghci> polTabla (consPol 5 2 (consPol 3 (-1) polCero))
--    Tbl (array (0,5) [(0,0),(1,0),(2,0),(3,-1),(4,0),(5,2)])
-- -- ---------------------------------------------------------------------

polTabla :: Num a => Polinomio a -> Tabla Integer a
polTabla p = tabla (zip [0..] [coeficiente c p | c <- [0..grado p]])

-- (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--    let pol = consPol 5 2 (consPol 3 1 (consPol 0 (-1) polCero))
--    pol                 ==  2*x^5 + x^3 + -1
--    coeficiente 5 pol   ==  2
--    coeficiente 6 pol   ==  0
--    coeficiente 4 pol   ==  0
--    coeficiente 3 pol   ==  1
```

```
coeficiente :: Num a => Int -> Polinomio a -> a
coeficiente k p | k > n     = 0
                | k == n    = c
                | otherwise = coeficiente k r
               where n = grado p
                     c = coefLider p
                     r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 3. Diremos que una matriz es creciente si para toda
-- posición (i,j), el valor de dicha posición es menor o igual que los
-- valores en las posiciones adyacentes de índice superior, es decir,
-- (i+1,j), (i,j+1) e (i+1,j+1) siempre y cuando dichas posiciones
-- existan en la matriz.
--
-- Definir la función
--    matrizCreciente :: (Num a,Ord a) =>  Array (Int,Int) a -> Bool
-- tal que (matrizCreciente p) se verifica si la matriz p es
-- creciente. Por ejemplo,
--    matrizCreciente p1 == True
--    matrizCreciente p2 == False
-- donde las matrices p1 y p2 están definidas por
--    p1, p2 :: Array (Int,Int) Int
--    p1 = array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
--                              ((2,1),2),((2,2),3),((2,3),4),
--                              ((3,1),3),((3,2),4),((3,3),5)]
--    p2 = array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
--                              ((2,1),2),((2,2),1),((2,3),4),
--                              ((3,1),3),((3,2),4),((3,3),5)]
-- ---------------------------------------------------------------------

p1, p2 :: Array (Int,Int) Int
p1 = array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
                          ((2,1),2),((2,2),3),((2,3),4),
                          ((3,1),3),((3,2),4),((3,3),5)]
p2 = array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
                          ((2,1),2),((2,2),1),((2,3),4),
                          ((3,1),3),((3,2),4),((3,3),5)]

matrizCreciente :: (Num a,Ord a) =>  Array (Int,Int) a -> Bool
```

```haskell
matrizCreciente p =
    and ([p!(i,j) <= p!(i,j+1)   | i <- [1..m],   j <- [1..n-1]] ++
         [p!(i,j) <= p!(i+1,j)   | i <- [1..m-1], j <- [1..n]] ++
         [p!(i,j) <= p!(i+1,j+1) | i <- [1..m-1], j <- [1..n-1]])
    where (m,n) = snd (bounds p)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Partiremos de la siguiente definición para el tipo de
-- datos de árbol binario:
--    data Arbol = H
--               | N Int Arbol Arbol
--               deriving Show
--
-- Diremos que un árbol está balanceado si para cada nodo v la
-- diferencia entre el número de nodos (con valor) de sus ramas
-- izquierda y derecha es menor o igual que uno.
--
-- Definir la función
--    balanceado :: Arbol -> Bool
-- tal que (balanceado a) se verifica si el árbol a está
-- balanceado. Por ejemplo,
--    balanceado (N 5 (N 1 (N 5 H H) H) (N 4 H (N 5 H H))) == True
--    balanceado (N 1 (N 2 (N 3 H H) H) H)                 == False
-- ---------------------------------------------------------------------

data Arbol = H
           | N Int Arbol Arbol
           deriving Show

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

-- (numeroNodos a) es el número de nodos del árbol a. Por ejemplo,
--    numeroNodos (N 7 (N 1 (N 7 H H) H) (N 4 H (N 7 H H)))  ==  5
numeroNodos :: Arbol -> Int
numeroNodos H         = 0
numeroNodos (N _ i d) = 1 + numeroNodos i + numeroNodos d

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

```haskell
-- Ejercicio 5. Hemos hecho un estudio en varias agencias de viajes
-- analizando las ciudades para las que se han comprado billetes de
-- avión en la última semana. Las siguientes listas muestran ejemplos de
-- dichos listados, donde es necesario tener en cuenta que en la misma
-- lista se puede repetir la misma ciudad en más de una ocasión, en cuyo
-- caso el valor total será la suma acumulada. A continuación se
-- muestran algunas de dichas listas:
--    lista1, lista2, lista3, lista4 :: [(String,Int)]
--    lista1 = [("Paris",17),("Londres",12),("Roma",21),("Atenas",16)]
--    lista2 = [("Roma",5),("Paris",4)]
--    lista3 = [("Atenas",2),("Paris",11),("Atenas",1),("Paris",5)]
--    lista4 = [("Paris",5),("Roma",5),("Atenas",4),("Londres",6)]
--
-- Definir la función
--    ciudadesOrdenadas :: [[(String,Int)]] -> [String]
-- tal que (ciudadesOrdenadas ls) es la lista de los nombres de ciudades
-- ordenadas según el número de visitas (de mayor a menor). Por ejemplo,
--    ghci> ciudadesOrdenadas [lista1]
--    ["Roma","Paris","Atenas","Londres"]
--    ghci> ciudadesOrdenadas [lista1,lista2,lista3,lista4]
--    ["Paris","Roma","Atenas","Londres"]
-- ---------------------------------------------------------------------

lista1, lista2, lista3, lista4 :: [(String,Int)]
lista1 = [("Paris",17),("Londres",12),("Roma",21),("Atenas",16)]
lista2 = [("Roma",5),("Paris",4)]
lista3 = [("Atenas",2),("Paris",11),("Atenas",1),("Paris",5)]
lista4 = [("Paris",5),("Roma",5),("Atenas",4),("Londres",6)]

ciudadesOrdenadas :: [[(String,Int)]] -> [String]
ciudadesOrdenadas ls = [c | (c,v) <- ordenaLista (uneListas ls)]

-- (uneListas ls) es la lista obtenida uniendo las listas ls y
-- acumulando los resultados. Por ejemplo,
--    ghci> uneListas [lista1,lista2]
--    [("Paris",21),("Londres",12),("Roma",26),("Atenas",16)]
uneListas :: [[(String,Int)]] -> [(String,Int)]
uneListas ls = acumulaLista (concat ls)

-- (acumulaLista cvs) es la lista obtenida acumulando el número de
```

```haskell
-- visitas de la lista cvs. Por ejemplo,
--    acumulaLista lista3  ==  [("Atenas",3),("Paris",16)]
acumulaLista :: [(String,Int)] -> [(String,Int)]
acumulaLista cvs =
    [(c,sum [t | (c',t) <- cvs, c' == c]) | c <- nub (map fst cvs)]

-- (ordenaLista cvs9 es la lista de los elementos de cvs ordenados por
-- el número de visitas (de mayor a menor). Por ejemplo,
--    ghci> ordenaLista lista1
--    [("Roma",21),("Paris",17),("Atenas",16),("Londres",12)]
ordenaLista :: [(String,Int)] -> [(String,Int)]
ordenaLista cvs =
    reverse [(c,v) | (v,c) <- sort [(v',c') | (c',v') <- cvs]]
```

## 4.7. Examen 7 (11 de Junio de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 7º examen de evaluación continua (11 de junio de 2012)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import Test.QuickCheck
import PolRepTDA
import GrafoConVectorDeAdyacencia
import ConjuntoConListasOrdenadasSinDuplicados

-- ---------------------------------------------------------------------
-- Ejercicio 1. Diremos que una lista de números es una reducción
-- general de un número entero N si el sumatorio de L es igual a N y,
-- además, L es una lista de números enteros consecutivos y ascendente,
-- con más de un elemento. Por ejemplo, las listas [1,2,3,4,5], [4,5,6]
-- y [7,8] son reducciones generales de 15
--
-- Definir, por comprensión, la función
--    reduccionesBasicas :: Integer -> [[Integer]]
-- tal que (reduccionesBasicas n) es la lista de reducciones de n cuya
-- longitud (número de elementos) sea menor o igual que la raíz cuadrada
-- de n. Por ejemplo,
--    reduccionesBasicas 15   ==  [[4,5,6],[7,8]]
```

```
--    reduccionesBasicas 232  ==  []
-- ---------------------------------------------------------------------

-- 1ª definición:
reduccionesBasicasC :: Integer -> [[Integer]]
reduccionesBasicasC n =
    [[i..j] | i <- [1..n-1], j <- [i+1..i+r-1], sum [i..j] == n]
    where r = truncate (sqrt (fromIntegral n))

-- 2ª definición:
reduccionesBasicasC2 :: Integer -> [[Integer]]
reduccionesBasicasC2 n =
    [[i..j] | i <- [1..n-1], j <- [i+1..i+r-1], (i+j)*(1+j-i) `div` 2 == n]
    where r = truncate (sqrt (fromIntegral n))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Dada una matriz numérica A de dimensiones (m,n) y una
-- matriz booleana B de las mismas dimensiones, y dos funciones f y g,
-- la transformada de A respecto de B, f y g es la matriz C (de las
-- mismas dimensiones), tal que, para cada celda (i,j):
--    C(i,j) = f(A(i,j)) si B(i,j) es verdadero
--    C(i,j) = f(A(i,j)) si B(i,j) es falso
-- Por ejemplo, si A y B son las matrices
--    |1 2|   |True  False|
--    |3 4|   |False True |
-- respectivamente, y f y g son dos funciones tales que f(x) = x+1 y
-- g(x) = 2*x, entonces la transformada de A respecto de B, f y g es
--    |2 4|
--    |6 5|
--
-- En Haskell,
--    a :: Array (Int,Int) Int
--    a = array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),3),((2,2),4)]
--
--    b :: Array (Int,Int) Bool
--    b = array ((1,1),(2,2)) [((1,1),True),((1,2),False),((2,1),False),((2,2),Tr
--
-- Definir la función
--    transformada :: Array (Int,Int) a -> Array (Int,Int) Bool ->
--                    (a -> b) -> (a -> b) -> Array (Int,Int) b
```

```haskell
-- tal que (transformada a b f g) es la transformada de A respecto de B,
-- f y g. Por ejemplo,
--    ghci> transformada a b (+1) (*2)
--    array ((1,1),(2,2)) [((1,1),2),((1,2),4),((2,1),6),((2,2),5)]
-- ---------------------------------------------------------------------

a :: Array (Int,Int) Int
a = array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),3),((2,2),4)]

b :: Array (Int,Int) Bool
b = array ((1,1),(2,2)) [((1,1),True),((1,2),False),((2,1),False),((2,2),True)]

transformada :: Array (Int,Int) a -> Array (Int,Int) Bool ->
                (a -> b) -> (a -> b) -> Array (Int,Int) b
transformada a b f g =
    array ((1,1),(m,n)) [((i,j),aplica i j) | i <- [1..m], j <- [1..m]]
    where (m,n) = snd (bounds a)
          aplica i j | b!(i,j)   = f (a!(i,j))
                     | otherwise = g (a!(i,j))

-- ---------------------------------------------------------------------
-- Ejercicio 3. Dado un grafo dirigido G, diremos que un nodo está
-- aislado si o bien de dicho nodo no sale ninguna arista o bien no
-- llega al nodo ninguna arista. Por ejemplo, en el siguiente grafo
-- (Tema 22, pag. 31)
--    g = creaGrafo D (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
--                           (5,4,0),(6,2,0),(6,5,0)]
-- podemos ver que del nodo 1 salen 3 aristas pero no llega ninguna, por
-- lo que lo consideramos aislado. Así mismo, a los nodos 2 y 4 llegan
-- aristas pero no sale ninguna, por tanto también estarán aislados.
--
-- Definir la función
--    aislados :: (Ix v, Num p) => Grafo v p -> [v]
-- tal que (aislados g) es la lista de nodos aislados del grafo g. Por
-- ejemplo,
--    aislados g == [1,2,4]
-- ---------------------------------------------------------------------

g = creaGrafo D (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
                       (5,4,0),(6,2,0),(6,5,0)]
```

```haskell
aislados :: (Ix v, Num p) => Grafo v p -> [v]
aislados g = [n | n <- nodos g, adyacentes g n == [] || incidentes g n == [] ]

-- (incidentes g v) es la lista de los nodos incidentes con v en el
-- grafo g. Por ejemplo,
--    incidentes g 2  ==  [1,6]
--    incidentes g 1  ==  []
incidentes :: (Ix v,Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    gradosCoeficientes :: (Ord a, Num a) =>
--                          [Polinomio a] -> [(Int,Conj a)]
-- tal que (gradosCoeficientes ps) es una lista de pares, donde cada par
-- de la lista contendrá como primer elemento un número entero
-- (correspondiente a un grado) y el segundo elemento será un conjunto
-- que contendrá todos los coeficientes distintos de 0 que aparecen para
-- dicho grado en la lista de polinomios ps. Esta lista estará
-- ordenada de menor a mayor para todos los grados posibles de la lista de
-- polinomios. Por ejemplo, dados los siguientes polinomios
--    p1, p2, p3, p4 :: Polinomio Int
--    p1 = consPol 5 2 (consPol 3 (-1) polCero)
--    p2 = consPol 7 (-2) (consPol 5 1 (consPol 4 5 (consPol 2 1 polCero)))
--    p3 = polCero
--    p4 = consPol 4 (-1) (consPol 3 2 (consPol 1 1 polCero))
-- se tiene que
--    ghci> gradosCoeficientes [p1,p2,p3,p4]
--    [(1,{0,1}),(2,{0,1}),(3,{-1,0,2}),(4,{-1,0,5}),(5,{0,1,2}),
--     (6,{0}),(7,{-2,0})]
-- ---------------------------------------------------------------------

p1, p2, p3, p4 :: Polinomio Int
p1 = consPol 5 2 (consPol 3 (-1) polCero)
p2 = consPol 7 (-2) (consPol 5 1 (consPol 4 5 (consPol 2 1 polCero)))
p3 = polCero
p4 = consPol 4 (-1) (consPol 3 2 (consPol 1 1 polCero))

gradosCoeficientes :: (Ord a, Num a) => [Polinomio a] -> [(Int,Conj a)]
```

```haskell
gradosCoeficientes ps =
    [(k,foldr (inserta . coeficiente k) vacio ps) | k <- [1..m]]
    where m = maximum (map grado ps)

-- (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--    let pol = consPol 5 2 (consPol 3 1 (consPol 0 (-1) polCero))
--    pol                 ==  2*x^5 + x^3 + -1
--    coeficiente 5 pol   ==  2
--    coeficiente 6 pol   ==  0
--    coeficiente 4 pol   ==  0
--    coeficiente 3 pol   ==  1
coeficiente :: Num a => Int -> Polinomio a -> a
coeficiente k p | k > n     = 0
                | k == n    = c
                | otherwise = coeficiente k r
                where n = grado p
                      c = coefLider p
                      r = restoPol p
```

## 4.8. Examen 8 (29 de Junio de 2012)

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

## 4.9. Examen 9 ( 9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 32).

## 4.10. Examen 10 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 36).

# 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.