Exámenes de "Programación funcional con Haskell"

Vol. 4 (Curso 2012–13)

José A. Alonso Jiménez

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

Sevilla, 20 de noviembre de 2013

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 2012–13) 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 2012–13) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2012–13) [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 cuarto 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-12/temas/2012-13-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-12/ejercicios/ejercicios-I1M-2012.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol4
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

- Exámenes de “Programación funcional con Haskell”. Vol. 2 (Curso 2010–11) [6]
- Exámenes de “Programación funcional con Haskell”. Vol. 3 (Curso 2011–12) [7]

José A. Alonso
Sevilla, 20 de noviembre de 2013

[6] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol2
[7] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol3

1

Exámenes del grupo 1

José A. Alonso y Miguel A. Martínez

1.1. Examen 1 (8 de noviembre de 2012)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (8 de noviembre de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función primosEntre tal que (primosEntre x y)
-- es la lista de los número primos entre x e y (ambos inclusive). Por
-- ejemplo,
--    primosEntre 11 44  ==  [11,13,17,19,23,29,31,37,41,43]
-- ---------------------------------------------------------------------

primosEntre x y = [n | n <- [x..y], primo n]

-- (primo x) se verifica si x es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
primo n = factores n == [1, n]

-- (factores n) es la lista de los factores del número n. Por ejemplo,
--    factores 30  \valor  [1,2,3,5,6,10,15,30]
factores n = [x | x <- [1..n], n `mod` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función posiciones tal que (posiciones x ys)
```

```haskell
-- es la lista de las posiciones ocupadas por el elemento x en la lista
-- ys. Por ejemplo,
--    posiciones 5 [1,5,3,5,5,7]  ==  [1,3,4]
--    posiciones 'a' "Salamanca"  ==  [1,3,5,8]
-- ---------------------------------------------------------------------

posiciones x xs = [i | (x',i) <- zip xs [0..], x == x']

-- ---------------------------------------------------------------------
-- Ejercicio 3. El tiempo se puede representar por pares de la forma
-- (m,s) donde m representa los minutos y s los segundos. Definir la
-- función duracion tal que (duracion t1 t2) es la duración del
-- intervalo de tiempo que se inicia en t1 y finaliza en t2. Por
-- ejemplo,
--    duracion (2,15) (6,40)  ==  (4,25)
--    duracion (2,40) (6,15)  ==  (3,35)
-- ---------------------------------------------------------------------

tiempo (m1,s1) (m2,s2)
       | s1 <= s2  = (m2-m1,s2-s1)
       | otherwise = (m2-m1-1,60+s2-s1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función cortas tal que (cortas xs) es la
-- lista de las palabras más cortas (es decir, de menor longitud) de la
-- lista xs. Por ejemplo,
--    ghci> cortas ["hoy", "es", "un", "buen", "dia", "de", "sol"]
--    ["es","un","de"]
-- ---------------------------------------------------------------------

cortas xs = [x | x <- xs, length x == n]
    where n = minimum [length x | x <- xs]
```

1.2. Examen 2 (20 de diciembre de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (20 de diciembre de 2012)
-- ---------------------------------------------------------------------

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

```haskell
-- Ejercicio 1. Un entero positivo n es libre de cuadrado si no es
-- divisible por ningún m^2 > 1. Por ejemplo, 10 es libre de cuadrado
-- (porque 10 = 2*5) y 12 no lo es (ya que es divisible por 2^2).
-- Definir la función
--    libresDeCuadrado :: Int -> [Int]
-- tal que (libresDeCuadrado n) es la lista de los primeros n números
-- libres de cuadrado. Por ejemplo,
--    libresDeCuadrado 15  ==  [1,2,3,5,6,7,10,11,13,14,15,17,19,21,22]
-- ---------------------------------------------------------------------

libresDeCuadrado :: Int -> [Int]
libresDeCuadrado n =
    take n [n | n <- [1..], libreDeCuadrado n]

-- (libreDeCuadrado n) se verifica si n es libre de cuadrado. Por
-- ejemplo,
--    libreDeCuadrado 10  ==  True
--    libreDeCuadrado 12  ==  False
libreDeCuadrado :: Int -> Bool
libreDeCuadrado n =
    null [m | m <- [2..n], rem n (m^2) == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    duplicaPrimo :: [Int] -> [Int]
-- tal que (duplicaPrimo xs) es la lista obtenida sustituyendo cada
-- número primo de xs por su doble. Por ejemplo,
--    duplicaPrimo [2,5,9,7,1,3]  ==  [4,10,9,14,1,6]
-- ---------------------------------------------------------------------

duplicaPrimo :: [Int] -> [Int]
duplicaPrimo []     = []
duplicaPrimo (x:xs) | primo x   = (2*x) : duplicaPrimo xs
                    | otherwise = x : duplicaPrimo xs

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. 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 3020000  ==  4
-- ---------------------------------------------------------------------

ceros :: Int -> Int
ceros n | rem n 10 /= 0 = 0
        | otherwise     = 1 + ceros (div n 10)

-- ---------------------------------------------------------------------
-- Ejercicio 4. [Problema 387 del Proyecto Euler]. Un número de Harshad
-- es un entero divisible entre la suma de sus dígitos. Por ejemplo, 201
-- es un número de Harshad porque es divisible por 3 (la suma de sus
-- dígitos). Cuando se elimina el último dígito de 201 se obtiene 20 que
-- también es un número de Harshad. Cuando se elimina el último dígito
-- de 20 se obtiene 2 que también es un número de Harshad. Los números
-- como el 201 que son de Harshad y que los números obtenidos eliminando
-- sus últimos dígitos siguen siendo de Harshad se llaman números de
-- Harshad hereditarios por la derecha. Definir la función
--    numeroHHD :: Int -> Bool
-- tal que (numeroHHD n) se verifica si n es un número de Harshad
-- hereditario por la derecha. Por ejemplo,
--    numeroHHD 201  ==  True
--    numeroHHD 140  ==  False
--    numeroHHD 1104 ==  False
-- Calcular el mayor número de Harshad hereditario por la derecha con
-- tres dígitos.
-- ---------------------------------------------------------------------

-- (numeroH n) se verifica si n es un número de Harshad.
--    numeroH 201  ==  True
```

```haskell
numeroH :: Int -> Bool
numeroH n = rem n (sum (digitos n)) == 0

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 201  ==  [2,0,1]
digitos :: Int -> [Int]
digitos n = [read [d] | d <- show n]

numeroHHD :: Int -> Bool
numeroHHD n | n < 10    = True
            | otherwise = numeroH n && numeroHHD (div n 10)

-- El cálculo es
--    ghci> head [n | n <- [999,998..100], numeroHHD n]
--    902
```

1.3. Examen 3 (6 de febrero de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 3º examen de evaluación continua (6 de febrero de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    sumaR :: Num a => [[a]] -> a
-- tal que (sumaR xss) es la suma de todos los elementos de todas las
-- listas de xss. Por ejemplo,
--    sumaR [[1,3,5],[2,4,1],[3,7,9]]  ==  35
-- ---------------------------------------------------------------------

sumaR :: Num a => [[a]] -> a
sumaR []       = 0
sumaR (xs:xss) = sum xs + sumaR xss

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por plegado, la función
--    sumaP :: Num a => [[a]] -> a
-- tal que (sumaP xss) es la suma de todos los elementos de todas las
-- listas de xss. Por ejemplo,
--    sumaP [[1,3,5],[2,4,1],[3,7,9]]  ==  35
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    raicesEnteras :: Int -> Int -> Int -> [Int]
-- tal que (raicesEnteras a b c) es la lista de las raices enteras de la
-- ecuación ax^2+bx+c = 0. Por ejemplo,
--    raicesEnteras 1 (-6) 9     ==  [3]
--    raicesEnteras 1 (-6) 0     ==  [0,6]
--    raicesEnteras 5 (-6) 0     ==  [0]
--    raicesEnteras 1 1 (-6)     ==  [2,-3]
--    raicesEnteras 2 (-1) (-6)  ==  [2]
--    raicesEnteras 2 0 0        ==  [0]
--    raicesEnteras 6 5 (-6)     ==  []
-- Usando raicesEnteras calcular las raíces de la ecuación
-- 7x^2-11281x+2665212 = 0.
-- ---------------------------------------------------------------------

raicesEnteras :: Int -> Int -> Int -> [Int]
raicesEnteras a b c
  | b == 0 && c == 0       = [0]
  | c == 0 && rem b a /= 0 = [0]
  | c == 0 && rem b a == 0 = [0,-b `div` a]
  | otherwise              = [x | x <- divisores c, a*(x^2) + b*x + c == 0]

-- (divisores n) es la lista de los divisores enteros de n. Por ejemplo,
--    divisores (-6)  ==  [1,2,3,6,-1,-2,-3,-6]
divisores :: Int -> [Int]
divisores n = ys ++ (map (0-) ys)
  where ys = [x | x <-[1..abs n], mod n x == 0]

-- Una definición alternativa es
raicesEnteras2 a b c = [floor x | x <- raices a b c, esEntero x]

-- (esEntero x) se verifica si x es un número entero.
esEntero x = ceiling x == floor x
```

```haskell
-- (raices a b c) es la lista de las raices reales de la ecuación
-- ax^2+b*x+c = 0.
raices a b c | d < 0     = []
             | d == 0    = [y1]
             | otherwise = [y1,y2]
  where d = b^2 - 4*a*c
        y1 = ((-b) + sqrt d)/(2*a)
        y2 = ((-b) - sqrt d)/(2*a)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    segmentos :: (a -> Bool) -> [a] -> [[a]]
-- tal que (segmentos p xs) es la lista de los segmentos de xs cuyos
-- elementos no verifican la propiedad p. Por ejemplo,
--    segmentos odd [1,2,0,4,5,6,48,7,2]   ==  [[],[2,0,4],[6,48],[2]]
--    segmentos odd [8,6,1,2,0,4,5,6,7,2]  ==  [[8,6],[2,0,4],[6],[2]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Un número n es especial si al concatenar n y n+1 se
-- obtiene otro número que es divisible entre la suma de n y n+1. Por
-- ejemplo, 1, 4, 16 y 49 son especiales ya que
--        1+2  divide a   12     -       12/3  =  4
--        4+5  divide a   45     -       45/9  =  5
--       16+17 divide a 1617     -     1617/33 = 49
--       49+50 divide a 4950     -     4950/99 = 50
-- Definir la función
--    esEspecial :: Integer -> Bool
-- tal que (esEspecial n) se verifica si el número obtenido concatenando
-- n y n+1 es divisible entre la suma de n y n+1. Por ejemplo,
--    esEspecial 4  ==  True
--    esEspecial 7  ==  False
-- ---------------------------------------------------------------------

esEspecial :: Integer -> Bool
```

```haskell
esEspecial n = pegaNumeros n (n+1) `rem` (2*n+1) == 0

-- (pegaNumeros x y) es el número resultante de "pegar" los
-- números x e y. Por ejemplo,
--    pegaNumeros 12 987   ==  12987
--    pegaNumeros 1204 7   ==  12047
--    pegaNumeros 100 100  ==  100100
pegaNumeros :: Integer -> Integer -> Integer
pegaNumeros x y
    | y < 10    = 10*x+y
    | otherwise = 10 * pegaNumeros x (y `div` 10) + (y `mod` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    especiales :: Int -> [Integer]
-- tal que (especiales n) es la lista de los n primeros números
-- especiales. Por ejemplo,
--    especiales 5  ==  [1,4,16,49,166]
-- ---------------------------------------------------------------------

especiales :: Int -> [Integer]
especiales n = take n [x | x <- [1..], esEspecial x]
```

1.4. Examen 4 (21 de marzo de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (21 de marzo de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Los pares de números impares se pueden
-- ordenar según su suma y, entre los de la misma suma, su primer
-- elemento como sigue:
--    (1,1),(1,3),(3,1),(1,5),(3,3),(5,1),(1,7),(3,5),(5,3),(7,1),...
-- Definir la función
--     paresDeImpares :: [(Int,Int)]
-- tal que paresDeImpares es la lista de pares de números impares con
-- dicha ordenación. Por ejemplo,
--    ghci> take 10 paresDeImpares
--    [(1,1),(1,3),(3,1),(1,5),(3,3),(5,1),(1,7),(3,5),(5,3),(7,1)]
```

```
-- Basándose en paresDeImpares, definir la función
--    posicion
-- tal que (posicion p) es la posición del par p en la sucesión. Por
-- ejemplo,
--    posicion (3,5)  ==  7
-- ---------------------------------------------------------------------

paresDeImpares :: [(Int,Int)]
paresDeImpares =
  [(x,n-x) | n <- [2,4..], x <- [1,3..n]]

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Definir la constante
--    cuadradosConcatenados :: [(Integer,Integer,Integer)]
-- de forma que su valor es la lista de ternas (x,y,z) de tres cuadrados
-- perfectos tales que z es la concatenación de x e y. Por ejemplo,
--   ghci> take 5 cuadradosConcatenados
--   [(4,9,49),(16,81,1681),(36,100,36100),(1,225,1225),(4,225,4225)]
-- ---------------------------------------------------------------------

cuadradosConcatenados :: [(Integer,Integer,Integer)]
cuadradosConcatenados =
  [(x,y,concatenacion x y) | y <- cuadrados,
                             x <- [1..y],
                             esCuadrado x,
                             esCuadrado (concatenacion x y)]

-- cuadrados es la lista de los números que son cuadrados perfectos. Por
-- ejemplo,
--    take 5 cuadrados  ==  [1,4,9,16,25]
cuadrados :: [Integer]
cuadrados = [x^2 | x <- [1..]]

-- (concatenacion x y) es el número obtenido concatenando los números x
-- e y. Por ejemplo,
--    concatenacion 3252 476  ==  3252476
```

```haskell
concatenacion :: Integer -> Integer -> Integer
concatenacion x y = read (show x ++ show y)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] La expresiones aritméticas se pueden
-- representar mediante el siguiente tipo
--    data Expr = V Char
--              | N Int
--              | S Expr Expr
--              | P Expr Expr
-- por ejemplo, la expresión "z*(3+x)" se representa por
-- (P (V 'z') (S (N 3) (V 'x'))).
--
-- Definir la función
--    sumas :: Expr -> Int
-- tal que (sumas e) es el número de sumas en la expresión e. Por
-- ejemplo,
--    sumas (P (V 'z') (S (N 3) (V 'x')))  ==  1
--    sumas (S (V 'z') (S (N 3) (V 'x')))  ==  2
--    sumas (P (V 'z') (P (N 3) (V 'x')))  ==  0
-- ---------------------------------------------------------------------

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos]  Los árboles binarios se pueden representar
-- mediante el tipo Arbol definido por
--    data Arbol = H2 Int
--               | N2 Int Arbol Arbol
-- Por ejemplo, el árbol
--         1
--        / \
--       /   \
--      2     5
--     / \   / \
--    3   4 6   7
-- se puede representar por
--    N2 1 (N2 2 (H2 3) (H2 4)) (N2 5 (H2 6) (H2 7))
--
-- Definir la función
--    ramas :: Arbol -> [[Int]]
-- tal que (ramas a) es la lista de las ramas del árbol. Por ejemplo,
--    ghci> ramas (N2 1 (N2 2 (H2 3) (H2 4)) (N2 5 (H2 6) (H2 7)))
--    [[1,2,3],[1,2,4],[1,5,6],[1,5,7]]
-- ---------------------------------------------------------------------

data Arbol = H2 Int
           | N2 Int Arbol Arbol

ramas :: Arbol -> [[Int]]
ramas (H2 x)     = [[x]]
ramas (N2 x i d) = [x:r | r <- ramas i ++  ramas d]
```

1.5. Examen 5 (9 de mayo de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (16 de mayo de 2013)
-- ---------------------------------------------------------------------

import Data.Array

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

```haskell
--    empiezanPorUno :: [Int] -> [Int]
-- tal que (empiezanPorUno xs) es la lista de los elementos de xs que
-- empiezan por uno. Por ejemplo,
--    empiezanPorUno [31,12,7,143,214]  ==  [12,143]
-- ---------------------------------------------------------------------

-- 1ª definición: Por comprensión:
empiezanPorUno1 :: [Int] -> [Int]
empiezanPorUno1 xs =
  [x | x <- xs, head (show x) == '1']

-- 2ª definición: Por filtrado:
empiezanPorUno2 :: [Int] -> [Int]
empiezanPorUno2 xs =
  filter empiezaPorUno xs

empiezaPorUno :: Int -> Bool
empiezaPorUno x =
  head (show x) == '1'

-- 3ª definición: Por recursión:
empiezanPorUno3 :: [Int] -> [Int]
empiezanPorUno3 [] = []
empiezanPorUno3 (x:xs) | empiezaPorUno x = x : empiezanPorUno3 xs
                       | otherwise       = empiezanPorUno3 xs

-- 4ª definición: Por plegado:
empiezanPorUno4 :: [Int] -> [Int]
empiezanPorUno4 = foldr f []
  where f x ys | empiezaPorUno x = x : ys
               | otherwise       = ys

-- ---------------------------------------------------------------------
-- Ejercicio 2. Esta semana A. Helfgott ha publicado la primera
-- demostración de la conjetura débil de Goldbach que dice que todo
-- número impar mayor que 5 es suma de tres números primos (puede
-- repetirse alguno).
--
-- Definir la función
--    sumaDe3Primos :: Int -> [(Int,Int,Int)]
```

```haskell
-- tal que (sumaDe3sPrimos n) es la lista de las distintas
-- descomposiciones de n como suma de tres números primos. Por ejemplo,
--    sumaDe3Primos 7  ==  [(2,2,3)]
--    sumaDe3Primos 9  ==  [(2,2,5),(3,3,3)]
-- Calcular cuál es el menor número que se puede escribir de más de 500
-- formas como suma de tres números primos.
-- ---------------------------------------------------------------------

sumaDe3Primos :: Int -> [(Int,Int,Int)]
sumaDe3Primos n =
    [(x,y,n-x-y) | y <- primosN,
                   x <- takeWhile (<=y) primosN,
                   x+y <= n,
                   y <= n-x-y,
                   elem (n-x-y) primosN]
    where primosN = takeWhile (<=n) primos

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

-- primos es la lista de los números primos.
primos :: [Int]
primos = 2 : [n | n <- [3,5..], esPrimo n]

-- El cálculo es
--    ghci> head [n | n <- [1..], length (sumaDe3Primos n) > 500]
--    587

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los polinomios pueden representarse de forma densa. Por
-- ejemplo, el polinomio 6x^4-5x^2+4x-7 se puede representar por
-- [(4,6),(2,-5),(1,4),(0,-7)].
--
-- Definir la función
--    suma :: (Num a, Eq a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]
-- tal que (suma p q) es suma de los polinomios p y q representados de
-- forma densa. Por ejemplo,
--    ghci> suma [(5,3),(1,2),(0,1)] [(1,6),(0,4)]
--    [(5,3),(1,8),(0,5)]
```

```
--    ghci> suma [(1,6),(0,4)] [(5,3),(1,2),(0,1)]
--    [(5,3),(1,8),(0,5)]
--    ghci> suma [(5,3),(1,2),(0,1)] [(5,-3),(1,6),(0,4)]
--    [(1,8),(0,5)]
--    ghci> suma [(5,3),(1,2),(0,1)] [(5,4),(1,-2),(0,4)]
--    [(5,7),(0,5)]
-- ---------------------------------------------------------------------

suma :: (Num a, Eq a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]
suma [] q = q
suma p [] = p
suma ((n,b):p) ((m,c):q)
    | n > m      = (n,b) : suma p ((m,c):q)
    | n < m      = (m,c) : suma ((n,b):p) q
    | b + c == 0 = suma p q
    | otherwise  = (n,b+c) : suma p q

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se define el tipo de las matrices enteras por
--    type Matriz = Array (Integer,Integer) Integer
-- Definir la función
--    borraCols :: Integer -> Integer -> Matriz -> Matriz
-- tal que (borraCols j1 j2 p) es la matriz obtenida borrando las
-- columnas j1 y j2 (con j1 < j2) de la matriz p. Por ejemplo,
--    ghci> let p = listArray ((1,1),(2,4)) [1..8]
--    ghci> p
--    array ((1,1),(2,4)) [((1,1),1),((1,2),2),((1,3),3),((1,4),4),
--                         ((2,1),5),((2,2),6),((2,3),7),((2,4),8)]
--    ghci> borraCols 1 3 p
--    array ((1,1),(2,2)) [((1,1),2),((1,2),4),((2,1),6),((2,2),8)]
--    ghci> borraCols 2 3 p
--    array ((1,1),(2,2)) [((1,1),1),((1,2),4),((2,1),5),((2,2),8)]
-- ---------------------------------------------------------------------

type Matriz = Array (Integer,Integer) Integer

-- 1ª definición:
borraCols :: Integer -> Integer -> Matriz -> Matriz
borraCols j1 j2 p =
  borraCol (j2-1) (borraCol j1 p)
```

```haskell
-- (borraCol j1 p) es la matriz obtenida borrando la columna j1 de la
-- matriz p. Por ejemplo,
--    ghci> let p = listArray ((1,1),(2,4)) [1..8]
--    ghci> borraCol 2 p
--    array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),4),((2,1),5),((2,2),7),((2,
--    ghci> borraCol 3 p
--    array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),4),((2,1),5),((2,2),6),((2,
borraCol :: Integer -> Matriz -> Matriz
borraCol j1 p =
  array ((1,1),(m,n-1))
        [((i,j), f i j)| i <- [1..m], j <- [1..n-1]]
  where (_,(m,n)) = bounds p
        f i j | j < j1    = p!(i,j)
              | otherwise = p!(i,j+1)

-- 2ª definición:
borraCols2 :: Integer -> Integer -> Matriz -> Matriz
borraCols2 j1 j2 p =
  array ((1,1),(m,n-2))
        [((i,j), f i j)| i <- [1..m], j <- [1..n-2]]
  where (_,(m,n)) = bounds p
        f i j | j < j1    = p!(i,j)
              | j < j2-1  = p!(i,j+1)
              | otherwise = p!(i,j+2)

-- 3ª definición:
borraCols3 :: Integer -> Integer -> Matriz -> Matriz
borraCols3 j1 j2 p =
  listArray ((1,1),(n,m-2)) [p!(i,j) | i <- [1..n], j <- [1..m], j/=j1 && j/=j2]
  where (_,(n,m)) = bounds p
```

1.6. Examen 6 (13 de junio de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (13 de junio de 2013)
-- ---------------------------------------------------------------------

import Data.Array
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Un número es creciente si cada una de sus
-- cifras es mayor o igual que su anterior. Definir la función
--    numerosCrecientes :: [Integer] -> [Integer]
-- tal que (numerosCrecientes xs) es la lista de los números crecientes
-- de xs. Por ejemplo,
--    ghci> numerosCrecientes [21..50]
--    [22,23,24,25,26,27,28,29,33,34,35,36,37,38,39,44,45,46,47,48,49]
-- Usando la definición de numerosCrecientes calcular la cantidad de
-- números crecientes de 3 cifras.
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
numerosCrecientes :: [Integer] -> [Integer]
numerosCrecientes xs = [n | n <- xs, esCreciente (cifras n)]

-- (esCreciente xs) se verifica si xs es una sucesión cerciente. Por
-- ejemplo,
--    esCreciente [3,5,5,12]  ==  True
--    esCreciente [3,5,4,12]  ==  False
esCreciente :: Ord a => [a] -> Bool
esCreciente (x:y:zs) = x <= y && esCreciente (y:zs)
esCreciente _        = True

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

-- El cálculo es
--    ghci> length (numerosCrecientes [100..999])
--    165

-- 2ª definición (por filtrado):
numerosCrecientes2 :: [Integer] -> [Integer]
numerosCrecientes2 = filter (\n -> esCreciente (cifras n))

-- 3ª definición (por recursión):
numerosCrecientes3 :: [Integer] -> [Integer]
numerosCrecientes3 [] = []
```

```haskell
numerosCrecientes3 (n:ns)
  | esCreciente (cifras n) = n : numerosCrecientes3 ns
  | otherwise              = numerosCrecientes3 ns

-- 4ª definición (por plegado):
numerosCrecientes4 :: [Integer] -> [Integer]
numerosCrecientes4 = foldr f []
  where f n ns | esCreciente (cifras n) = n : ns
               | otherwise              = ns

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    sublistasIguales :: Eq a => [a] -> [[a]]
-- tal que (sublistasIguales xs) es la listas de elementos consecutivos
-- de xs que son iguales. Por ejemplo,
--    ghci> sublistasIguales [1,5,5,10,7,7,7,2,3,7]
--    [[1],[5,5],[10],[7,7,7],[2],[3],[7]]
-- ---------------------------------------------------------------------

-- 1ª definición:
sublistasIguales :: Eq a => [a] -> [[a]]
sublistasIguales [] = []
sublistasIguales (x:xs) =
  (x : takeWhile (==x) xs) : sublistasIguales (dropWhile (==x) xs)

-- 2ª definición:
sublistasIguales2 :: Eq a => [a] -> [[a]]
sublistasIguales2 []     = []
sublistasIguales2 [x]    = [[x]]
sublistasIguales2 (x:y:zs)
  | x == u    = (x:u:us):vss
  | otherwise = [x]:((u:us):vss)
  where ((u:us):vss) = sublistasIguales2 (y:zs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Los árboles binarios se pueden representar
-- con el de dato algebraico
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
```

```haskell
-- Por ejemplo, los árboles
--         9                9
--        / \              / \
--       /   \            /   \
--      8     6          8     6
--     / \   / \        / \   / \
--    3   2 4   5      3   2 4   7
-- se pueden representar por
--    ej1, ej2:: Arbol Int
--    ej1 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--    ej2 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 7 H H))
-- Un árbol binario ordenado es un árbol binario (ABO) en el que los
-- valores de cada nodo es mayor o igual que los valores de sus
-- hijos. Por ejemplo, ej1 es un ABO, pero ej2 no lo es.
--
-- Definir la función esABO
--    esABO :: Ord t => Arbol t -> Bool
-- tal que (esABO a) se verifica si a es un árbol binario ordenado. Por
-- ejemplo.
--    esABO ej1 == True
--    esABO ej2 == False
-- ----------------------------------------------------------------------

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

ej1, ej2 :: Arbol Int
ej1 = N 9 (N 8 (N 3 H H) (N 2 H H))
          (N 6 (N 4 H H) (N 5 H H))

ej2 = N 9 (N 8 (N 3 H H) (N 2 H H))
          (N 6 (N 4 H H) (N 7 H H))

-- 1ª definición
esABO :: Ord a => Arbol a -> Bool
esABO H                       = True
esABO (N x H H)               = True
esABO (N x m1@(N x1 a1 b1) H) = x >= x1 && esABO m1
esABO (N x H m2@(N x2 a2 b2)) = x >= x2 && esABO m2
```

```haskell
esABO (N x m1@(N x1 a1 b1) m2@(N x2 a2 b2)) =
      x >= x1 && esABO m1 && x >= x2 && esABO m2

-- 2ª definición
esABO2 :: Ord a => Arbol a -> Bool
esABO2 H         = True
esABO2 (N x i d) = mayor x i && mayor x d && esABO2 i && esABO2 d
       where  mayor x H         = True
              mayor x (N y _ _) = x >= y

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Definir la función
--    paresEspecialesDePrimos :: Integer -> [(Integer,Integer)]
-- tal que (paresEspecialesDePrimos n) es la lista de los pares de
-- primos (p,q) tales que p < q y q-p es divisible por n. Por ejemplo,
--    ghci> take 9 (paresEspecialesDePrimos 2)
--    [(3,5),(3,7),(5,7),(3,11),(5,11),(7,11),(3,13),(5,13),(7,13)]
--    ghci> take 9 (paresEspecialesDePrimos 3)
--    [(2,5),(2,11),(5,11),(7,13),(2,17),(5,17),(11,17),(7,19),(13,19)]
-- ---------------------------------------------------------------------

paresEspecialesDePrimos :: Integer -> [(Integer,Integer)]
paresEspecialesDePrimos n =
  [(p,q) | (p,q) <- paresPrimos, rem (q-p) n == 0]

-- paresPrimos es la lista de los pares de primos (p,q) tales que p < q.
-- Por ejemplo,
--    ghci> take 9 paresPrimos
--    [(2,3),(2,5),(3,5),(2,7),(3,7),(5,7),(2,11),(3,11),(5,11)]
paresPrimos :: [(Integer,Integer)]
paresPrimos = [(p,q) | q <- primos, p <- takeWhile (<q) primos]

-- primos es la lista de primos. Por ejemplo,
--    take 9 primos  ==  [2,3,5,7,11,13,17,19,23]
primos :: [Integer]
primos = criba [2..]

criba :: [Integer] -> [Integer]
criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5. Las matrices enteras se pueden representar mediante
-- tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
--
-- Definir la función
--    ampliaColumnas :: Matriz -> Matriz -> Matriz
-- tal que (ampliaColumnas p q) es la matriz construida añadiendo las
-- columnas de la matriz q a continuación de las de p (se supone que
-- tienen el mismo número de filas). Por ejemplo, si p y q representa
-- las dos primeras matrices, entonces (ampliaColumnas p q) es la
-- tercera
--    |0 1|    |4 5 6|    |0 1 4 5 6|
--    |2 3|    |7 8 9|    |2 3 7 8 9|
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

ampliaColumnas :: Matriz -> Matriz -> Matriz
ampliaColumnas p1 p2 =
  array ((1,1),(m,n1+n2)) [((i,j), f i j) | i <- [1..m], j <- [1..n1+n2]]
    where ((_,_),(m,n1)) = bounds p1
          ((_,_),(_,n2)) = bounds p2
          f i j | j <= n1   = p1!(i,j)
                | otherwise = p2!(i,j-n1)

-- Ejemplo
--    ghci> let p = listArray ((1,1),(2,2)) [0..3] :: Matriz
--    ghci> let q = listArray ((1,1),(2,3)) [4..9] :: Matriz
--    ghci> ampliaColumnas p q
--    array ((1,1),(2,5))
--          [((1,1),0),((1,2),1),((1,3),4),((1,4),5),((1,5),6),
--           ((2,1),2),((2,2),3),((2,3),7),((2,4),8),((2,5),9)]
```

1.7. Examen 7 (3 de julio de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 7º examen de evaluación continua (3 de julio de 2013)
-- ---------------------------------------------------------------------
```

```haskell
import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Dos listas son cíclicamente iguales si tienen
-- el mismo número de elementos en el mismo orden. Por ejemplo, son
-- cíclicamente iguales los siguientes pares de listas
--    [1,2,3,4,5] y [3,4,5,1,2],
--    [1,1,1,2,2] y [2,1,1,1,2],
--    [1,1,1,1,1] y [1,1,1,1,1]
-- pero no lo son
--    [1,2,3,4] y [1,2,3,5],
--    [1,1,1,1] y [1,1,1],
--    [1,2,2,1] y [2,2,1,2]
-- Definir la función
--    iguales :: Eq a => [a] -> [a] -> Bool
-- tal que (iguales xs ys) se verifica si xs es ys son cíclicamente
-- iguales. Por ejemplo,
--    iguales [1,2,3,4,5] [3,4,5,1,2]  ==  True
--    iguales [1,1,1,2,2] [2,1,1,1,2]  ==  True
--    iguales [1,1,1,1,1] [1,1,1,1,1]  ==  True
--    iguales [1,2,3,4] [1,2,3,5]      ==  False
--    iguales [1,1,1,1] [1,1,1]        ==  False
--    iguales [1,2,2,1] [2,2,1,2]      ==  False
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

iguales1 :: Ord a => [a] -> [a] -> Bool
iguales1 xs ys =
    permutacionApares xs == permutacionApares ys

-- (permutacionApares xs) es la lista ordenada de los pares de elementos
-- consecutivos de elementos de xs. Por ejemplo,
--    permutacionApares [2,1,3,5,4]  ==  [(1,3),(2,1),(3,5),(4,2),(5,4)]
permutacionApares :: Ord a => [a] -> [(a, a)]
permutacionApares xs =
    sort (zip xs (tail xs) ++ [(last xs, head xs)])
```

```haskell
-- 2ª solucion
-- ===========

-- (iguales2 xs ys) se verifica si las listas xs e ys son cíclicamente
-- iguales. Por ejemplo,
iguales2 :: Eq a => [a] -> [a] -> Bool
iguales2 xs ys =
    elem ys (ciclos xs)

-- (ciclo xs) es la lista obtenida pasando el último elemento de xs al
-- principio. Por ejemplo,
--    ciclo [2,1,3,5,4]  ==  [4,2,1,3,5]
ciclo :: [a] -> [a]
ciclo xs = (last xs): (init xs)

-- (kciclo k xs) es la lista obtenida pasando los k últimos elementos de
-- xs al principio. Por ejemplo,
--    kciclo 2 [2,1,3,5,4]  ==  [5,4,2,1,3]
kciclo :: (Eq a, Num a) => a -> [a1] -> [a1]
kciclo 1 xs = ciclo xs
kciclo k xs = kciclo (k-1) (ciclo xs)

-- (ciclos xs) es la lista de las listas cíclicamente iguales a xs. Por
-- ejemplo,
--    ghci> ciclos [2,1,3,5,4]
--    [[4,2,1,3,5],[5,4,2,1,3],[3,5,4,2,1],[1,3,5,4,2],[2,1,3,5,4]]
ciclos :: [a] -> [[a]]
ciclos xs = [kciclo k xs | k <- [1..length xs]]

-- 3º solución
-- ===========

iguales3 :: Eq a => [a] -> [a] -> Bool
iguales3 xs ys =
    length xs == length ys && isInfixOf xs (ys ++ ys)

-- ---------------------------------------------------------------------
-- Ejercicio ?. Un número natural n es casero respecto de f si las
-- cifras de f(n) es una sublista de las de n. Por ejemplo,
-- * 1234 es casero repecto de resto de dividir por 173, ya que el resto
```

```
--   de dividir 1234 entre 173 es 23 que es una sublista de 1234;
-- * 1148 es casero respecto de la suma de cifras, ya que la suma de las
--   cifras de 1148 es 14 que es una sublista de 1148.
-- Definir la función
--    esCasero :: (Integer -> Integer) -> Integer -> Bool
-- tal que (esCasero f x) se verifica si x es casero respecto de f. Por
-- ejemplo,
--    esCasero (\x -> rem x 173) 1234  ==  True
--    esCasero (\x -> rem x 173) 1148  ==  False
--    esCasero sumaCifras 1148         ==  True
--    esCasero sumaCifras 1234         ==  False
-- donde (sumaCifras n) es la suma de las cifras de n.
--
-- ¿Cuál es el menor número casero respecto de la suma de cifras mayor
-- que 2013?
-- ---------------------------------------------------------------------

esCasero :: (Integer -> Integer) -> Integer -> Bool
esCasero f x =
    esSublista (cifras (f x)) (cifras x)

-- (esSublista xs ys) se verifica si xs es una sublista de ys; es decir,
-- si existen dos listas as y bs tales que
--    ys = as ++ xs ++ bs
esSublista :: Eq a => [a] -> [a] -> Bool
esSublista = isInfixOf

-- Se puede definir por
esSublista2 :: Eq a => [a] -> [a] -> Bool
esSublista2 xs ys =
    or [esPrefijo xs zs | zs <- sufijos ys]

-- (esPrefijo xs ys) se verifica si xs es un prefijo de ys. Por
-- ejemplo,
--    esPrefijo "ab" "abc"  ==  True
--    esPrefijo "ac" "abc"  ==  False
--    esPrefijo "bc" "abc"  ==  False
esPrefijo :: Eq a => [a] -> [a] -> Bool
esPrefijo []  _          = True
esPrefijo _   []         = False
```

```haskell
esPrefijo (x:xs) (y:ys) = x == y && isPrefixOf xs ys

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

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

-- (sumaCifras x) es la suma de las cifras de x. Por ejemplo,
--    sumaCifras 325  ==  10
sumaCifras :: Integer -> Integer
sumaCifras = sum . cifras

-- El cálculo del menor número casero respecto de la suma mayor que 2013
-- es
--    ghci> head [n | n <- [2014..], esCasero sumaCifras n]
--    2099

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    interseccion :: Ord a => [a] -> [a] -> [a]
-- tal que (interseccion xs ys) es la intersección de las dos listas,
-- posiblemente infinitas, ordenadas de menor a mayor xs e ys. Por ejemplo,
--    take 5 (interseccion [2,4..] [3,6..])  ==  [6,12,18,24,30]
-- ---------------------------------------------------------------------

interseccion :: Ord a => [a] -> [a] -> [a]
interseccion [] _ = []
interseccion _ [] = []
interseccion (x:xs) (y:ys)
    | x == y    = x : interseccion xs ys
    | x < y     = interseccion (dropWhile (<y) xs) (y:ys)
    | otherwise = interseccion (x:xs) (dropWhile (<x) ys)

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Los árboles binarios se pueden representar
```

```haskell
-- mediante el tipo Arbol definido por
--    data Arbol = H Int
--               | N Int Arbol Arbol
-- Por ejemplo, el árbol
--         1
--        / \
--       /   \
--      2     5
--     / \   / \
--    3   4 6   7
-- se puede representar por
--    N 1 (N 2 (H 3) (H 4)) (N 5 (H 6) (H 7))
-- Definir la función
--    esSubarbol :: Arbol -> Arbol -> Bool
-- tal que (esSubarbol a1 a2) se verifica si a1 es un subárbol de
-- a2. Por ejemplo,
--    esSubarbol (H 2) (N 2 (H 2) (H 4))              ==  True
--    esSubarbol (H 5) (N 2 (H 2) (H 4))              ==  False
--    esSubarbol (N 2 (H 2) (H 4)) (N 2 (H 2) (H 4))  ==  True
--    esSubarbol (N 2 (H 4) (H 2)) (N 2 (H 2) (H 4))  ==  False
-- ---------------------------------------------------------------------

data Arbol= H Int
          | N Int Arbol Arbol

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

-- (igualArbol a1 a2) se verifica si los árboles a1 y a2 son iguales.
igualArbol :: Arbol -> Arbol -> Bool
igualArbol (H x) (H y) = x == y
igualArbol (N r1 i1 d1) (N r2 i2 d2) =
    r1 == r2 && igualArbol i1 i2 && igualArbol d1 d2
igualArbol _ _ = False
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] Las matrices enteras se pueden representar
-- mediante tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
-- Por ejemplo, las matrices
--    | 1  2  3  4  5 |    | 1 2 3 |
--    | 2  6  8  9  4 |    | 2 6 8 |
--    | 3  8  0  8  3 |    | 3 8 0 |
--    | 4  9  8  6  2 |
--    | 5  4  3  2  1 |
-- se puede definir por
--    ejM1, ejM2 :: Matriz
--    ejM1 = listArray ((1,1),(5,5)) [1,2,3,4,5,
--                                    2,6,8,9,4,
--                                    3,8,0,8,3,
--                                    4,9,8,6,2,
--                                    5,4,3,2,1]
--
--    ejM2 = listArray ((1,1),(3,3)) [1,2,3,
--                                    2,6,8,
--                                    3,8,0]
-- Una matriz cuadrada es bisimétrica si es simétrica respecto de su
-- diagonal principal y de su diagonal secundaria. Definir la función
--    esBisimetrica :: Matriz -> Bool
-- tal que (esBisimetrica p) se verifica si p es bisimétrica. Por
-- ejemplo,
--    esBisimetrica ejM1  ==  True
--    esBisimetrica ejM2  ==  False
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

ejM1, ejM2 :: Matriz
ejM1 = listArray ((1,1),(5,5)) [1,2,3,4,5,
                                2,6,8,9,4,
                                3,8,0,8,3,
                                4,9,8,6,2,
                                5,4,3,2,1]
```

```haskell
ejM2 = listArray ((1,1),(3,3)) [1,2,3,
                                2,6,8,
                                3,8,0]

-- 1ª definición:
esBisimetrica :: Matriz -> Bool
esBisimetrica p =
    and [p!(i,j) == p!(j,i) | i <- [1..n], j <- [1..n]] &&
    and [p!(i,j) == p!(n+1-j,n+1-i) | i <- [1..n], j <- [1..n]]
    where ((_,_),(n,_)) = bounds p

-- 2ª definición:
esBisimetrica2 :: Matriz -> Bool
esBisimetrica2 p = p == simetrica p && p == simetricaS p

-- (simetrica p) es la simétrica de la matriz p respecto de la diagonal
-- principal. Por ejemplo,
--    ghci> simetrica (listArray ((1,1),(4,4)) [1..16])
--    array ((1,1),(4,4)) [((1,1),1),((1,2),5),((1,3), 9),((1,4),13),
--                         ((2,1),2),((2,2),6),((2,3),10),((2,4),14),
--                         ((3,1),3),((3,2),7),((3,3),11),((3,4),15),
--                         ((4,1),4),((4,2),8),((4,3),12),((4,4),16)]
simetrica :: Matriz -> Matriz
simetrica p =
    array ((1,1),(n,n)) [((i,j),p!(j,i)) | i <- [1..n], j <- [1..n]]
    where ((_,_),(n,_)) = bounds p

-- (simetricaS p) es la simétrica de la matriz p respecto de la diagonal
-- secundaria. Por ejemplo,
--    ghci> simetricaS (listArray ((1,1),(4,4)) [1..16])
--    array ((1,1),(4,4)) [((1,1),16),((1,2),12),((1,3),8),((1,4),4),
--                         ((2,1),15),((2,2),11),((2,3),7),((2,4),3),
--                         ((3,1),14),((3,2),10),((3,3),6),((3,4),2),
--                         ((4,1),13),((4,2), 9),((4,3),5),((4,4),1)]
simetricaS :: Matriz -> Matriz
simetricaS p =
    array ((1,1),(n,n)) [((i,j),p!(n+1-j,n+1-i)) | i <- [1..n], j <- [1..n]]
    where ((_,_),(n,_)) = bounds p
```

1.8. Examen 8 (13 de septiembre de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 2ª convocatoria (13 de septiembre de 2013)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [1 punto] Las notas se pueden agrupar de distinta
-- formas. Una es por la puntuación; por ejemplo,
--    [(4,[”juan”,”ana”]),(9,[”rosa”,”luis”,”mar”])]
-- Otra es por nombre; por ejemplo,
--    [(”ana”,4),(”juan”,4),(”luis”,9),(”mar”,9),(”rosa”,9)]
--
-- Definir la función
--    transformaPaN :: [(Int,[String])] -> [(String,Int)]
-- tal que (transformaPaN xs) es la agrupación de notas por nombre
-- correspondiente a la agrupación de notas por puntuación xs. Por
-- ejemplo,
--    > transformaPaN [(4,[”juan”,”ana”]),(9,[”rosa”,”luis”,”mar”])]
--    [(”ana”,4),(”juan”,4),(”luis”,9),(”mar”,9),(”rosa”,9)]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
transformaPaN :: [(Int,[String])] -> [(String,Int)]
transformaPaN xs = sort [(a,n) | (n,as) <- xs, a <- as]

-- 2ª definición (por recursión):
transformaPaN2 :: [(Int,[String])] -> [(String,Int)]
transformaPaN2 []          = []
transformaPaN2 ((n,xs):ys) = [(x,n)|x<-xs] ++ transformaPaN2 ys

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. [1 punto] Definir la función
--    transformaNaP :: [(String,Int)] -> [(Int,[String])]
-- tal que (transformaPaN xs) es la agrupación de notas por nombre
-- correspondiente a la agrupación de notas por puntuación xs. Por
-- ejemplo,
```

```
--    > transformaNaP [("ana",4),("juan",4),("luis",9),("mar",9),("rosa",9)]
--    [(4,["ana","juan"]),(9,["luis","mar","rosa"])]
-- ---------------------------------------------------------------------

transformaNaP :: [(String,Int)] -> [(Int,[String])]
transformaNaP xs = [(n, [a | (a,n') <- xs, n' == n]) | n <- notas]
    where notas = sort (nub [n | (_,n) <- xs])

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    multiplosCon9 :: Integer -> [Integer]
-- tal que (multiplosCon9 n) es la lista de los múltiplos de n cuya
-- única cifra es 9. Por ejemplo,
--    take 3 (multiplosCon9 3)  ==  [9,99,999]
--    take 3 (multiplosCon9 7)  ==  [999999,999999999999,999999999999999999]
-- Calcular el menor múltiplo de 2013 formado sólo por nueves.
-- ---------------------------------------------------------------------

multiplosCon9 :: Integer -> [Integer]
multiplosCon9 n = [x | x <- numerosCon9, rem x n == 0]

-- numerosCon9 es la lista de los número cuyas cifras son todas iguales
-- a 9. Por ejemplo,
--    take 5 numerosCon9  ==  [9,99,999,9999,99999]
numerosCon9 :: [Integer]
numerosCon9 = [10^n-1 | n <- [1..]]

-- 2ª definición (por recursión):
numerosCon9R :: [Integer]
numerosCon9R = 9 : sig 9
    where sig x = (10*x+9) : sig (10*x+9)

-- El cálculo es
--    ghci> head (multiplosCon9 2013)
--    99999999999999999999999999999999999999999999999999999999999

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Una sucesión es suave si valor absoluto de la
-- diferencia de sus términos consecutivos es 1. Definir la función
--    suaves :: Int -> [[Int]]
```

```haskell
-- tal que (suaves n) es la lista de las sucesiones suaves de longitud n
-- cuyo último término es 0. Por ejemplo,
--    suaves 2  ==  [[1,0],[-1,0]]
--    suaves 3  ==  [[2,1,0],[0,1,0],[0,-1,0],[-2,-1,0]]
-- ---------------------------------------------------------------------

suaves :: Int -> [[Int]]
suaves 0 = []
suaves 1 = [[0]]
suaves n = concat [[x+1:x:xs,x-1:x:xs] | (x:xs) <- suaves (n-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Los árboles binarios se pueden representar
-- mediante el tipo Arbol definido por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo, el árbol
--          1
--         / \
--        /   \
--       4     6
--      / \   / \
--     0   7 4   3
-- se puede definir por
--    ej1 :: Arbol Int
--    ej1 = N 1 (N 4 (H 0) (H 7)) (N 6 (H 4) (H 3))
--
-- Definir la función
--    algunoArbol :: Arbol t -> (t -> Bool) -> Bool
-- tal que (algunoArbol a p) se verifica si algún elemento del árbol a
-- cumple la propiedad p. Por ejemplo,
--    algunoArbol ej1 (>9)  ==  False
--    algunoArbol ej1 (>5)  ==  True
-- ---------------------------------------------------------------------

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

```haskell
ej1 :: Arbol Int
ej1 = N 1 (N 4 (H 0) (H 7)) (N 6 (H 4) (H 3))

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] Las matrices enteras se pueden representar
-- mediante tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
--
-- Definir la función
--    matrizPorBloques :: Matriz -> Matriz -> Matriz -> Matriz -> Matriz
-- tal que (matrizPorBloques p1 p2 p3 p4) es la matriz cuadrada de orden
-- 2nx2n construida con las matrices cuadradas de orden nxn p1, p2 p3 y
-- p4 de forma que p1 es su bloque superior izquierda, p2 es su bloque
-- superior derecha, p3 es su bloque inferior izquierda y p4 es su bloque
-- inferior derecha. Por ejemplo, si p1, p2, p3 y p4 son las matrices
-- definidas por
--    p1, p2, p3, p4 :: Matriz
--    p1 = listArray ((1,1),(2,2)) [1,2,3,4]
--    p2 = listArray ((1,1),(2,2)) [6,5,7,8]
--    p3 = listArray ((1,1),(2,2)) [0,6,7,1]
--    p4 = listArray ((1,1),(2,2)) [5,2,8,3]
-- entonces
--    ghci> matrizPorBloques p1 p2 p3 p4
--    array ((1,1),(4,4)) [((1,1),1),((1,2),2),((1,3),6),((1,4),5),
--                         ((2,1),3),((2,2),4),((2,3),7),((2,4),8),
--                         ((3,1),0),((3,2),6),((3,3),5),((3,4),2),
--                         ((4,1),7),((4,2),1),((4,3),8),((4,4),3)]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

p1, p2, p3, p4 :: Matriz
p1 = listArray ((1,1),(2,2)) [1,2,3,4]
p2 = listArray ((1,1),(2,2)) [6,5,7,8]
p3 = listArray ((1,1),(2,2)) [0,6,7,1]
p4 = listArray ((1,1),(2,2)) [5,2,8,3]
```

```haskell
matrizPorBloques :: Matriz -> Matriz -> Matriz -> Matriz -> Matriz
matrizPorBloques p1 p2 p3 p4 =
  array ((1,1),(m,m)) [((i,j), f i j) | i <- [1..m], j <- [1..m]]
  where ((_,_),(n,_)) = bounds p1
        m = 2*n
        f i j | i <= n && j <= n = p1!(i,j)
              | i <= n && j >  n = p2!(i,j-n)
              | i >  n && j <= n = p3!(i-n,j)
              | i >  n && j >  n = p4!(i-n,j-n)
```

1.9. Examen 9 (20 de noviembre de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 3º convocatoria (20 de noviembre de 2012)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Definir la función
--    mayorProducto :: Int -> [Int] -> Int
-- tal que (mayorProducto n xs) es el mayor producto de una sublista de
-- xs de longitud n. Por ejemplo,
--    mayorProducto 3 [3,2,0,5,4,9,1,3,7]  ==  180
-- ya que de todas las sublistas de longitud 3 de [3,2,0,5,4,9,1,3,7] la
-- que tiene mayor producto es la [5,4,9] cuyo producto es 180.
-- ---------------------------------------------------------------------

mayorProducto :: Int -> [Int] -> Int
mayorProducto n cs
    | length cs < n = 1
    | otherwise     = maximum [product xs | xs <- segmentos n cs]
   where segmentos n cs = [take n xs | xs <- tails cs]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sinDobleCero :: Int -> [[Int]]
-- tal que (sinDobleCero n) es la lista de las listas de longitud n
```

```haskell
-- formadas por el 0 y el 1 tales que no contiene dos ceros
-- consecutivos. Por ejemplo,
--    ghci> sinDobleCero 2
--    [[1,0],[1,1],[0,1]]
--    ghci> sinDobleCero 3
--    [[1,1,0],[1,1,1],[1,0,1],[0,1,0],[0,1,1]]
--    ghci> sinDobleCero 4
--    [[1,1,1,0],[1,1,1,1],[1,1,0,1],[1,0,1,0],[1,0,1,1],
--     [0,1,1,0],[0,1,1,1],[0,1,0,1]]
-- ---------------------------------------------------------------------

sinDobleCero :: Int -> [[Int]]
sinDobleCero 0 = [[]]
sinDobleCero 1 = [[0],[1]]
sinDobleCero n = [1:xs | xs <- sinDobleCero (n-1)] ++
                 [0:1:ys | ys <- sinDobleCero (n-2)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] La sucesión A046034 de la OEIS (The On-Line
-- Encyclopedia of Integer Sequences) está formada por los números tales
-- que todos sus dígitos son primos. Los primeros términos de A046034
-- son
--    2,3,5,7,22,23,25,27,32,33,35,37,52,53,55,57,72,73,75,77,222,223
--
-- Definir la constante
--    numerosDigitosPrimos :: [Int]
-- cuyos elementos son los términos de la sucesión A046034. Por ejemplo,
--    ghci> take 22 numerosDigitosPrimos
--    [2,3,5,7,22,23,25,27,32,33,35,37,52,53,55,57,72,73,75,77,222,223]
-- ¿Cuántos elementos hay en la sucesión menores que 2013?
-- ---------------------------------------------------------------------

numerosDigitosPrimos :: [Int]
numerosDigitosPrimos =
    [n | n <- [2..], digitosPrimos n]

-- (digitosPrimos n) se verifica si todos los dígitos de n son
-- primos. Por ejemplo,
--    digitosPrimos 352  ==  True
--    digitosPrimos 362  ==  False
```

```haskell
digitosPrimos :: Int -> Bool
digitosPrimos n = all (`elem` "2357") (show n)

-- 2ª definición de digitosPrimos:
digitosPrimos2 :: Int -> Bool
digitosPrimos2 n = subconjunto (cifras n) [2,3,5,7]

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

-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys. Por
-- ejemplo,
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]

-- El cálculo es
--    ghci> length (takeWhile (<2013) numerosDigitosPrimos)
--    84

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Entre dos matrices de la misma dimensión se
-- puede aplicar distintas operaciones binarias entre los elementos en
-- la misma posición. Por ejemplo, si a y b son las matrices
--    |3 4 6|     |1 4 2|
--    |5 6 7|     |2 1 2|
-- entonces a+b y a-b son, respectivamente
--    |4 8 8|     |2 0 4|
--    |7 7 9|     |3 5 5|
--
-- Las matrices enteras se pueden representar mediante tablas con
-- índices enteros:
--    type Matriz = Array (Int,Int) Int
-- y las matrices anteriores se definen por
--    a, b :: Matriz
--    a = listArray ((1,1),(2,3)) [3,4,6,5,6,7]
--    b = listArray ((1,1),(2,3)) [1,4,2,2,1,2]
--
-- Definir la función
--    opMatriz :: (Int -> Int -> Int) -> Matriz -> Matriz -> Matriz
```

```haskell
-- tal que (opMatriz f p q) es la matriz obtenida aplicando la operación
-- f entre los elementos de p y q de la misma posición. Por ejemplo,
--    ghci> opMatriz (+) a b
--    array ((1,1),(2,3)) [((1,1),4),((1,2),8),((1,3),8),
--                         ((2,1),7),((2,2),7),((2,3),9)]
--    ghci> opMatriz (-) a b
--    array ((1,1),(2,3)) [((1,1),2),((1,2),0),((1,3),4),
--                         ((2,1),3),((2,2),5),((2,3),5)]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

a, b :: Matriz
a = listArray ((1,1),(2,3)) [3,4,6,5,6,7]
b = listArray ((1,1),(2,3)) [1,4,2,2,1,2]

-- 1ª definición
opMatriz :: (Int -> Int -> Int) -> Matriz -> Matriz -> Matriz
opMatriz f p q =
    array ((1,1),(m,n)) [((i,j), f (p!(i,j)) (q!(i,j)))
                         | i <- [1..m], j <- [1..n]]
    where (_,(m,n)) = bounds p

-- 2ª definición
opMatriz2 :: (Int -> Int -> Int) -> Matriz -> Matriz -> Matriz
opMatriz2 f p q =
    listArray (bounds p) [f x y | (x,y) <- zip (elems p) (elems q)]

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] Las expresiones aritméticas se pueden definir
-- usando el siguiente tipo de datos
--    data Expr = N Int
--              | X
--              | S Expr Expr
--              | R Expr Expr
--              | P Expr Expr
--              | E Expr Int
--              deriving (Eq, Show)
-- Por ejemplo, la expresión
--    3*x - (x+2)^7
```

```haskell
-- se puede definir por
--    R (P (N 3) X) (E (S X (N 2)) 7)
--
-- Definir la función
--    maximo :: Expr -> [Int] -> (Int,[Int])
-- tal que (maximo e xs) es el par formado por el máximo valor de la
-- expresión e para los puntos de xs y en qué puntos alcanza el
-- máximo. Por ejemplo,
--    ghci> maximo (E (S (N 10) (P (R (N 1) X) X)) 2) [-3..3]
--    (100,[0,1])
-- ----------------------------------------------------------------------

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

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

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

2

Exámenes del grupo 2

Antonia M. Chávez

2.1. Examen 1 (7 de noviembre de 2012)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función ocurrenciasDelMaximo tal que
-- (ocurrenciasDelMaximo xs) es el par formado por el mayor de los
-- números de xs y el número de veces que este aparece en la lista
-- xs, si la lista es no vacía y es (0,0) si xs es la lista vacía. Por
-- ejemplo,
--    ocurrenciasDelMaximo [1,3,2,4,2,5,3,6,3,2,1,8,7,6,5]  ==  (8,1)
--    ocurrenciasDelMaximo [1,8,2,4,8,5,3,6,3,2,1,8]        ==  (8,3)
--    ocurrenciasDelMaximo [8,8,2,4,8,5,3,6,3,2,1,8]        ==  (8,4)
-- ---------------------------------------------------------------------

ocurrenciasDelMaximo [] = (0,0)
ocurrenciasDelMaximo xs = (maximum xs, sum [1 | y <- xs, y == maximum xs])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por comprensión, la función tienenS tal que
-- (tienenS xss) es la lista de las longitudes de las cadenas de xss que
-- contienen el caracter 's' en mayúsculas o minúsculas. Por ejemplo,
--    tienenS [”Este”,”es”,”un”,”examen”,”de”,”hoy”,”Suerte”]  ==  [4,2,6]
--    tienenS [”Este”]                                         ==  [4]
```

```haskell
--    tienenS []                                                    ==  []
--    tienenS [" "]                                                 ==  []
-- ---------------------------------------------------------------------

tienenS xss = [length xs | xs <- xss, (elem 's' xs) || (elem 'S' xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Decimos que una lista está algo ordenada si para todo
-- par de elementos consecutivos se cumple que el primero es menor o
-- igual que el doble del segundo. Definir, por comprensión, la función
-- (algoOrdenada xs) que se verifica si la lista xs está algo ordenada.
-- Por ejemplo,
--    algoOrdenada [1,3,2,5,3,8]  ==  True
--    algoOrdenada [3,1]          ==  False
-- ---------------------------------------------------------------------

algoOrdenada xs = and [x <= 2*y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por comprensión, la función tripletas tal que
-- (tripletas xs) es la listas de tripletas de elementos consecutivos de
-- la lista xs. Por ejemplo,
--    tripletas [8,7,6,5,4] == [[8,7,6],[7,6,5],[6,5,4]]
--    tripletas "abcd"      == ["abc","bcd"]
--    tripletas [2,4,3]     == [[2,3,4]]
--    tripletas [2,4]       == []
-- ---------------------------------------------------------------------

-- 1ª definición:
tripletas xs =
    [[a,b,c] | ((a,b),c) <- zip (zip xs (tail xs)) (tail (tail xs))]

-- 2ª definición:
tripletas2 xs =
    [[xs!!n,xs!!(n+1),xs!!(n+2)] | n <- [0..length xs -3]]

-- 3ª definición:
tripletas3 xs = [take 3 (drop n xs)| n <- [0..(length xs - 3)]]

-- Se puede definir por recursión
```

```
tripletas4 (x1:x2:x3:xs) = [x1,x2,x3] : tripletas (x2:x3:xs)
tripletas4 _             = []

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función tresConsecutivas tal que
-- (tresConsecutivas x ys) se verifica si x tres veces seguidas en la
-- lista ys. Por ejemplo,
--    tresConsecutivas 3 [1,4,2,3,3,4,3,5,3,4,6]  ==  Falsese
--    tresConsecutivas 'a' "abcaaadfg"            ==  True
-- ---------------------------------------------------------------------

tresConsecutivas x ys = elem [x,x,x] (tripletas ys)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Se dice que un número n es malo si el número 666 aparece
-- en 2^n. Por ejemplo, 157 y 192 son malos, ya que:
--    2^157 = 182687704666362864775460604089535377456991567872
--    2^192 = 6277101735386680763835789423207666416102355444464034512896
--
-- Definir una función (malo x) que se verifica si el número x es
-- malo. Por ejemplo,
--    malo 157  ==  True
--    malo 192  ==  True
--    malo 221  ==  False
-- ---------------------------------------------------------------------

malo n = tresConsecutivas '6' (show (2^n))
```

2.2. Examen 2 (19 de diciembre de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (19 de diciembre de 2012)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    maximaDiferenciaC :: [Integer] -> Integer
-- tal que (maximaDiferenciaC xs) es la mayor de las diferencias en
```

```haskell
-- valor absoluto entre elementos consecutivos de la lista xs. Por
-- ejemplo,
--    maximaDiferenciaC [2,5,-3]           ==  8
--    maximaDiferenciaC [1,5]              == 4
--    maximaDiferenciaC [10,-10,1,4,20,-2] == 22
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    maximaDiferenciaR :: [Integer] -> Integer
-- tal que (maximaDiferenciaR xs) es la mayor de las diferencias en
-- valor absoluto entre elementos consecutivos de la lista xs. Por
-- ejemplo,
--    maximaDiferenciaR [2,5,-3]           ==  8
--    maximaDiferenciaR [1,5]              == 4
--    maximaDiferenciaR [10,-10,1,4,20,-2] == 22
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que las definiciones
-- maximaDiferenciaC y maximaDiferenciaR son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_maximaDiferencia :: [Integer] -> Property
prop_maximaDiferencia xs =
    length xs > 1 ==> maximaDiferenciaC xs == maximaDiferenciaR xs

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por comprensión, la función acumuladaC tal
-- que (acumuladaC 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,
--    acumuladaC [2,5,1,4,3] == [2,7,8,12,15]
--    acumuladaC [1,-1,1,-1] == [1,0,1,0]
-- ---------------------------------------------------------------------

acumuladaC xs = [sum (take n xs) | n <- [1..length xs]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función acumuladaR tal que
-- (acumuladaR 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,
--    acumuladaR [2,5,1,4,3] == [2,7,8,12,15]
--    acumuladaR [1,-1,1,-1] == [1,0,1,0]
-- ---------------------------------------------------------------------

-- 1ª definición:
acumuladaR [] =  []
acumuladaR xs =  acumuladaR (init xs) ++ [sum xs]

-- 2ª definición:
acumuladaR2 [] = []
acumuladaR2 (x:xs) = reverse (aux xs [x])
    where aux [] ys = ys
          aux (x:xs) (y:ys) = aux xs (x+y:y:ys)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función unitarios tal (unitarios n) es
-- la lista de números [n,nn, nnn, ....]. Por ejemplo.
--    take 7 (unitarios 3) == [3,33,333,3333,33333,333333,3333333]
--    take 3 (unitarios 1) == [1,11,111]
-- ---------------------------------------------------------------------

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

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

```haskell
-- Ejercicio 3.2. Definir la función multiplosUnitarios tal que
-- (multiplosUnitarios x y n) es la lista de los n primeros múltiplos de
-- x cuyo único dígito es y. Por ejemplo,
--    multiplosUnitarios 7 1 2  == [111111,111111111111]
--    multiplosUnitarios 11 3 5 == [33,3333,333333,33333333,3333333333]
-- ---------------------------------------------------------------------

multiplosUnitarios x y n = take n [z | z <- unitarios y, mod z x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir, por recursión, la función inicialesDistintosR
-- tal que (inicialesDistintosR xs) es el número de elementos que hay en
-- xs antes de que aparezca el primer repetido. Por ejemplo,
--    inicialesDistintosR [1,2,3,4,5,3] == 2
--    inicialesDistintosR [1,2,3]       == 3
--    inicialesDistintosR "ahora"       == 0
--    inicialesDistintosR "ahorA"       == 5
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por comprensión, la función
-- inicialesDistintosC tal que (inicialesDistintosC xs) es el número de
-- elementos que hay en xs antes de que aparezca el primer repetido. Por
-- ejemplo,
--    inicialesDistintosC [1,2,3,4,5,3] == 2
--    inicialesDistintosC [1,2,3]       == 3
--    inicialesDistintosC "ahora"       == 0
--    inicialesDistintosC "ahorA"       == 5
-- ---------------------------------------------------------------------

inicialesDistintosC xs =
    length (takeWhile (==1) (listaOcurrencias xs))

-- (listaOcurrencias xs) es la lista con el número de veces que aparece
-- cada elemento de xs en xs. Por ejemplo,
```

```
--    listaOcurrencias [1,2,3,4,5,3]    ==  [1,1,2,1,1,2]
--    listaOcurrencias "repetidamente"  ==  [1,4,1,4,2,1,1,1,1,4,1,2,4]
listaOcurrencias xs = [ocurrencias x xs | x <- xs]

-- (ocurrencias x ys) es el número de ocurrencias de x en ys. Por
-- ejemplo,
--    ocurrencias 1 [1,2,3,1,5,3,3]  ==  2
--    ocurrencias 3 [1,2,3,1,5,3,3]  ==  3
ocurrencias x ys = length [y | y <- ys, x == y]
```

2.3. Examen 3 (6 de febrero de 2013)

El examen es común con el del grupo 1 (ver página 11).

2.4. Examen 4 (3 de abril de 2013)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 4º examen de evaluación continua (3 de abril de 2013)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Se denomina resto de una lista a una sublista no vacia
-- formada el último o últimos elementos. Por ejemplo, [3,4,5] es un
-- resto de lista [1,2,3,4,5].
--
-- Definir la función
--    restos :: [a] -> [[a]]
-- tal que (restos xs) es la lista de los restos de la lista xs. Por
-- ejemplo,
--    restos [2,5,6]  ==  [[2,5,6],[5,6],[6]]
--    restos [4,5]    ==  [[4,5],[5]]
--    restos []       ==  []
-- ---------------------------------------------------------------------

restos :: [a] -> [[a]]
restos []     = []
restos (x:xs) = (x:xs) : restos xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Se denomina corte de una lista a una sublista no vacía
-- formada por el primer elemento y los siguientes hasta uno dado.
-- Por ejemplo, [1,2,3] es un corte de [1,2,3,4,5].
--
-- Definir, por recursión, la función
--    cortesR :: [a] -> [[a]]
-- tal que (cortesR xs) es la lista de los cortes de la lista xs. Por
-- ejemplo,
--    cortesR []         ==  []
--    cortesR [2,5]      ==  [[2],[2,5]]
--    cortesR [4,8,6,0]  ==  [[4],[4,8],[4,8,6],[4,8,6,0]]
-- ---------------------------------------------------------------------

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

-- 2ª definición:
cortesR2 :: [a] -> [[a]]
cortesR2 []     = []
cortesR2 (x:xs) = [x] : map (\y -> x:y) (cortesR2 xs)

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por composición, la función
--    cortesC :: [a] -> [[a]]
-- tal que (cortesC xs) es la lista de los cortes de la lista xs. Por
-- ejemplo,
--    cortesC []         ==  []
--    cortesC [2,5]      ==  [[2],[2,5]]
--    cortesC [4,8,6,0]  ==  [[4],[4,8],[4,8,6],[4,8,6,0]]
-- ---------------------------------------------------------------------

cortesC :: [a] -> [[a]]
cortesC = reverse . map reverse . restos . reverse

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios se pueden representar con el de
```

```haskell
-- dato algebraico
--    data Arbol = H Int
--               | N Arbol Int Arbol
--               deriving (Show, Eq)
-- Por ejemplo, los árboles
--         9                9
--        / \              / \
--       /   \            /   \
--      8     6          7     3
--     / \   / \        / \   / \
--    3   2 4   5      3   2 4   7
-- se pueden representar por
--    ej1, ej2:: Arbol
--    ej1 = N (N (H 3) 8 (H 2)) 9 (N (H 4) 6 (H 5))
--    ej2 = N (N (H 3) 7 (H 2)) 9 (N (H 4) 3 (H 7))
--
-- Decimos que un árbol binario es par si la mayoría de sus nodos son
-- pares e impar en caso contrario. Por ejemplo, el primer ejemplo es
-- par y el segundo es impar.
--
-- Para representar la paridad se define el tipo Paridad
--    data Paridad = Par | Impar deriving Show
--
-- Definir la función
--    paridad :: Arbol -> Paridad
-- tal que (paridad a) es la paridad del árbol a. Por ejemplo,
--    paridad ej1  ==  Par
--    paridad ej2  ==  Impar
-- ---------------------------------------------------------------------

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

ej1, ej2:: Arbol
ej1 = N (N (H 3) 8 (H 2)) 9 (N (H 4) 6 (H 5))
ej2 = N (N (H 3) 7 (H 2)) 9 (N (H 4) 3 (H 7))

data Paridad = Par | Impar deriving Show
```

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

-- (paridades a) es un par (x,y) donde x es el número de valores pares
-- en el árbol a e i es el número de valores impares en el árbol a. Por
-- ejemplo,
--    paridades ej1  ==  (4,3)
--    paridades ej2  ==  (2,5)
paridades :: Arbol -> (Int,Int)
paridades (H x) | even x    = (1,0)
                | otherwise = (0,1)
paridades (N i x d) | even x    = (1+a1+a2,b1+b2)
                    | otherwise = (a1+a2,1+b1+b2)
                    where (a1,b1) = paridades i
                          (a2,b2) = paridades d

-- ---------------------------------------------------------------------
-- Ejercicio 3. Según la Wikipedia, un número feliz se define por el
-- siguiente proceso. Se comienza reemplazando el número por la suma del
-- cuadrado de sus cifras y se repite el proceso hasta que se obtiene el
-- número 1 o se entra en un ciclo que no contiene al 1. Aquellos
-- números para los que el proceso termina en 1 se llaman números
-- felices y los que entran en un ciclo sin 1 se llaman números
-- desgraciados.
--
-- Por ejemplo, 7 es un número feliz porque
--       7 ~> 7^2                          =  49
--         ~> 4^2 + 9^2       = 16 + 81    =  97
--         ~> 9^2 + 7^2       = 81 + 49    = 130
--         ~> 1^2 + 3^2 + 0^2 =  1 + 9 + 0 =  10
--         ~> 1^2 + 0^2       =  1 + 0     =   1
-- Pero 17 es un número desgraciado porque
--    17 ~> 1^2 + 7^2       =  1 + 49      =  50
--       ~> 5^2 + 0^2       = 25 +  0      =  25
--       ~> 2^2 + 5^2       =  4 + 25      =  29
--       ~> 2^2 + 9^2       =  4 + 81      =  85
--       ~> 8^2 + 5^2       = 64 + 25      =  89
--       ~> 8^2 + 9^2       = 64 + 81      = 145
```

```
--       ~> 1^2 + 4^2 + 5^2 =  1 + 16 + 25 =  42
--       ~> 4^2 + 2^2       = 16 +  4      =  20
--       ~> 2^2 + 0^2       =  4 +  0      =   4
--       ~> 4^2                            =  16
--       ~> 1^2 + 6^2       = 1 + 36       =  37
--       ~> 3^2 + 7^2       = 9 + 49       =  58
--       ~> 5^2 + 8^2       = 25 + 64      =  89
-- que forma un bucle al repetirse el 89.
--
-- El objetivo del ejercicio es definir una función que calcule todos
-- los números felices hasta un límite dado.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    sumaCuadrados :: Int -> Int
-- tal que (sumaCuadrados n) es la suma de los cuadrados de los dígitos
-- de n. Por ejemplo,
--    sumaCuadrados 145  ==  42
-- ---------------------------------------------------------------------

sumaCuadrados :: Int -> Int
sumaCuadrados n = sum [x^2 | x <- digitos n]

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 145  ==  [1,4,5]
digitos :: Int -> [Int]
digitos n = [read [x]|x<-show n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    caminoALaFelicidad :: Int -> [Int]
-- tal que (caminoALaFelicidad n) es la lista de los números obtenidos
-- en el proceso de la determinación si n es un número feliz: se
-- comienza con la lista [n], ampliando la lista con la suma del
-- cuadrado de las cifras de su primer elemento y se repite el proceso
-- hasta que se obtiene el número 1 o se entra en un ciclo que no
-- contiene al 1. Por ejemplo,
--    ghci> take 20 (caminoALaFelicidad 7)
--    [7,49,97,130,10,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

```
--    ghci> take 20 (caminoALaFelicidad 17)
--    [17,50,25,29,85,89,145,42,20,4,16,37,58,89,145,42,20,4,16,37]
-- ---------------------------------------------------------------------

caminoALaFelicidad :: Int -> [Int]
caminoALaFelicidad n =
    n : [sumaCuadrados x | x <- caminoALaFelicidad n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. En el camino a la felicidad, pueden ocurrir dos casos:
-- + aparece un 1 y a continuación solo aparece 1,
-- + llegamos a 4 y se entra en el ciclo 4,16,37,58,89,145,42,20.
--
-- Definir la función
--    caminoALaFelicidadFundamental :: Int -> [Int]
-- tal que (caminoALaFelicidadFundamental n) es el camino de la
-- felicidad de n hasta que aparece un 1 o un 4. Por ejemplo,
--    caminoALaFelicidadFundamental 34    == [34,25,29,85,89,145,42,20,4]
--    caminoALaFelicidadFundamental 203   == [203,13,10,1]
--    caminoALaFelicidadFundamental 23018 == [23018,78,113,11,2,4]
-- ---------------------------------------------------------------------

caminoALaFelicidadFundamental :: Int -> [Int]
caminoALaFelicidadFundamental n = selecciona (caminoALaFelicidad n)

-- (selecciona xs) es la lista de los elementos hasta que aparece un 1 o
-- un 4. Por ejemplo,
--    selecciona [3,2,1,5,4]  ==  [3,2,1]
--    selecciona [3,2]        ==  [3,2]
selecciona [] = []
selecciona (x:xs) | x == 1 || x == 4 = [x]
                  | otherwise        = x : selecciona xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir la función
--    esFeliz :: Int -> Bool
-- tal que (esFeliz n) s verifica si n es feliz. Por ejemplo,
--    esFeliz  7  ==  True
--    esFeliz 17  ==  False
-- ---------------------------------------------------------------------
```

```haskell
esFeliz :: Int -> Bool
esFeliz n = last (caminoALaFelicidadFundamental n) == 1

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Comprobar con QuickCheck que si n es es feliz,
-- entonces todos los números de (caminoALaFelicidadFundamental n)
-- también lo son.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_esFeliz :: Int -> Property
prop_esFeliz n =
    n>0 && esFeliz n
    ==> and [esFeliz x | x <- caminoALaFelicidadFundamental n]

-- La comprobación es
--    ghci> quickCheck prop_esFeliz
--    *** Gave up! Passed only 38 tests.
```

2.5. Examen 5 (15 de mayo de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 5º examen de evaluación continua (22 de mayo de 2013)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Array
import PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    conFinales :: Int -> [Int] -> [Int]
-- tal que (conFinales x xs) es la lista de los elementos de xs que
-- terminan en x. Por ejemplo,
--    conFinales 2 [31,12,7,142,214]  ==  [12,142]
```

```haskell
-- Dar cuatro definiciones distintas: recursiva, por comprensión, con
-- filtrado y por plegado.
-- ---------------------------------------------------------------------

-- 1ª definición (recursiva):
conFinales1 :: Int -> [Int] -> [Int]
conFinales1 x [] = []
conFinales1 x (y:ys) | mod y 10 == x = y : conFinales1 x ys
                     | otherwise     = conFinales1 x ys

-- 2ª definición (por comprensión):
conFinales2 :: Int -> [Int] -> [Int]
conFinales2 x xs = [y | y <- xs, mod y 10 == x]

-- 3ª definición (por filtrado):
conFinales3 :: Int -> [Int] -> [Int]
conFinales3 x xs = filter (\z -> mod z 10 == x) xs

-- 4ª definición (por plegado):
conFinales4 :: Int -> [Int] -> [Int]
conFinales4 x = foldr f []
    where f y ys | mod y 10 == x = y:ys
                 | otherwise     = ys

-- ---------------------------------------------------------------------
-- Ejercicio 2. (OME 2010) Una sucesión pucelana es una sucesión
-- creciente de dieciseis números impares positivos consecutivos, cuya
-- suma es un cubo perfecto.
--
-- Definir la función
--    pucelanasDeTres :: [[Int]]
-- tal que pucelanasDeTres es la lista de la sucesiones pucelanas
-- formadas por números de tres cifras. Por ejemplo,
--    ghci> take 2 pucelanasDeTres
--    [[241,243,245,247,249,251,253,255,257,259,261,263,265,267,269,271],
---    [485,487,489,491,493,495,497,499,501,503,505,507,509,511,513,515]]
-- ¿Cuántas sucesiones pucelanas tienen solamente números de tres
-- cifras?
-- ---------------------------------------------------------------------
```

```haskell
pucelanasDeTres :: [[Int]]
pucelanasDeTres = [[x,x+2 .. x+30] | x <- [101, 103 .. 999-30],
                                     esCubo (sum [x,x+2 .. x+30])]

esCubo x = or [y^3 == x | y <- [1..x]]

-- El número se calcula con
--    ghci> length pucelanasDeTres
--    3


-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función:
--    extraePares :: Polinomio Integer -> Polinomio Integer
-- tal que (extraePares p) es el polinomio que resulta de extraer los
-- monomios de grado par de p. Por ejemplo, si p es el polinomio
-- x^4 + 5*x^3 + 7*x^2 + 6*x, entonces (extraePares p) es
-- x^4 + 7*x^2.
--    > let p1 = consPol 4 1 (consPol 3 5 (consPol 2 7 (consPol 1 6 polCero)))
--    > p1
--    x^4 + 5*x^3 + 7*x^2 + 6*x
--    > extraePares p1
--    x^4 + 7*x^2
-- ---------------------------------------------------------------------

extraePares :: Polinomio Integer -> Polinomio Integer
extraePares p
    | esPolCero p = polCero
    | even n      = consPol n (coefLider p) (extraePares rp)
    | otherwise   = extraePares rp
    where n  = grado p
          rp = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    rellenaPol :: Polinomio Integer -> Polinomio Integer
-- tal que (rellenaPol p) es el polinomio obtenido completando con
-- monomios del tipo 1*x^n aquellos monomios de grado n que falten en
-- p. Por ejemplo,
--    ghci> let p1 = consPol 4 2 (consPol 2 1 (consPol 0 5 polCero))
```

```
--    ghci> p1
--    2*x^4 + x^2 + 5
--    ghci> rellenaPol p1
--    2*x^4 + x^3 + x^2 + 1*x + 5
-- ---------------------------------------------------------------------

rellenaPol :: Polinomio Integer -> Polinomio Integer
rellenaPol p
    | n == 0 = p
    | n == grado r + 1 = consPol n c (rellenaPol r)
    | otherwise = consPol n c (consPol (n-1) 1 (rellenaPol r))
    where n = grado p
          c = coefLider p
          r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Consideremos el tipo de las matrices
--    type Matriz a = Array (Int,Int) a
-- y, para los ejemplos, la matriz
--    m1 :: Matriz Int
--    m1 = array ((1,1),(3,3))
--               [((1,1),1),((1,2),0),((1,3),1),
--                ((2,1),0),((2,2),1),((2,3),1),
--                ((3,1),1),((3,2),1),((3,3),1)])

-- Definir la función
--    cambiaM :: (Int, Int) -> Matriz Int -> Matriz Int
-- tal que (cambiaM i p) es la matriz obtenida cambiando en p los
-- elementos de la fila y la columna en i transformando los 0 en 1 y
-- viceversa. El valor en i cambia solo una vez. Por ejemplo,
--    ghci> cambiaM (2,3) m1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                         ((2,1),1),((2,2),7),((2,3),0),
--                         ((3,1),1),((3,2),1),((3,3),0)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

m1 :: Matriz Int
m1 = array ((1,1),(3,3))
```

```
           [((1,1),1),((1,2),0),((1,3),1),
            ((2,1),0),((2,2),7),((2,3),1),
            ((3,1),1),((3,2),1),((3,3),1)]

cambiaM :: (Int, Int) -> Matriz Int -> Matriz Int
cambiaM (a,b) p = array (bounds p) [((i,j),f i j) | (i,j) <- indices p]
      where f i j  | i == a || j == b = cambia (p!(i,j))
                   | otherwise = p!(i,j)
            cambia x | x == 0    = 1
                     | x == 1    = 0
                     | otherwise = x

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    quitaRepetidosFila :: Int -> Matriz Int -> Matriz Int
-- tal que (quitaRepetidosFila i p) es la matriz obtenida a partir de p
-- eliminando los elementos repetidos de la fila i y rellenando con
-- ceros al final hasta completar la fila. Por ejemplo,
--    ghci> m1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                         ((2,1),0),((2,2),7),((2,3),1),
--                         ((3,1),1),((3,2),1),((3,3),1)]
--    ghci> quitaRepetidosFila 1 m1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                         ((2,1),0),((2,2),7),((2,3),1),
--                         ((3,1),1),((3,2),1),((3,3),1)]
--    ghci> quitaRepetidosFila 2 m1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                         ((2,1),0),((2,2),7),((2,3),1),
--                         ((3,1),1),((3,2),1),((3,3),1)]
--    ghci> quitaRepetidosFila 3 m1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                         ((2,1),0),((2,2),7),((2,3),1),
--                         ((3,1),1),((3,2),0),((3,3),0)]
-- ---------------------------------------------------------------------

quitaRepetidosFila :: Int -> Matriz Int -> Matriz Int
quitaRepetidosFila x p =
    array (bounds p) [((i,j),f i j) | (i,j) <- indices p]
    where f i j | i == x    = (cambia (fila i p)) !! (j-1)
```

```
                | otherwise = p!(i,j)

-- (fila i p) es la fila i-ésima de la matriz p. Por ejemplo,
--    ghci> m1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                         ((2,1),0),((2,2),7),((2,3),1),
--                         ((3,1),1),((3,2),1),((3,3),1)]
--    ghci> fila 2 m1
--    [0,7,1]
fila :: Int -> Matriz Int -> [Int]
fila i p = [p!(i,j) | j <- [1..n]]
    where (_,(_,n)) = bounds p

-- (cambia xs) es la lista obtenida eliminando los elementos repetidos
-- de xs y completando con ceros al final para que tenga la misma
-- longitud que xs. Por ejemplo,
--   cambia [2,3,2,5,3,2]  ==  [2,3,5,0,0,0]
cambia :: [Int] -> [Int]
cambia xs = ys ++ replicate (n-m) 0
    where ys = nub xs
          n  = length xs
          m  = length ys
```

2.6. Examen 6 (13 de junio de 2013)

```
-- Informática (1º del Grado en Matemáticas, Grupos 1 y 4)
-- 6º examen de evaluación continua (13 de junio de 2013)
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número es alternado si cifras son par/impar
-- alternativamente. Por ejemplo, 123456 y 2785410 son alternados.
--
-- Definir la función
--    numerosAlternados :: [Integer] -> [Integer]
-- tal que (numerosAlternados xs) es la lista de los números alternados
-- de xs. Por ejemplo,
```

```
--    ghci> numerosAlternados [21..50]
--    [21,23,25,27,29,30,32,34,36,38,41,43,45,47,49,50]
-- Usando la definición de numerosAlternados calcular la cantidad de
-- números alternados de 3 cifras.
-- ---------------------------------------------------------------------

-- 1ª definición (por comprension):
numerosAlternados :: [Integer] -> [Integer]
numerosAlternados xs = [n | n <- xs, esAlternado (cifras n)]

-- (esAlternado xs) se verifica si los elementos de xs son par/impar
-- alternativamente. Por ejemplo,
--   esAlternado [1,2,3,4,5,6]    ==  True
--   esAlternado [2,7,8,5,4,1,0]  ==  True
esAlternado :: [Integer] -> Bool
esAlternado [_] = True
esAlternado xs = and [odd (x+y) | (x,y) <- zip xs (tail xs)]

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

-- El cálculo es
--    ghci> length (numerosAlternados [100..999])
--    225

-- 2ª definición (por filtrado):
numerosAlternados2 :: [Integer] -> [Integer]
numerosAlternados2 = filter (\n -> esAlternado (cifras n))

-- la definición anterior se puede simplificar:
numerosAlternados2' :: [Integer] -> [Integer]
numerosAlternados2' = filter (esAlternado . cifras)

-- 3ª definición (por recursion):
numerosAlternados3 :: [Integer] -> [Integer]
numerosAlternados3 [] = []
numerosAlternados3 (n:ns)
  | esAlternado (cifras n) = n : numerosAlternados3 ns
```

```
  | otherwise              = numerosAlternados3 ns

-- 4ª definición (por plegado):
numerosAlternados4 :: [Integer] -> [Integer]
numerosAlternados4 = foldr f []
  where f n ns | esAlternado (cifras n) = n : ns
               | otherwise              = ns

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    borraSublista :: Eq a => [a] -> [a] -> [a]
-- tal que (borraSublista xs ys) es la lista que resulta de borrar la
-- primera ocurrencia de la sublista xs en ys. Por ejemplo,
--    borraSublista [2,3] [1,4,2,3,4,5]      == [1,4,4,5]
--    borraSublista [2,4] [1,4,2,3,4,5]      == [1,4,2,3,4,5]
--    borraSublista [2,3] [1,4,2,3,4,5,2,3]  ==  [1,4,4,5,2,3]
-- ---------------------------------------------------------------------

borraSublista :: Eq a => [a] -> [a] -> [a]
borraSublista [] ys = ys
borraSublista _ []  = []
borraSublista (x:xs) (y:ys)
    | esPrefijo (x:xs) (y:ys) = drop (length xs) ys
    | otherwise               = y : borraSublista (x:xs) ys

-- (esPrefijo xs ys) se verifica si xs es un prefijo de ys. Por ejemplo,
--    esPrefijo [2,5] [2,5,7,9]  ==  True
--    esPrefijo [2,5] [2,7,5,9]  ==  False
--    esPrefijo [2,5] [7,2,5,9]  ==  False
esPrefijo :: Eq a => [a] -> [a] -> Bool
esPrefijo [] ys         = True
esPrefijo _ []          = False
esPrefijo (x:xs) (y:ys) = x==y && esPrefijo xs ys

-- .....................................................................
-- Ejercicio 3. Dos números enteros positivos a y b se dicen "parientes"
-- si la suma de sus divisores coincide. Por ejemplo, 16 y 25 son
-- parientes ya que sus divisores son [1,2,4,8,16] y [1,5,25],
-- respectivamente, y 1+2+4+8+16 = 1+5+25.
--
```

```haskell
-- Definir la lista infinita
--     parientes :: [(Int,Int)]
-- que contiene los pares (a,b) de números parientes tales que
-- 1 <= a < b. Por ejemplo,
--    take 5 parientes  ==  [(6,11),(14,15),(10,17),(14,23),(15,23)]
-- ------------------------------------------------------------------

parientes :: [(Int,Int)]
parientes = [(a,b) | b <- [1..], a <- [1..b-1], sonParientes a b]

-- (sonParientes a b) se verifica si a y b son parientes. Por ejemplo,
--    sonParientes 16 25  ==  True
sonParientes :: Int -> Int -> Bool
sonParientes a b = sum (divisores a) == sum (divisores b)

-- (divisores a) es la lista de los divisores de a. Por ejemplo,
--    divisores 16  ==  [1,2,4,8,16]
--    divisores 25  ==  [1,5,25]
divisores :: Int -> [Int]
divisores a = [x | x <- [1..a], rem a x == 0]

-- ------------------------------------------------------------------
-- Ejercicio 4.1. Los árboles binarios se pueden representar con el de
-- dato algebraico
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
-- Por ejemplo, los árboles
--         9                9
--        / \              / \
--       /   \            /   \
--      8     6          7     9
--     / \   / \        / \   / \
--    3   2 4   5      3   2 9   7
-- se pueden representar por
--    ej1, ej2:: Arbol Int
--    ej1 = N 9 (N 8 (H 3) (H 2)) (N 6 (H 4) (H 5))
--    ej2 = N 9 (N 7 (H 3) (H 2)) (N 9 (H 9) (H 7))
--
-- Definir la función
--    nodosInternos :: Arbol t -> [t]
```

```haskell
-- tal que (nodosInternos a) es la lista de los nodos internos del
-- árbol a. Por ejemplo,
--    nodosInternos ej1  ==  [9,8,6]
--    nodosInternos ej2  ==  [9,7,9]
-- ...........................................................................

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

ej1, ej2:: Arbol Int
ej1 = N 9 (N 8 (H 3) (H 2)) (N 6 (H 4) (H 5))
ej2 = N 9 (N 7 (H 3) (H 2)) (N 9 (H 9) (H 7))

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    ramaIguales :: Eq t => Arbol t -> Bool
-- tal que (ramaIguales a) se verifica si el árbol a contiene al menos
-- una rama tal que todos sus elementos son iguales. Por ejemplo,
--    ramaIguales ej1  ==  False
--    ramaIguales ej2  ==  True
-- ---------------------------------------------------------------------

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

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

-- (ramas a) es la lista de las ramas del árbol a. Por ejemplo,
--    ramas ej1  ==  [[9,8,3],[9,8,2],[9,6,4],[9,6,5]]
--    ramas ej2  ==  [[9,7,3],[9,7,2],[9,9,9],[9,9,7]]
ramas :: Arbol a -> [[a]]
```

```haskell
ramas (H x)     = [[x]]
ramas (N x i d) = map (x:) (ramas i) ++ map (x:) (ramas d)

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

-- Otra definición de iguales, por comprensión, es
iguales2 :: Eq a => [a] -> Bool
iguales2 [] = True
iguales2 (x:xs) = and [x == y | y <- xs]

-- Otra, usando nub, es
iguales3 :: Eq a => [a] -> Bool
iguales3 xs = length (nub xs) <= 1

-- 3ª solución:
ramaIguales3 :: Eq a => Arbol a -> Bool
ramaIguales3  = any iguales . ramas

-- ---------------------------------------------------------------------
-- Ejercicio 5. Las matrices enteras se pueden representar mediante
-- tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
-- Por ejemplo, la matriz
--    |0 1 3|
--    |1 2 0|
--    |0 5 7|
-- se puede definir por
--    m :: Matriz
--    m = listArray ((1,1),(3,3)) [0,1,3, 1,2,0, 0,5,7]
--
-- Definir la función
--    sumaVecinos :: Matriz -> Matriz
-- tal que (sumaVecinos p) es la matriz obtenida al escribir en la
-- posicion (i,j) la suma de los todos vecinos del elemento que ocupa
```

```haskell
-- el lugar (i,j) en la matriz p. Por ejemplo,
--    ghci> sumaVecinos m
--    array ((1,1),(3,3)) [((1,1),4),((1,2), 6),((1,3), 3),
--                         ((2,1),8),((2,2),17),((2,3),18),
--                         ((3,1),8),((3,2),10),((3,3), 7)]
-- ----------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

m :: Matriz
m = listArray ((1,1),(3,3)) [0,1,3, 1,2,0, 0,5,7]

sumaVecinos :: Matriz -> Matriz
sumaVecinos p =
    array ((1,1),(m,n))
          [((i,j), f i j) | i <- [1..m], j <- [1..n]]
    where (_,(m,n)) = bounds p
          f i j = sum [p!(i+a,j+b) | a <- [-1..1], b <- [-1..1],
                                     a /= 0 || b /= 0,
                                     inRange (bounds p) (i+a,j+b)]
```

2.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 1 (ver página 26).

2.8. Examen 8 (13 de septiembre de 2013)

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

2.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 1 (ver página 38).

3

Exámenes del grupo 3

María J. Hidalgo

3.1. Examen 1 (16 de noviembre de 2012)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 1º examen de evaluación continua (15 de noviembre de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función numeroPrimos, donde (numeroPrimos m n)
-- es la cantidad de número primos entre 2^m y 2^n. Por ejemplo,
--   numerosPrimos 2 6   == 16
--   numerosPrimos 2 7   == 29
--   numerosPrimos 10 12 == 392
-- ---------------------------------------------------------------------

numerosPrimos:: Int -> Int -> Int
numerosPrimos m n = length [x | x <- [2^m..2^n], primo x]

-- (primo x) se verifica si x es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
primo n = factores n == [1, n]

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función masOcurrentes tal que
-- (masOcurrentes xs) es la lista de los elementos de xs que ocurren el
-- máximo número de veces. Por ejemplo,
--    masOcurrentes [1,2,3,4,3,2,3,1,4] == [3,3,3]
--    masOcurrentes [1,2,3,4,5,2,3,1,4] == [1,2,3,4,2,3,1,4]
--    masOcurrentes "Salamanca"         == "aaaa"
-- ---------------------------------------------------------------------

masOcurrentes xs = [x | x <- xs, ocurrencias x xs == m]
    where m = maximum [ocurrencias x xs | x <-xs]

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. En este esjercicio se consideran listas de ternas de
-- la forma (nombre, edad, población).
--
-- Definir la función puedenVotar tal que (puedenVotar t) es la
-- lista de las personas de t que tienen edad para votar. Por ejemplo,
--    ghci> :{
--    *Main| puedenVotar [("Ana", 16, "Sevilla"), ("Juan", 21, "Coria"),
--    *Main|              ("Alba", 19, "Camas"), ("Pedro",18,"Sevilla")]
--    *Main| :}
--    ["Juan","Alba","Pedro"]
-- ---------------------------------------------------------------------

puedenVotar t = [x | (x,y,_) <- t, y >= 18]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función puedenVotarEn tal que (puedenVotar
-- t p) es la lista de las personas de t que pueden votar en la
-- población p. Por ejemplo,
--    ghci> :{
--    *Main| puedenVotarEn [("Ana", 16, "Sevilla"), ("Juan", 21, "Coria"),
--    *Main|                ("Alba", 19, "Camas"),("Pedro",18,"Sevilla")]
--    *Main|               "Sevilla"
```

```
--    *Main| :}
--    ["Pedro"]
-- ---------------------------------------------------------------------

puedenVotarEn t c = [x | (x,y,z) <- t, y >= 18, z == c]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Dos listas xs, ys de la misma longitud son
-- perpendiculares si el producto escalar de ambas es 0, donde el
-- producto escalar de dos listas de enteros xs e ys viene
-- dado por la suma de los productos de los elementos correspondientes.
--
-- Definir la función perpendiculares tal que (perpendiculares xs yss)
-- es la lista de los elementos de yss que son perpendiculares a xs.
-- Por ejemplo,
--    ghci> perpendiculares [1,0,1] [[0,1,0], [2,3,1], [-1,7,1],[3,1,0]]
--    [[0,1,0],[-1,7,1]]
-- ---------------------------------------------------------------------

perpendiculares xs yss = [ys | ys <-yss, productoEscalar xs ys == 0]

-- (productoEscalar xs ys) es el producto escalar de xs por ys. Por
-- ejemplo,
--    productoEscalar [2,3,5] [6,0,2]  ==  22
productoEscalar xs ys = sum [x*y | (x,y) <- zip xs ys]
```

3.2. Examen 2 (21 de diciembre de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 2º examen de evaluación continua (21 de diciembre de 2012)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función f
--   f :: Int -> Integer
-- tal que (f k) es el menor número natural x tal que x^k comienza
-- exactamente por k unos. Por ejemplo,
```

```haskell
--    f 3 = 481
--    f 4 = 1826
-- ---------------------------------------------------------------------

f :: Int -> Integer
f 1 = 1
f k = head [x | x <- [1..], empiezaCon1 k (x^k)]

-- (empiezaCon1 k n) si el número x empieza exactamento con k unos. Por
-- ejemplo,
--    empiezaCon1 3 111461  ==  True
--    empiezaCon1 3 111146  ==  False
--    empiezaCon1 3 114116  ==  False
empiezaCon1 :: Int -> Integer -> Bool
empiezaCon1 k n = length (takeWhile (==1) (cifras n)) == k

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 111321  ==  [1,1,1,3,2,1]
cifras:: Integer -> [Integer]
cifras n = [read [x]| x <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función verificaE tal que
-- (verificaE k ps x) se cumple si x verifica exactamente k propiedades
-- de la lista ps. Por ejemplo,
--    verificaE 2 [(>0),even,odd] 5  ==  True
--    verificaE 1 [(>0),even,odd] 5  ==  False
-- ---------------------------------------------------------------------

verificaE :: Int -> [t -> Bool] -> t -> Bool
verificaE k ps x = length [p | p <- ps, p x] == k

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función verificaA tal que
-- (verificaA k ps x) se cumple si x verifica, como máximo, k
-- propiedades de la lista ps. Por ejemplo,
--    verificaA 2 [(>10),even,(<20)] 5    == True
--    verificaA 2 [(>0),even,odd,(<20)] 5 == False
-- ---------------------------------------------------------------------
```

```haskell
verificaA :: Int -> [t -> Bool] -> t -> Bool
verificaA k ps x = length [p | p <- ps, p x] <= k

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función verificaE tal que
-- (verificaE k ps x) se cumple si x verifica, al menos, k propiedades
-- de la lista ps. Por ejemplo,
--    verificaM 2 [(>0),even,odd,(<20)] 5 == True
--    verificaM 4 [(>0),even,odd,(<20)] 5   ==  False
-- ---------------------------------------------------------------------

verificaM :: Int -> [t -> Bool] -> t -> Bool
verificaM k ps x = length [p | p <- ps, p x] >= k

-- Nota: Otra forma de definir las funciones anteriores es la siguiente

verificaE2 k ps x = verifica ps x == k

verificaA2 k ps x = verifica ps x >= k

verificaM2 k ps x = verifica ps x <= k

-- donde (verifica ps x) es el número de propiedades de ps que verifica
-- el elemento x. Por ejemplo,
--    verifica [(>0),even,odd,(<20)] 5   ==  3
verifica ps x = sum [1 | p <- ps, p x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función intercalaDigito tal que
-- (intercalaDigito d n) es el número que resulta de intercalar el
-- dígito d delante de los dígitos de n menores que d. Por ejemplo,
--    intercalaDigito 5 1263709 == 51526537509
--    intercalaDigito 5 6798    == 6798
-- ---------------------------------------------------------------------

intercalaDigito :: Integer -> Integer -> Integer
intercalaDigito d n = listaNumero (intercala d (cifras n))

-- (intercala y xs) es la lista que resulta de intercalar el
-- número y delante de los elementos de xs menores que y. Por ejemplo,
```

```haskell
--    intercala 5 [1,2,6,3,7,0,9]  ==  [5,1,5,2,6,5,3,7,5,0,9]
intercala y [] = []
intercala y (x:xs) | x < y     = y : x : intercala y xs
                   | otherwise = x : intercala y xs

-- (listaNumero xs) es el número correspondiente a la lista de dígitos
-- xs. Por ejemplo,
--    listaNumero [5,1,5,2,6,5,3,7,5,0,9]  ==  51526537509
listaNumero :: [Integer] -> Integer
listaNumero xs = sum [x*(10^k) | (x,k) <- zip (reverse xs) [0..n]]
    where n = length xs -1

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. (Problema 302 del Proyecto Euler) Un número natural n
-- es se llama fuerte si p^2 es un divisor de n, para todos los factores
-- primos de n.
--
-- Definir la función
--    esFuerte :: Int -> Bool
-- tal que (esFuerte n) se verifica si n es fuerte. Por ejemplo,
--    esFuerte 800      == True
--    esFuerte 24       == False
--    esFuerte 14567429 == False
-- ---------------------------------------------------------------------

-- 1ª definición (directa)
-- =======================

esFuerte :: Int -> Bool
esFuerte n = and [rem n (p*p) == 0 | p <- xs]
    where xs = [p | p <- takeWhile (<=n) primos, rem n p == 0]

-- primos es la lista de los números primos.
primos :: [Int]
primos = 2 : [x | x <- [3,5..], esPrimo x]

-- (esPrimo x) se verifica si x es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 9  ==  False
esPrimo :: Int -> Bool
```

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

-- 2ª definición (usando la factorización de n)
-- ============================================

esFuerte2 :: Int -> Bool
esFuerte2 n = and [rem n (p*p) == 0 | (p,_) <- factorizacion n]

-- (factorización n) es la factorización de n. Por ejemplo,
--    factorizacion 300  ==  [(2,2),(3,1),(5,2)]
factorizacion :: Int -> [(Int,Int)]
factorizacion n =
    [(head xs, fromIntegral (length xs)) | xs <- group (factorizacion' n)]

-- (factorizacion' n) es la lista de todos los factores primos de n; es
-- decir, es una lista de números primos cuyo producto es n. Por ejemplo,
--    factorizacion 300  ==  [2,2,3,5,5]
factorizacion' :: Int -> [Int]
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 :: Int -> Int
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- Comparación de eficiencia:
-- ==========================

--    ghci> :set +s
--    ghci> esFuerte 14567429
--    False
--    (0.90 secs, 39202696 bytes)
--    ghci> esFuerte2 14567429
--    False
--    (0.01 secs, 517496 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    esPotencia:: Int -> Bool
-- tal que (esPotencia n) se verifica si n es potencia de algún número
-- entero. Por ejemplo,
--   esPotencia 81   == True
--   esPotencia 1234 == False
-- ---------------------------------------------------------------------

-- 1ª definición:
-- ==============

esPotencia:: Int -> Bool
esPotencia n = esPrimo n || or [esPotenciaDe n m | m  <- [0..n-1]]

-- (esPotenciaDe n m) se verifica si n es una potencia de m. Por
-- ejemplo,
--    esPotenciaDe 16 2  ==  True
--    esPotenciaDe 24 2  ==  False
esPotenciaDe:: Int -> Int -> Bool
esPotenciaDe n m = or [m^k == n | k <- [0..n]]

-- 2ª definición
-- =============

esPotencia2 :: Int -> Bool
esPotencia2 1 = True
esPotencia2 n = or [esPotenciaDe2 n m | m <- [2..n-1]]

-- (esPotenciaDe2 n m) se verifica si n es una potencia de m. Por
-- ejemplo,
--    esPotenciaDe2 16 2  ==  True
--    esPotenciaDe2 24 2  ==  False
esPotenciaDe2 :: Int -> Int -> Bool
esPotenciaDe2 n 1 = n == 1
esPotenciaDe2 n m = aux 1
  where aux k | y == n    = True
              | y > n     = False
              | otherwise = aux (k+1)
              where y = m^k
```

```haskell
-- 3ª definición
-- =============

esPotencia3 :: Int -> Bool
esPotencia3 n = todosIguales [x | (_,x) <- factorizacion n]

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    todosIguales [2,2,2]  ==  True
--    todosIguales [2,3,2]  ==  False
todosIguales :: [Int] -> Bool
todosIguales []       = True
todosIguales [_]      = True
todosIguales (x:y:xs) = x == y && todosIguales (y:xs)

-- Comparación de eficiencia
-- =========================

--    ghci> :set +s
--    ghci> esPotencia 1234
--    False
--    (16.87 secs, 2476980760 bytes)
--    ghci> esPotencia2 1234
--    False
--    (0.03 secs, 1549232 bytes)
--    ghci> esPotencia3 1234
--    True
--    (0.01 secs, 520540 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Un número natural se llama número de Aquiles si es
-- fuerte, pero no es una potencia perfecta; es decir, no es potencia de
-- un número. Por ejemplo, 864 y 1800 son números de Aquiles, pues
-- 864 = 2^5·3^3 y 1800 = 2^3·3^2·5^2.
--
-- Definir la función
--    esAquileo:: Int -> Bool
-- tal que (esAquileo n) se verifica si n es fuerte y no es potencia
-- perfecta. Por ejemplo,
--    esAquileo 864  ==  True
```

```haskell
--    esAquileo 865  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición:
esAquileo :: Int -> Bool
esAquileo n = esFuerte n && not (esPotencia n)

-- 2ª definición:
esAquileo2 :: Int -> Bool
esAquileo2 n = esFuerte2 n && not (esPotencia2 n)

-- 3ª definición:
esAquileo3 :: Int -> Bool
esAquileo3 n = esFuerte2 n && not (esPotencia3 n)

-- Comparación de eficiencia
-- =========================

--    ghci> take 10 [n | n <- [1..], esAquileo n]
--    [72,108,200,288,392,432,500,648,675,800]
--    (24.69 secs, 3495004684 bytes)
--    ghci> take 10 [n | n <- [1..], esAquileo2 n]
--    [72,108,200,288,392,432,500,648,675,800]
--    (0.32 secs, 12398516 bytes)
--    ghci> take 10 [n | n <- [1..], esAquileo3 n]
--    [72,108,144,200,288,324,392,400,432,500]
--    (0.12 secs, 3622968 bytes)
```

3.3. Examen 3 (6 de febrero de 2013)

El examen es común con el del grupo 1 (ver página 11).

3.4. Examen 4 (22 de marzo de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 4º examen de evaluación continua (22 de marzo de 2013)
-- ---------------------------------------------------------------------

import Data.List
```

```haskell
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Consideremos un número n y sumemos reiteradamente sus
-- cifras hasta un número de una única cifra. Por ejemplo,
--    477 -> 18 -> 9
--    478 -> 19 -> 10 -> 1
-- El número de pasos se llama la persistencia aditiva de n y el último
-- número su raíz digital. Por ejemplo,
--    la persistencia aditiva de 477 es 2 y su raíz digital es 9;
--    la persistencia aditiva de 478 es 3 y su raíz digital es 1.
--
-- Definir la función
--    persistenciaAditiva :: Integer -> Int
-- tal que (persistenciaAditiva n) es el número de veces que hay que
-- reiterar el proceso anterior hasta llegar a un número de una
-- cifra. Por ejemplo,
--    persistenciaAditiva 477  ==  2
--    persistenciaAditiva 478  ==  3
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

persistenciaAditiva :: Integer -> Int
persistenciaAditiva n = length (listaSumas n) -1

-- (listaSumas n) es la lista de las sumas de las cifras de los números
-- desde n hasta su raíz digital. Por ejemplo,
--    listaSumas 477  ==  [477,18,9]
--    listaSumas 478  ==  [478,19,10,1]
listaSumas :: Integer -> [Integer]
listaSumas n | n < 10    = [n]
             | otherwise = n: listaSumas (sumaCifras n)

-- (sumaCifras) es la suma de las cifras de n. Por ejemplo,
--    sumaCifras 477  ==  18
sumaCifras :: Integer -> Integer
sumaCifras = sum . cifras
```

```haskell
-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 477  ==  [4,7,7]
cifras:: Integer -> [Integer]
cifras n = [read [x] | x <- show n]

-- 2ª definición
-- =============

persistenciaAditiva2 :: Integer -> Int
persistenciaAditiva2 n
    | n < 10    = 0
    | otherwise = 1 + persistenciaAditiva2 (sumaCifras n)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    raizDigital :: Integer -> Integer
-- tal que (raizDigital n) es la raíz digital de n. Por ejemplo,
--    raizDigital 477  ==  9
--    raizDigital 478  ==  1
-- ---------------------------------------------------------------------

-- 1ª definición:
raizDigital :: Integer -> Integer
raizDigital n = last (listaSumas n)

-- 2ª definición:
raizDigital2 :: Integer -> Integer
raizDigital2 n
    | n < 10    = n
    | otherwise = raizDigital2 (sumaCifras n)

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar experimentalmente que si n/=0 es múltiplo de
-- 9, entonces la raíz digital n es 9; y en los demás casos, es el resto
-- de la división de n entre 9.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizDigital :: Integer -> Property
prop_raizDigital n =
```

```
    n > 0 ==>
    if n `rem` 9 == 0 then raizDigital n == 9
                      else raizDigital n == rem n 9

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Basándose en estas propiedades, dar una nueva
-- definición de raizDigital.
-- ---------------------------------------------------------------------

raizDigital3 :: Integer -> Integer
raizDigital3 n | r /= 0    = r
               | otherwise = 9
               where r = n `rem` 9

-- Puede definirse sin condicionales:
raizDigital3' :: Integer -> Integer
raizDigital3' n = 1 + (n-1) `rem` 9

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Comprobar con QuickCheck que las definiciones de raíz
-- digital son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_equivalencia_raizDigital :: Integer -> Property
prop_equivalencia_raizDigital n =
    n > 0 ==>
    raizDigital2  n == x &&
    raizDigital3  n == x &&
    raizDigital3' n == x
    where x = raizDigital n

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Con las definiciones anteriores, calcular la raíz
-- digital del número 987698764521^23456 y comparar su eficiencia.
-- ---------------------------------------------------------------------

-- ghci> :set +s
-- ghci> raizDigital (987698764521^23456)
-- 9
-- (6.55 secs, 852846660 bytes)
-- ghci> raizDigital2 (987698764521^23456)
-- 9
-- (6.42 secs, 852934412 bytes)
-- ghci> raizDigital3 (987698764521^23456)
-- 9
-- (0.10 secs, 1721860 bytes)
-- ghci> raizDigital3' (987698764521^23456)
-- 9
-- (0.10 secs, 1629752 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    interVerifican :: Eq a => (b -> Bool) -> (b -> a) -> [[b]] -> [a]
-- tal que (interVerifican p f xss) calcula la intersección de las
-- imágenes por f de los elementos de las listas de xss que verifican p.
-- Por ejemplo,
--    interVerifican even (\x -> x+1) [[1,3,4,2], [4,8], [9,4]] == [5]
--    interVerifican even (\x -> x+1) [[1,3,4,2], [4,8], [9]]   == []
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
interVerifican :: Eq a => (b -> Bool) -> (b -> a) -> [[b]] -> [a]
interVerifican p f xss = interseccion [[f x | x <- xs, p x] | xs <- xss]

-- (interseccion xss) es la intersección de los elementos de xss. Por
-- ejemplo,
--    interseccion [[1,3,4,2], [4,8,3], [9,3,4]]  ==  [3,4]
interseccion :: Eq a => [[a]] -> [a]
interseccion []   = []
interseccion (xs:xss) = [x | x<-xs, and [x `elem` ys| ys <-xss]]
```

```haskell
-- 2ª definición (con map y filter):
interVerifican2 :: Eq a => (b -> Bool) -> (b -> a) -> [[b]] -> [a]
interVerifican2 p f = interseccion . map (map f . filter p)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. La sucesión autocontadora
--    1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, ...
-- está formada por 1 copia de1 1, 2 copias del 2, 3 copias del 3, ...
--
-- Definir la constante
--    autocopiadora :: [Integer]
-- tal que autocopiadora es lista de los términos de la sucesión
-- anterior. Por ejemplo,
--    take 20 autocopiadora == [1,2,2,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6,6]
-- ---------------------------------------------------------------------

autocopiadora :: [Integer]
autocopiadora = concat [genericReplicate n n | n <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    terminoAutocopiadora :: Integer -> Integer
-- tal que (terminoAutocopiadora n) es el lugar que ocupa en la sucesión
-- la primera ocurrencia de n. Por ejemplo,
--    terminoAutocopiadora 4  == 6
--    terminoAutocopiadora 5  == 10
--    terminoAutocopiadora 10 == 45
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
terminoAutocopiadora :: Integer -> Integer
terminoAutocopiadora x =
    head [n | n <- [1..], genericIndex autocopiadora n == x]

-- 2ª definición (con takeWhile):
terminoAutocopiadora2 :: Integer -> Integer
terminoAutocopiadora2 x = genericLength (takeWhile (/=x) autocopiadora)

-- 3ª definición (por recursión)
terminoAutocopiadora3 :: Integer -> Integer
```

```haskell
terminoAutocopiadora3 x = aux x autocopiadora 0
  where aux x (y:ys) k | x == y    = k
                       | otherwise = aux x ys (k+1)

-- 4ª definición (sumando):
terminoAutocopiadora4 :: Integer -> Integer
terminoAutocopiadora4 x = sum [1..x-1]

-- 5ª definición (explícitamente):
terminoAutocopiadora5 :: Integer -> Integer
terminoAutocopiadora5 x = (x-1)*x `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Calcular el lugar que ocupa en la sucesión la
-- primera ocurrencia de 2013. Y también el de 20132013.
-- ---------------------------------------------------------------------

-- El cálculo es
--    terminoAutocopiadora5 2013      ==  2025078
--    terminoAutocopiadora5 20132013  ==  202648963650078

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los árboles binarios definidos por
--    data Arbol = H Int
--               | N Arbol Int Arbol
--               deriving (Show, Eq)
-- Por ejemplo, los árboles siguientes
--         5             8            5          5
--        / \           / \          / \        / \
--       /   \         /   \        /   \      /   \
--      9     7       9     3      9     2    4     7
--     / \   / \     / \   / \    / \              / \
--    1   4 6   8   1   4 6   2  1   4            6   2
-- se representan por
--    arbol1, arbol2, arbol3, arbol4 :: Arbol
--    arbol1 = N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8))
--    arbol2 = N (N (H 1) 9 (H 4)) 8 (N (H 6) 3 (H 2))
--    arbol3 = N (N (H 1) 9 (H 4)) 5 (H 2)
--    arbol4 = N (H 4) 5 (N (H 6) 7 (H 2))
--
```

```haskell
-- Observad que los árboles arbol1 y arbol2 tiene la misma estructura,
-- pero los árboles arbol1 y arbol3 o arbol1 y arbol4 no la tienen
--
-- Definir la función
--    igualEstructura :: Arbol -> Arbol -> Bool
-- tal que (igualEstructura a1 a1) se verifica si los árboles a1 y a2
-- tienen la misma estructura. Por ejemplo,
--    igualEstructura arbol1 arbol2 == True
--    igualEstructura arbol1 arbol3 == False
--    igualEstructura arbol1 arbol4 == False
-- ------------------------------------------------------------------

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

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

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

3.5. Examen 5 (10 de mayo de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 5º examen de evaluación continua (10 de mayo de 2013)
-- ------------------------------------------------------------------

import Data.Array
import Data.Ratio
import PolOperaciones

-- ------------------------------------------------------------------
-- Ejercicio 1 (370 del Proyecto Euler). Un triángulo geométrico es un
```

```haskell
-- triángulo de lados enteros, representados por la terna (a,b,c) tal
-- que a ≤ b ≤ c y están en progresión geométrica, es decir,
-- b^2 = a*c. Por ejemplo, un triángulo de lados a = 144, b = 156 y
-- c = 169.
--
-- Definir la función
--    numeroTG :: Integer -> Int
-- tal que (numeroTG n) es el número de triángulos geométricos de
-- perímetro menor o igual que n. Por ejemplo
--     numeroTG 10  == 4
--     numeroTG 100 == 83
--     numeroTG 200 == 189
-- ---------------------------------------------------------------------

-- 1ª definición:
numeroTG :: Integer -> Int
numeroTG n =
    length [(a,b,c) | c <- [1..n],
                      b <- [1..c],
                      a <- [1..b],
                      a+b+c <= n,
                      b^2 == a*c]

-- 2ª definición:
numeroTG2 :: Integer -> Int
numeroTG2 n =
    length [(a,b,c) | c <- [1..n],
                      b <- [1..c],
                      b^2 `rem` c == 0,
                      let a = b^2 `div` c,
                      a+b+c <= n]

-- 3ª definición:
numeroTG3 :: Integer -> Int
numeroTG3 n =
    length [(b^2 `div` c,b,c) | c <- [1..n],
                                b <- [1..c],
                                b^2 `rem` c == 0,
                                (b^2 `div` c)+b+c <= n]
```

```
-- Comparación de eficiencia:
--    ghci> numeroTG 200
--    189
--    (2.32 secs, 254235740 bytes)
--    ghci> numeroTG2 200
--    189
--    (0.06 secs, 5788844 bytes)
--    ghci> numeroTG3 200
--    189
--    (0.06 secs, 6315900 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Cálculo numérico) 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
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3 Definir la función
--    numeroAPol :: Int -> Polinomio Int
-- tal que (numeroAPol n) es el polinomio cuyas raices son las
-- cifras de n. Por ejemplo,
--   numeroAPol 5703 == x^4 + -15*x^3 + 71*x^2 + -105*x
-- ---------------------------------------------------------------------

numeroAPol :: Int -> Polinomio Int
numeroAPol n = numerosAPol (cifras n)

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 5703  ==  [5,7,0,3]
cifras :: Int -> [Int]
cifras n = [read [c] | c <- show n]

-- (numeroAPol xs) es el polinomio cuyas raices son los elementos de
-- xs. Por ejemplo,
--    numerosAPol [5,7,0,3]  ==  x^4 + -15*x^3 + 71*x^2 + -105*x
numerosAPol :: [Int] -> Polinomio Int
numerosAPol [] = polUnidad
numerosAPol (x:xs) =
    multPol (consPol 1 1 (consPol 0 (-x) polCero))
            (numerosAPol xs)

-- La función anterior se puede definir mediante plegado
numerosAPol2 :: [Int] -> Polinomio Int
numerosAPol2 =
     foldr (\ x -> multPol (consPol 1 1 (consPol 0 (-x) polCero)))
           polUnidad
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Consideremos el tipo de los vectores y de las matrices
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
-- y los ejemplos siguientes:
--    p1 :: (Fractional a, Eq a) => Matriz a
--    p1 = listArray ((1,1),(3,3)) [1,0,0,0,0,1,0,1,0]
--
--    v1,v2 :: (Fractional a, Eq a) => Vector a
--    v1 = listArray (1,3) [0,-1,1]
--    v2 = listArray (1,3) [1,2,1]
--
-- Definir la función
--    esAutovector :: (Fractional a, Eq a) =>
--                    Vector a -> Matriz a -> Bool
-- tal que (esAutovector v p) compruebe si v es un autovector de p
-- (es decir, el producto de v por p es un vector proporcional a
-- v). Por ejemplo,
--    esAutovector v2 p1 == False
--    esAutovector v1 p1 == True
-- ---------------------------------------------------------------------

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

p1:: (Fractional a, Eq a) => Matriz a
p1 = listArray ((1,1),(3,3)) [1,0,0,0,0,1,0,1,0]

v1,v2:: (Fractional a, Eq a) => Vector a
v1 = listArray (1,3) [0,-1,1]
v2 = listArray (1,3) [1,2,1]

esAutovector :: (Fractional a, Eq a) => Vector a -> Matriz a -> Bool
esAutovector v p = proporcional (producto p v) v

-- (producto p v) es el producto de la matriz p por el vector v. Por
-- ejemplo,
--    producto p1 v1  = array (1,3) [(1,0.0),(2,1.0),(3,-1.0)]
--    producto p1 v2  = array (1,3) [(1,1.0),(2,1.0),(3,2.0)]
```

```haskell
producto :: (Fractional a, Eq a) => Matriz a -> Vector a -> Vector a
producto p v =
    array (1,n) [(i, sum [p!(i,j)*v!j | j <- [1..n]]) | i <- [1..m]]
    where (_,n)     = bounds v
          (_,(m,_)) = bounds p

-- (proporcional v1 v2) se verifica si los vectores v1 y v2 son
-- proporcionales. Por ejemplo,
--    proporcional v1 v1                            = True
--    proporcional v1 v2                            = False
--    proporcional v1 (listArray (1,3) [0,-5,5])    = True
--    proporcional v1 (listArray (1,3) [0,-5,4])    = False
--    proporcional (listArray (1,3) [0,-5,5]) v1    = True
--    proporcional v1 (listArray (1,3) [0,0,0])     = True
--    proporcional (listArray (1,3) [0,0,0]) v1     = False
proporcional :: (Fractional a, Eq a) => Vector a -> Vector a -> Bool
proporcional v1 v2
    | esCero v1 = esCero v2
    | otherwise = and [v2!i == k*(v1!i) | i <- [1..n]]
    where (_,n) = bounds v1
          j     = minimum [i | i <- [1..n], v1!i /= 0]
          k     = (v2!j) / (v1!j)

-- (esCero v) se verifica si v es el vector 0.
esCero :: (Fractional a, Eq a) => Vector a -> Bool
esCero v = null [x | x <- elems v, x /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    autovalorAsociado :: (Fractional a, Eq a) =>
--                         Matriz a -> Vector a -> Maybe a
-- tal que si v es un autovector de p, calcule el autovalor asociado.
-- Por ejemplo,
--    autovalorAsociado p1 v1 == Just (-1.0)
--    autovalorAsociado p1 v2 == Nothing
-- ---------------------------------------------------------------------

autovalorAsociado :: (Fractional a, Eq a) =>
                     Matriz a -> Vector a -> Maybe a
autovalorAsociado p v
```

```
    | esAutovector v p = Just (producto p v ! j / v ! j)
    | otherwise        = Nothing
    where (_,n) = bounds v
          j     = minimum [i | i <- [1..n], v!i /= 0]
```

3.6. Examen 6 (13 de junio de 2013)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 6º examen de evaluación continua (13 de junio de 2013)
-- ---------------------------------------------------------------------

import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número es creciente si cada una de sus cifras es
-- mayor o igual que su anterior.
--
-- Definir la función
--    numerosCrecientes :: [Integer] -> [Integer]
-- tal que (numerosCrecientes xs) es la lista de los números crecientes
-- de xs. Por ejemplo,
--    ghci> numerosCrecientes [21..50]
--    [22,23,24,25,26,27,28,29,33,34,35,36,37,38,39,44,45,46,47,48,49]
-- Usando la definición de numerosCrecientes calcular la cantidad de
-- números crecientes de 3 cifras.
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
numerosCrecientes :: [Integer] -> [Integer]
numerosCrecientes xs = [n | n <- xs, esCreciente (cifras n)]

-- (esCreciente xs) se verifica si xs es una sucesión cerciente. Por
-- ejemplo,
--    esCreciente [3,5,5,12]  ==  True
--    esCreciente [3,5,4,12]  ==  False
esCreciente :: Ord a => [a] -> Bool
esCreciente (x:y:zs) = x <= y && esCreciente (y:zs)
esCreciente _        = True

-- (cifras x) es la lista de las cifras del número x. Por ejemplo,
```

```haskell
--    cifras 325  ==  [3,2,5]
cifras :: Integer -> [Integer]
cifras x = [read [d] | d <- show x]

-- El cálculo es
--    ghci> length (numerosCrecientes [100..999])
--    165

-- 2ª definición (por filtrado):
numerosCrecientes2 :: [Integer] -> [Integer]
numerosCrecientes2 = filter (\n -> esCreciente (cifras n))

-- 3ª definición (por recursión):
numerosCrecientes3 :: [Integer] -> [Integer]
numerosCrecientes3 [] = []
numerosCrecientes3 (n:ns)
  | esCreciente (cifras n) = n : numerosCrecientes3 ns
  | otherwise              = numerosCrecientes3 ns

-- 4ª definición (por plegado):
numerosCrecientes4 :: [Integer] -> [Integer]
numerosCrecientes4 = foldr f []
  where f n ns | esCreciente (cifras n) = n : ns
               | otherwise              = ns

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sublistasIguales :: Eq a => [a] -> [[a]]
-- tal que (sublistasIguales xs) es la listas de elementos consecutivos
-- de xs que son iguales. Por ejemplo,
--    ghci> sublistasIguales [1,5,5,10,7,7,7,2,3,7]
--    [[1],[5,5],[10],[7,7,7],[2],[3],[7]]
-- ---------------------------------------------------------------------

-- 1ª definición:
sublistasIguales :: Eq a => [a] -> [[a]]
sublistasIguales [] = []
sublistasIguales (x:xs) =
  (x : takeWhile (==x) xs) : sublistasIguales (dropWhile (==x) xs)
```

```haskell
-- 2ª definición:
sublistasIguales2 :: Eq a => [a] -> [[a]]
sublistasIguales2 []     = []
sublistasIguales2 [x]    = [[x]]
sublistasIguales2 (x:y:zs)
  | x == u    = (x:u:us):vss
  | otherwise = [x]:((u:us):vss)
  where ((u:us):vss) = sublistasIguales2 (y:zs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios se pueden representar con el de
-- dato algebraico
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo, los árboles
--         9                9
--        / \              / \
--       /   \            /   \
--      8     6          8     6
--     / \   / \        / \   / \
--    3   2 4   5      3   2 4   7
-- se pueden representar por
--    ej1, ej2:: Arbol Int
--    ej1 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--    ej2 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 7 H H))
-- Un árbol binario ordenado es un árbol binario (ABO) en el que los
-- valores de cada nodo es mayor o igual que los valores de sus
-- hijos. Por ejemplo, ej1 es un ABO, pero ej2 no lo es.
--
-- Definir la función esABO
--    esABO :: Ord t => Arbol t -> Bool
-- tal que (esABO a) se verifica si a es un árbol binario ordenado. Por
-- ejemplo.
--    esABO ej1 == True
--    esABO ej2 == False
-- ---------------------------------------------------------------------

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

```
             deriving Show

ej1, ej2 :: Arbol Int
ej1 = N 9 (N 8 (N 3 H H) (N 2 H H))
          (N 6 (N 4 H H) (N 5 H H))

ej2 = N 9 (N 8 (N 3 H H) (N 2 H H))
          (N 6 (N 4 H H) (N 7 H H))

-- 1ª definición
esABO :: Ord a => Arbol a -> Bool
esABO H                       = True
esABO (N x H H)               = True
esABO (N x m1@(N x1 a1 b1) H) = x >= x1 && esABO m1
esABO (N x H m2@(N x2 a2 b2)) = x >= x2 && esABO m2
esABO (N x m1@(N x1 a1 b1) m2@(N x2 a2 b2)) =
      x >= x1 && esABO m1 && x >= x2 && esABO m2

-- 2ª definición
esABO2 :: Ord a => Arbol a -> Bool
esABO2 H         = True
esABO2 (N x i d) = mayor x i && mayor x d && esABO2 i && esABO2 d
       where  mayor x H         = True
              mayor x (N y _ _) = x >= y

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    paresEspecialesDePrimos :: Integer -> [(Integer,Integer)]
-- tal que (paresEspecialesDePrimos n) es la lista de los pares de
-- primos (p,q) tales que p < q y q-p es divisible por n. Por ejemplo,
--    ghci> take 9 (paresEspecialesDePrimos 2)
--    [(3,5),(3,7),(5,7),(3,11),(5,11),(7,11),(3,13),(5,13),(7,13)]
--    ghci> take 9 (paresEspecialesDePrimos 3)
--    [(2,5),(2,11),(5,11),(7,13),(2,17),(5,17),(11,17),(7,19),(13,19)]
-- ---------------------------------------------------------------------

paresEspecialesDePrimos :: Integer -> [(Integer,Integer)]
paresEspecialesDePrimos n =
  [(p,q) | (p,q) <- paresPrimos, rem (q-p) n == 0]
```

```haskell
-- paresPrimos es la lista de los pares de primos (p,q) tales que p < q.
-- Por ejemplo,
--    ghci> take 9 paresPrimos
--    [(2,3),(2,5),(3,5),(2,7),(3,7),(5,7),(2,11),(3,11),(5,11)]
paresPrimos :: [(Integer,Integer)]
paresPrimos = [(p,q) | q <- primos, p <- takeWhile (<q) primos]

-- primos es la lista de primos. Por ejemplo,
--    take 9 primos  ==  [2,3,5,7,11,13,17,19,23]
primos :: [Integer]
primos = 2 : [n | n <- [3,5..], esPrimo n]

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Las matrices enteras se pueden representar mediante
-- tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
--
-- Definir la función
--    ampliaColumnas :: Matriz -> Matriz -> Matriz
-- tal que (ampliaColumnas p q) es la matriz construida añadiendo las
-- columnas de la matriz q a continuación de las de p (se supone que
-- tienen el mismo número de filas). Por ejemplo, si p y q representa
-- las dos primeras matrices, entonces (ampliaColumnas p q) es la
-- tercera
--    |0 1|    |4 5 6|    |0 1 4 5 6|
--    |2 3|    |7 8 9|    |2 3 7 8 9|
-- En Haskell,
--    ghci> :{
--    *Main| ampliaColumnas (listArray ((1,1),(2,2)) [0..3])
--    *Main|                (listArray ((1,1),(2,3)) [4..9])
--    *Main| :}
--    array ((1,1),(2,5))
--          [((1,1),0),((1,2),1),((1,3),4),((1,4),5),((1,5),6),
--           ((2,1),2),((2,2),3),((2,3),7),((2,4),8),((2,5),9)]
```

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

type Matriz = Array (Int,Int) Int

ampliaColumnas :: Matriz -> Matriz -> Matriz
ampliaColumnas p1 p2 =
  array ((1,1),(m,n1+n2)) [((i,j), f i j) | i <- [1..m], j <- [1..n1+n2]]
    where ((_,_),(m,n1)) = bounds p1
          ((_,_),(_,n2)) = bounds p2
          f i j | j <= n1   = p1!(i,j)
                | otherwise = p2!(i,j-n1)
```

3.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 1 (ver página 26).

3.8. Examen 8 (13 de septiembre de 2013)

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

3.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 1 (ver página 38).

4

Exámenes del grupo 4

Andrés Cordón e Ignacio Pérez

4.1. Examen 1 (12 de noviembre de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (12 de noviembre de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Dada una ecuación de tercer grado de la forma
--    x^3 + ax^2 + bx + c = 0,
-- donde a, b y c son números reales, se define el discriminante de la
-- ecuación como
--    d = 4p^3+ 27q^2,
-- donde p = b - a^3/3 y q = 2a^3/27 - ab/3 + c.
--
-- Definir la función
--    disc :: Float -> Float -> Float -> Float
-- tal que (disc a b c) es el discriminante de la ecuación
-- x^3 + ax^2 + bx + c = 0. Por ejemplo,
--    disc 1 (-11) (-9) == -5075.9995
-- ---------------------------------------------------------------------

disc :: Float -> Float -> Float -> Float
disc a b c = 4*p^3 + 27*q^2
    where p = b - (a^3)/3
          q = (2*a^3)/27 - (a*b)/3 + c
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.2. El signo del discriminante permite determinar el
-- número de raíces reales de la ecuación:
--    d > 0 : 1 solución,
--    d = 0 : 2 soluciones y
--    d < 0 : 3 soluciones
--
-- Definir la función
--    numSol :: Float -> Float > Float -> Int
-- tal que (numSol a b c) es el número de raíces reales de la ecuación
-- x^3 + ax^2 + bx + c = 0. Por ejemplo,
--    numSol 1 (-11) (-9) == 3
-- ---------------------------------------------------------------------

numSol :: Float -> Float -> Float -> Int
numSol a b c
    | d > 0     = 1
    | d == 0    = 2
    | otherwise = 3
    where d = disc a b c

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    numDiv :: Int -> Int
-- tal que (numDiv x) es el número de divisores del número natural
-- x. Por ejemplo,
--    numDiv 11 == 2
--    numDiv 12 == 6
-- ---------------------------------------------------------------------

numDiv :: Int -> Int
numDiv x = length [n | n <- [1..x], rem x n == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    entre :: Int -> Int -> Int -> [Int]
-- tal que (entre a b c) es la lista de los naturales entre a y b con,
-- al menos, c divisores. Por ejemplo,
--    entre 11 16 5 == [12, 16]
-- ---------------------------------------------------------------------
```

```
entre :: Int -> Int -> Int -> [Int]
entre a b c = [x | x <- [a..b], numDiv x >= c]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    conPos :: [a] -> [(a,Int)]
-- tal que (conPos xs) es la lista obtenida a partir de xs especificando
-- las posiciones de sus elementos. Por ejemplo,
--    conPos [1,5,0,7] == [(1,0),(5,1),(0,2),(7,3)]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    pares :: String -> String
-- tal que (pares cs) es la cadena formada por los caracteres en
-- posición par de cs. Por ejemplo,
--    pares "el cielo sobre berlin" == "e il or eln"
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir el predicado
--    comparaFecha :: (Int,String,Int) -> (Int,String,Int) -> Bool
-- que recibe dos fechas en el formato (dd,"mes",aaaa) y se verifica si
-- la primera fecha es anterior a la segunda. Por ejemplo:
--    comparaFecha (12, "noviembre", 2012) (01, "enero", 2015) == True
--    comparaFecha (12, "noviembre", 2012) (01, "enero", 2012) == False
-- ---------------------------------------------------------------------

comparaFecha :: (Int,String,Int) -> (Int,String,Int) -> Bool
comparaFecha (d1,m1,a1) (d2,m2,a2) =
    (a1,mes m1,d1) < (a2,mes m2,d2)
    where mes "enero"      = 1
          mes "febrero"    = 2
```

```
mes "marzo"      = 3
mes "abril"      = 4
mes "mayo"       = 5
mes "junio"      = 6
mes "julio"      = 7
mes "agosto"     = 8
mes "septiembre" = 9
mes "octubre"    = 10
mes "noviembre"  = 11
mes "diciembre"  = 12
```

4.2. Examen 2 (17 de diciembre de 2012)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (17 de diciembre de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, usando funciones de orden superior (map,
-- filter, ... ), la función
--    sumaCuad :: [Int] -> (Int,Int)
-- tal que (sumaCuad xs) es el par formado porla suma de los cuadrados
-- de los elementos pares de xs, por una parte, y la suma de los
-- cuadrados de los elementos impares, por otra. Por ejemplo,
--    sumaCuad [1,3,2,4,5] == (20,35)
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
sumaCuad1 :: [Int] -> (Int,Int)
sumaCuad1 xs =
    (sum [x^2 | x <- xs, even x],sum [x^2 | x <- xs, odd x])

-- 2ª definición (con filter):
sumaCuad2 :: [Int] -> (Int,Int)
sumaCuad2 xs =
    (sum [x^2 | x <- filter even xs],sum [x^2 | x <- filter odd xs])

-- 3ª definición (con map yfilter):
sumaCuad3 :: [Int] -> (Int,Int)
sumaCuad3 xs =
```

```
    (sum (map (^2) (filter even xs)),sum (map (^2) (filter odd xs)))

-- 4ª definición (por recursión):
sumaCuad4 :: [Int] -> (Int,Int)
sumaCuad4 xs = aux xs (0,0)
    where aux [] (a,b) = (a,b)
          aux (x:xs) (a,b) | even x    = aux xs (x^2+a,b)
                           | otherwise = aux xs (a,x^2+b)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por recursión, el predicado
--    alMenosR :: Int -> [Int] -> Bool
-- tal que (alMenosR k xs) se verifica si xs contiene, al menos, k
-- números primos. Por ejemplo,
--    alMenosR 1 [1,3,7,10,14] == True
--    alMenosR 3 [1,3,7,10,14] == False
-- ---------------------------------------------------------------------

alMenosR :: Int -> [Int] -> Bool
alMenosR 0 _  = True
alMenosR _ [] = False
alMenosR k (x:xs) | esPrimo x = alMenosR (k-1) xs
                  | otherwise = alMenosR k xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por comprensión, el predicado
--    alMenosC :: Int -> [Int] -> Bool
-- tal que (alMenosC k xs) se verifica si xs contiene, al menos, k
-- números primos. Por ejemplo,
--    alMenosC 1 [1,3,7,10,14] == True
--    alMenosC 3 [1,3,7,10,14] == False
-- ---------------------------------------------------------------------
```

```haskell
alMenosC :: Int -> [Int] -> Bool
alMenosC k xs = length [x | x <- xs, esPrimo x] >= k

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la La función
--    alternos :: (a -> b) -> (a -> b) -> [a] -> [b]
-- tal que (alternos f g xs) es la lista obtenida aplicando
-- alternativamente las funciones f y g a los elementos de la lista
-- xs. Por ejemplo,
--    ghci> alternos (+1) (*3) [1,2,3,4,5]
--    [2,6,4,12,6]
--    ghci> alternos (take 2) reverse [”todo”,”para”,”nada”]
--    [”to”,”arap”,”na”]
-- ---------------------------------------------------------------------

alternos :: (a -> b) -> (a -> b) -> [a] -> [b]
alternos _ _ [] = []
alternos f g (x:xs) = f x : alternos g f xs
```

4.3. Examen 3 (6 de febrero de 2013)

El examen es común con el del grupo 1 (ver página 11).

4.4. Examen 4 (18 de marzo de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 4º examen de evaluación continua (18 de marzo de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. 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]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, usando map y filter, la función
--    filtraAplicaMF :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaMF f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaMF (4+) (< 3) [1..7] == [5,6]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por recursión, la función
--    filtraAplicaR :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaR f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaR (4+) (< 3) [1..7] == [5,6]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir, por plegado, la función
--    filtraAplicaP :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplicaP f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplicaP (4+) (< 3) [1..7] == [5,6]
-- ---------------------------------------------------------------------

filtraAplicaP :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplicaP f p = foldr g []
    where g x y | p x       = f x : y
                | otherwise = y

-- Se puede usar lambda en lugar de la función auxiliar
filtraAplicaP' :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

```haskell
filtraAplicaP' f p = foldr (\x y -> if p x then f x : y else y) []

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios se pueden representar con el de
-- tipo de dato algebraico
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
-- Por ejemplo, los árboles
--         9                9
--        / \              / \
--       /   \            /   \
--      8     8          4     8
--     / \   / \        / \   / \
--    3   2 4   5      3   2 5   7
-- se pueden representar por
--    ej1, ej2:: Arbol Int
--    ej1 = N 9 (N 8 (H 3) (H 2)) (N 8 (H 4) (H 5))
--    ej2 = N 9 (N 4 (H 3) (H 2)) (N 8 (H 5) (H 7))
--
-- Se considera la definición de tipo de dato:
--
-- Definir el predicado
--    contenido :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (contenido a1 a2) es verdadero si todos los elementos que
-- aparecen en el árbol a1 también aparecen en el árbol a2. Por ejemplo,
--    contenido ej1 ej2  ==  True
--    contenido ej2 ej1  ==  False
-- ---------------------------------------------------------------------

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

ej1, ej2:: Arbol Int
ej1 = N 9 (N 8 (H 3) (H 2)) (N 8 (H 4) (H 5))
ej2 = N 9 (N 4 (H 3) (H 2)) (N 8 (H 5) (H 7))

contenido :: Eq a => Arbol a -> Arbol a -> Bool
contenido (H x) a     = pertenece x a
contenido (N x i d) a = pertenece x a && contenido i a && contenido d a
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    esCubo :: Int -> Bool
-- tal que (esCubo x) se verifica si el entero x es un cubo
-- perfecto. Por ejemplo,
--    esCubo 27  ==  True
--    esCubo 50  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición:
esCubo :: Int -> Bool
esCubo x = y^3 == x
    where y = ceiling ((fromIntegral x)**(1/3))

-- 2ª definición:
esCubo2 :: Int -> Bool
esCubo2 x = elem x (takeWhile (<=x) [i^3 | i <- [1..]])

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la lista (infinita)
--    soluciones :: [Int]
-- cuyos elementos son los números naturales que pueden escribirse como
-- suma de dos cubos perfectos, al menos, de dos maneras distintas. Por
-- ejemplo,
--    take 3 soluciones  ==  [1729,4104,13832]
-- ---------------------------------------------------------------------

soluciones :: [Int]
soluciones = [x | x <- [1..], length (sumas x) >= 2]

-- (sumas x) es la lista de pares de cubos cuya suma es x. Por ejemplo,
--    sumas 1729  ==  [(1,1728),(729,1000)]
sumas :: Int -> [(Int,Int)]
sumas x = [(a^3,x-a^3) | a <- [1..cota], a^3 <= x-a^3,  esCubo (x-a^3)]
```

```haskell
    where cota = floor ((fromIntegral x)**(1/3))

-- La definición anterior se pued simplificar:
sumas2 :: Int -> [(Int,Int)]
sumas2 x = [(a^3,x-a^3) | a <- [1..cota],  esCubo (x-a^3)]
    where cota = floor ((fromIntegral x / 2)**(1/3))

-- ---------------------------------------------------------------------
-- Ejercicio 4. Disponemos de una mochila que tiene una capacidad
-- limitada de c kilos. Nos encontramos con una serie de objetos cada
-- uno con un valor v y un peso p. El problema de la mochila consiste en
-- escoger subconjuntos de objetos tal que la suma de sus valores sea
-- máxima y la suma de sus pesos no rebase la capacidad de la mochila.
--
-- Se definen los tipos sinónimos:
--    type Peso a   = [(a,Int)]
--    type Valor a  = [(a,Int)]
-- para asignar a cada objeto, respectivamente, su peso o valor.
--
-- Definir la función:
--    mochila :: Eq a => [a] -> Int -> Peso a -> Valor a -> [[a]]
-- tal que (mochila xs c ps vs) devuelve todos los subconjuntos de xs
-- tal que la suma de sus valores sea máxima y la suma de sus pesos sea
-- menor o igua que cota c. Por ejemplo,
--    ghci> :{
--    *Main| mochila ["linterna", "oro", "bocadillo", "apuntes"] 10
--    *Main|         [("oro",7),("bocadillo",1),("linterna",2),("apuntes",5)]
--    *Main|         [("apuntes",8),("linterna",1),("oro",100),("bocadillo",10)]
--    *Main| :}
-- ---------------------------------------------------------------------

type Peso a  = [(a,Int)]
type Valor a = [(a,Int)]

mochila :: Eq a => [a] -> Int -> Peso a -> Valor a -> [[a]]
mochila xs c ps vs = [ys | ys <- rellenos, pesoTotal ys vs == maximo]
    where rellenos = posibles xs c ps
          maximo   = maximum [pesoTotal ys vs | ys <- rellenos]

-- (posibles xs c ps) es la lista de objetos de xs cuyo peso es menor o
```

```
-- igual que c y sus peso están indicada por ps. Por ejemplo,
--    ghci> posibles [”a”,”b”,”c”] 9 [(”a”,3),(”b”,7),(”c”,2)]
--    [[],[”c”],[”b”],[”b”,”c”],[”a”],[”a”,”c”]]
posibles :: Eq a => [a] -> Int -> Peso a -> [[a]]
posibles xs c ps = [ys | ys <- subconjuntos xs, pesoTotal ys ps <= c]

-- (subconjuntos xs) es la lista de los subconjuntos de xs. Por ejemplo,
--    subconjuntos [2,5,3]  ==  [[],[3],[5],[5,3],[2],[2,3],[2,5],[2,5,3]]
subconjuntos :: [a] -> [[a]]
subconjuntos []     = [[]]
subconjuntos (x:xs) = subconjuntos xs ++ [x:ys | ys <- subconjuntos xs]

-- (pesoTotal xs ps) es el peso de todos los objetos de xs tales que los
-- pesos de cada uno están indicado por ps. Por ejemplo,
--    pesoTotal [”a”,”b”,”c”] [(”a”,3),(”b”,7),(”c”,2)]  ==  12
pesoTotal :: Eq a => [a] -> Peso a -> Int
pesoTotal xs ps = sum [peso x ps | x <- xs]

-- (peso x ps) es el peso de x en la lista de pesos ps. Por ejemplo,
--    peso ”b” [(”a”,3),(”b”,7),(”c”,2)]  ==  7
peso :: Eq a => a -> [(a,b)]  -> b
peso x ps = head [b | (a,b) <- ps, a ==x]
```

4.5. Examen 5 (6 de mayo de 2013)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 5º examen de evaluación continua (6 de mayo de 2013)
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando la primera
-- ocurrencia del elemento x en la lista xs. Por ejemplo,
--    borra 'a' ”salamanca” == ”slamanca”
-- ---------------------------------------------------------------------

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

```haskell
borra _ [] = []
borra x (y:ys) | x == y    = ys
               | otherwise = y : borra x ys

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    borraTodos :: Eq a => a -> [a] -> [a]
-- tal que (borraTodos x xs) es la lista obtenida borrando todas las
-- ocurrencias de x en la lista xs. Por ejemplo,
--    borraTodos 'a' "salamanca" == "slmnc"
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por plegado, la función
--    borraTodosP :: Eq a => a -> [a] -> [a]
-- tal que (borraTodosP x xs) es la lista obtenida borrando todas las
-- ocurrencias de x en la lista xs. Por ejemplo,
--    borraTodosP 'a' "salamanca" == "slmnc"
-- ---------------------------------------------------------------------

borraTodosP :: Eq a => a -> [a] -> [a]
borraTodosP x = foldr f []
    where f y ys | x == y    = ys
                 | otherwise = y:ys

-- usando funciones anónimas la definición es
borraTodosP' :: Eq a => a -> [a] -> [a]
borraTodosP' x = foldr (\ y z -> if x == y then z else (y:z)) []

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir, por recursión, la función
--    borraN :: Eq a => Int -> a -> [a] -> [a]
-- tal que (borraN n x xs) es la lista obtenida borrando las n primeras
-- ocurrencias de x en la lista xs. Por ejemplo,
--    borraN 3 'a' "salamanca" == "slmnca"
```

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

borraN :: Eq a => Int -> a -> [a] -> [a]
borraN _ _ [] = []
borraN 0 _ xs = xs
borraN n x (y:ys) | x == y    = borraN (n-1) x ys
                  | otherwise = y : borraN n x ys

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Un número entero positivo x se dirá especial si puede
-- reconstruirse a partir de las cifras de sus factores primos; es decir
-- si el conjunto de sus cifras es igual que la unión de las cifras de
-- sus factores primos. Por ejemplo, 11913 es especial porque sus cifras
-- son [1,1,1,3,9] y sus factores primos son: 3, 11 y 19.
--
-- Definir la función
--    esEspecial :: Int -> Bool
-- tal que (esEspecial x) se verifica si x es especial. Por ejemplo,
--    ???
-- Calcular el menor entero positivo especial que no sea un número
-- primo.
-- ---------------------------------------------------------------------

esEspecial :: Int -> Bool
esEspecial x =
    sort (cifras x) == sort (concat [cifras n | n <- factoresPrimos x])

-- (cifras x) es la lista de las cifras de x. Por ejemplo,
--    cifras 11913  ==  [1,1,9,1,3]
cifras :: Int -> [Int]
cifras x = [read [i] |  i <- show x]

-- (factoresPrimos x) es la lista de los factores primos de x. Por ejemplo,
--    factoresPrimos 11913  ==  [3,11,19]
factoresPrimos :: Int -> [Int]
factoresPrimos x = filter primo (factores x)

-- (factores x) es la lista de los factores de x. Por ejemplo,
--    ghci> factores 11913
--    [1,3,11,19,33,57,209,361,627,1083,3971,11913]
```

```haskell
factores :: Int -> [Int]
factores x = [i | i <- [1..x], mod x i == 0]

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

-- El cálculo es
--    ghci> head [x | x <- [1..], esEspecial x, not (primo x)]
--    735

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una lista de listas de xss se dirá encadenada si el
-- último elemento de cada lista de xss coincide con el primero de la
-- lista siguiente. Por ejemplo, [[1,2,3],[3,4],[4,7]] está encadenada.
--
-- Definir la función
--    encadenadas :: Eq a => [[a]] -> [[[a]]]
-- tal que (encadenadas xss) es la lista de las permutaciones de xss que
-- son encadenadas. Por ejemplo,
--    ghci> encadenadas ["el","leon","ruge","nicanor"]
--    [["ruge","el","leon","nicanor"],
--     ["leon","nicanor","ruge","el"],
--     ["el","leon","nicanor","ruge"],
--     ["nicanor","ruge","el","leon"]]
-- ---------------------------------------------------------------------

encadenadas :: Eq a => [[a]] -> [[[a]]]
encadenadas xss = filter encadenada (permutations xss)

encadenada :: Eq a => [[a]] -> Bool
encadenada xss = and [last xs == head ys | (xs,ys) <-  zip xss (tail xss)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Representamos los polinomios de una variable mediante un
-- tipo algebraico de datos como en el tema 21 de la asignatura:
--    data Polinomio a = PolCero | ConsPol Int a (Polinomio a)
-- Por ejemplo, el polinomio x^3 + 4x^2 + x - 6  se representa por
```

```haskell
--    ej :: Polinomio Int
--    ej = ConsPol 3 1 (ConsPol 2 4 (ConsPol 1 1 (ConsPol 0 (-6) PolCero)))
--
-- Diremos que un polinomio es propio si su término independiente es no
-- nulo.
--
-- Definir la función
--    raices :: Polinomio Int -> [Int]
-- tal que (raices p) es la lista de todas las raíces enteras del
-- polinomio propio p. Por ejemplo,
--    raices ej == [1,-2,-3]
-- ---------------------------------------------------------------------

data Polinomio a = PolCero | ConsPol Int a (Polinomio a)

ej :: Polinomio Int
ej = ConsPol 3 1 (ConsPol 2 4 (ConsPol 1 1 (ConsPol 0 (-6) PolCero)))

raices :: Polinomio Int -> [Int]
raices p = [z | z <- factoresEnteros (termInd p), valor z p == 0]

-- (termInd p) es el término independiente del polinomio p. Por ejemplo,
--    termInd (ConsPol 3 1 (ConsPol 0 5 PolCero))  ==  5
--    termInd (ConsPol 3 1 (ConsPol 2 5 PolCero))  ==  0
termInd :: Num a => Polinomio a -> a
termInd PolCero = 0
termInd (ConsPol n x p) | n == 0    = x
                        | otherwise = termInd p

-- (valor c p) es el valor del polinomio p en el punto c. Por ejemplo,
--    valor 2 (ConsPol 3 1 (ConsPol 2 5 PolCero))  ==  28
valor :: Num a => a -> Polinomio a  -> a
valor _ PolCero = 0
valor z (ConsPol n x p) = x*z^n + valor z p

-- (factoresEnteros x) es la lista de los factores enteros de x. Por
-- ejemplo,
--    factoresEnteros 12  ==  [-1,1,-2,2,-3,3,-4,4,-6,6,-12,12]
factoresEnteros :: Int -> [Int]
factoresEnteros x = concat [[-z,z] | z <- factores (abs x)]
```

4.6. Examen 6 (13 de junio de 2013)

El examen es común con el del grupo 1 (ver página 60).

4.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 1 (ver página 26).

4.8. Examen 8 (13 de septiembre de 2013)

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

4.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 1 (ver página 38).

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.