Exámenes de "Programación funcional con Haskell" (2009–2014)

José A. Alonso (coord.) y

Gonzalo Aranda, Antonia M. Chávez, Andrés Cordón, María J. Hidalgo, Francisco J. Martín Miguel A. Martínez, Ignacio Pérez, José F. Quesada, Agustín Riscos y Luis Valencia

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 27 de julio de 2013 (Versión de 22 de diciembre de 2013)

Índice general

Introducción **7**

1 Exámenes del curso 2009–10 **9**
1.1 Exámenes del grupo 1 (José A. Alonso y Gonzalo Aranda) 9
1.1.1 Examen 1 (30 de noviembre de 2009) 9
1.1.2 Examen 2 (12 de febrero de 2010) 11
1.1.3 Examen 3 (15 de marzo de 2010) 13
1.1.4 Examen 4 (12 de abril de 2010) 18
1.1.5 Examen 5 (17 de mayo de 2010) 22
1.1.6 Examen 6 (21 de junio de 2010) 24
1.1.7 Examen 7 (5 de julio de 2010) 27
1.1.8 Examen 8 (15 de septiembre de 2010) 29
1.1.9 Examen 9 (17 de diciembre de 2010) 34
1.2 Exámenes del grupo 3 (María J. Hidalgo) 38
1.2.1 Examen 1 (4 de diciembre de 2009) 38
1.2.2 Examen 2 (16 de marzo de 2010) 41
1.2.3 Examen 3 (5 de julio de 2010) 45
1.2.4 Examen 4 (15 de septiembre de 2010) 45
1.2.5 Examen 5 (17 de diciembre de 2010) 45

2 Exámenes del curso 2010–11 **47**
2.1 Exámenes del grupo 3 (María J. Hidalgo) 47
2.1.1 Examen 1 (29 de Octubre de 2010) 47
2.1.2 Examen 2 (26 de Noviembre de 2010) 50
2.1.3 Examen 3 (17 de Diciembre de 2010) 53
2.1.4 Examen 4 (11 de Febrero de 2011) 59
2.1.5 Examen 5 (14 de Marzo de 2011) 59
2.1.6 Examen 6 (15 de abril de 2011) 59
2.1.7 Examen 7 (27 de mayo de 2011) 64
2.1.8 Examen 8 (24 de Junio de 2011) 73
2.1.9 Examen 9 (8 de Julio de 2011) 73
2.1.10 Examen 10 (16 de Septiembre de 2011) 74

2.1.11 Examen 11 (22 de Noviembre de 2011) . . . 74
2.2 Exámenes del grupo 4 (José A. Alonso y Agustín Riscos) . . . 74
2.2.1 Examen 1 (25 de Octubre de 2010) . . . 74
2.2.2 Examen 2 (22 de Noviembre de 2010) . . . 75
2.2.3 Examen 3 (20 de Diciembre de 2010) . . . 77
2.2.4 Examen 4 (11 de Febrero de 2011) . . . 81
2.2.5 Examen 5 (14 de Marzo de 2011) . . . 83
2.2.6 Examen 6 (11 de Abril de 2011) . . . 86
2.2.7 Examen 7 (23 de Mayo de 2011) . . . 90
2.2.8 Examen 8 (24 de Junio de 2011) . . . 94
2.2.9 Examen 9 (8 de Julio de 2011) . . . 100
2.2.10 Examen 10 (16 de Septiembre de 2011) . . . 107
2.2.11 Examen 11 (22 de Noviembre de 2011) . . . 112

3 Exámenes del curso 2011–12 119
3.1 Exámenes del grupo 1 (José A. Alonso y Agustín Riscos) . . . 119
3.1.1 Examen 1 (26 de Octubre de 2011) . . . 119
3.1.2 Examen 2 (30 de Noviembre de 2011) . . . 120
3.1.3 Examen 3 (25 de Enero de 2012) . . . 122
3.1.4 Examen 4 (29 de Febrero de 2012) . . . 126
3.1.5 Examen 5 (21 de Marzo de 2012) . . . 129
3.1.6 Examen 6 (2 de Mayo de 2012) . . . 131
3.1.7 Examen 7 (25 de Junio de 2012) . . . 135
3.1.8 Examen 8 (29 de Junio de 2012) . . . 138
3.1.9 Examen 9 (9 de Septiembre de 2012) . . . 143
3.1.10 Examen 10 (10 de Diciembre de 2012) . . . 148
3.2 Exámenes del grupo 2 (María J. Hidalgo) . . . 151
3.2.1 Examen 1 (27 de Octubre de 2011) . . . 151
3.2.2 Examen 2 (1 de Diciembre de 2011) . . . 153
3.2.3 Examen 3 (26 de Enero de 2012) . . . 157
3.2.4 Examen 4 (1 de Marzo de 2012) . . . 163
3.2.5 Examen 5 (22 de Marzo de 2012) . . . 168
3.2.6 Examen 6 (3 de Mayo de 2012) . . . 171
3.2.7 Examen 7 (24 de Junio de 2012) . . . 177
3.2.8 Examen 8 (29 de Junio de 2012) . . . 184
3.2.9 Examen 9 (9 de Septiembre de 2012) . . . 184
3.2.10 Examen 10 (10 de Diciembre de 2012) . . . 184
3.3 Exámenes del grupo 3 (Antonia M. Chávez) . . . 184
3.3.1 Examen 1 (14 de Noviembre de 2011) . . . 184
3.3.2 Examen 2 (12 de Diciembre de 2011) . . . 186
3.3.3 Examen 7 (29 de Junio de 2012) . . . 190

3.3.4 Examen 8 (09 de Septiembre de 2012) . . . 190
3.3.5 Examen 9 (10 de Diciembre de 2012) . . . 190
3.4 Exámenes del grupo 4 (José F. Quesada) . . . 190
3.4.1 Examen 1 (7 de Noviembre de 2011) . . . 190
3.4.2 Examen 2 (30 de Noviembre de 2011) . . . 193
3.4.3 Examen 3 (16 de Enero de 2012) . . . 197
3.4.4 Examen 4 (7 de Marzo de 2012) . . . 202
3.4.5 Examen 5 (28 de Marzo de 2012) . . . 205
3.4.6 Examen 6 (9 de Mayo de 2012) . . . 211
3.4.7 Examen 7 (11 de Junio de 2012) . . . 215
3.4.8 Examen 8 (29 de Junio de 2012) . . . 220
3.4.9 Examen 9 (9 de Septiembre de 2012) . . . 220
3.4.10 Examen 10 (10 de Diciembre de 2012) . . . 220

4 Exámenes del curso 2012–13 221
4.1 Exámenes del grupo 1 (Antonia M. Chávez) . . . 221
4.1.1 Examen 1 (7 de noviembre de 2012) . . . 221
4.1.2 Examen 2 (19 de diciembre de 2012) . . . 223
4.1.3 Examen 3 (6 de febrero de 2013) . . . 227
4.1.4 Examen 4 (3 de abril de 2013) . . . 227
4.1.5 Examen 5 (15 de mayo de 2013) . . . 233
4.1.6 Examen 6 (13 de junio de 2013) . . . 238
4.1.7 Examen 7 (3 de julio de 2013) . . . 244
4.1.8 Examen 8 (13 de septiembre de 2013) . . . 244
4.1.9 Examen 9 (20 de noviembre de 2013) . . . 244
4.2 Exámenes del grupo 2 (José A. Alonso y Miguel A. Martínez) . . . 244
4.2.1 Examen 1 (8 de noviembre de 2012) . . . 244
4.2.2 Examen 2 (20 de diciembre de 2012) . . . 246
4.2.3 Examen 3 (6 de febrero de 2013) . . . 249
4.2.4 Examen 4 (21 de marzo de 2013) . . . 252
4.2.5 Examen 5 (9 de mayo de 2013) . . . 255
4.2.6 Examen 6 (13 de junio de 2013) . . . 259
4.2.7 Examen 7 (3 de julio de 2013) . . . 264
4.2.8 Examen 8 (13 de septiembre de 2013) . . . 271
4.2.9 Examen 9 (20 de noviembre de 2013) . . . 275
4.3 Exámenes del grupo 3 (María J. Hidalgo) . . . 280
4.3.1 Examen 1 (16 de noviembre de 2012) . . . 280
4.3.2 Examen 2 (21 de diciembre de 2012) . . . 282
4.3.3 Examen 3 (6 de febrero de 2013) . . . 289
4.3.4 Examen 4 (22 de marzo de 2013) . . . 289
4.3.5 Examen 5 (10 de mayo de 2013) . . . 296

4.3.6 Examen 6 (13 de junio de 2013) . . . 302
4.3.7 Examen 7 (3 de julio de 2013) . . . 307
4.3.8 Examen 8 (13 de septiembre de 2013) . . . 307
4.3.9 Examen 9 (20 de noviembre de 2013) . . . 307
4.4 Exámenes del grupo 4 (Andrés Cordón e Ignacio Pérez) . . . 307
4.4.1 Examen 1 (12 de noviembre de 2012) . . . 307
4.4.2 Examen 2 (17 de diciembre de 2012) . . . 310
4.4.3 Examen 3 (6 de febrero de 2013) . . . 312
4.4.4 Examen 4 (18 de marzo de 2013) . . . 312
4.4.5 Examen 5 (6 de mayo de 2013) . . . 317
4.4.6 Examen 6 (13 de junio de 2013) . . . 322
4.4.7 Examen 7 (3 de julio de 2013) . . . 322
4.4.8 Examen 8 (13 de septiembre de 2013) . . . 322
4.4.9 Examen 9 (20 de noviembre de 2013) . . . 322

5 Exámenes del curso 2013–14 **323**
5.1 Exámenes del grupo 1 (María J. Hidalgo) . . . 323
5.1.1 Examen 1 (7 de Noviembre de 2013) . . . 323
5.1.2 Examen 2 (19 de Diciembre de 2013) . . . 325
5.2 Exámenes del grupo 3 (José A. Alonso y Luis Valencia) . . . 328
5.2.1 Examen 1 (5 de Noviembre de 2013) . . . 328
5.2.2 Examen 2 (17 de Diciembre de 2013) . . . 331
5.3 Exámenes del grupo 4 (Francisco J. Martín) . . . 334
5.3.1 Examen 1 (5 de Noviembre de 2013) . . . 334
5.3.2 Examen 2 (16 de Diciembre de 2013) . . . 337
5.4 Exámenes del grupo 5 (Andrés Cordón y Miguel A. Martínez) . . . 340
5.4.1 Examen 1 (5 de Noviembre de 2013) . . . 340
5.4.2 Examen 2 (16 de Diciembre de 2013) . . . 343

A Resumen de funciones predefinidas de Haskell **347**
A.1 Resumen de funciones sobre TAD en Haskell . . . 349
A.1.1 Polinomios . . . 349
A.1.2 Vectores y matrices (`Data.Array`) . . . 349
A.1.3 Tablas . . . 350
A.1.4 Grafos . . . 350

B Método de Pólya para la resolución de problemas **353**
B.1 Método de Pólya para la resolución de problemas matemáticos . . . 353
B.2 Método de Pólya para resolver problemas de programación . . . 354

Introducción

Desde el inicio (en el curso 2009–10) del Grado en Matemática de la Universidad de Sevilla se estudia, en la asignatura de Informática de primero, una introducción a la programación funcional con Haskell.

Durante este tiempo he ido publicando materiales para la asignatura que he recopilado en dos libros:

- Temas de programación funcional[1]
- Piensa en Haskell (Ejercicios de programación funcional con Haskell)[2]

Este libro completa los anteriores presentando una recopilación de los exámenes de la asignatura durante estos años.

Los exámenes se realizaron en el aula de informática y su duración fue de 2 horas. Durante el examen se podía usar el resumen de funciones de Haskell del apéndice A. La materia de cada examen es la impartida desde el comienzo del curso (generalmente, el 1 de octubre) hasta la fecha del examen.

La asignatura está más orientada a la resolución de problemas con Haskell (usando el método de Polya del apéndice B), que al estudio de las particularidades de Haskell.

En algunos ejercicios se usan tipos abstractos de datos estudiados en la asignatura tal como se explica en [**?**]. Sus implementaciones se encuentra en la página de los códigos[3]

El libro consta de 5 capítulos correspondientes a los 5 cursos en los que en se ha impartido la asignatura. En cada capítulo hay una sección, por cada uno de los grupos de la asignatura, y una subsección 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 de la sección). Sin embargo, los he modificado para unificar el estilo de su presentación.

En resumen, el libro contiene 80 exámenes y 535 ejercicios.

José A. Alonso
Sevilla, 22 de diciembre de 2013

[1] http://www.cs.us.es/~jalonso/publicaciones/2013-Temas_de_PF_con_Haskell.pdf
[2] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[3] http://www.cs.us.es/~jalonso/cursos/i1m-12/codigos.html

1

Exámenes del curso 2009–10

1.1. Exámenes del grupo 1 (José A. Alonso y Gonzalo Aranda)

1.1.1. Examen 1 (30 de noviembre de 2009)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (30 de noviembre de 2009)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por recursión ,la función
--    sumaFactR :: Int -> Int
-- tal que (sumaFactR n) es la suma de los factoriales de los números
-- desde 0 hasta n. Por ejemplo,
--    sumaFactR 3  ==  10
-- ---------------------------------------------------------------------

sumaFactR :: Int -> Int
sumaFactR 0     = 1
sumaFactR (n+1) = factorial (n+1) + sumaFactR n

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por comprensión, la función
```

```
--    sumaFactC :: Int -> Int
-- tal que (sumaFactC n) es la suma de los factoriales de los números
-- desde 0 hasta n. Por ejemplo,
--    sumaFactC 3  ==  10
-- ---------------------------------------------------------------------

sumaFactC :: Int -> Int
sumaFactC n = sum [factorial x | x <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión, la función
--    copia :: [a] -> Int -> [a]
-- tal que (copia xs n) es la lista obtenida copiando n veces la lista
-- xs. Por ejemplo,
--    copia "abc" 3  ==  "abcabcabc"
-- ---------------------------------------------------------------------

copia :: [a] -> Int -> [a]
copia xs 0 = []
copia xs n = xs ++ copia xs (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por recursión, la función
--    incidenciasR :: Eq a => a -> [a] -> Int
-- tal que (incidenciasR x ys) es el número de veces que aparece el
-- elemento x en la lista ys. Por ejemplo,
--    incidenciasR 3 [7,3,5,3]  ==  2
-- ---------------------------------------------------------------------

incidenciasR :: Eq a => a -> [a] -> Int
incidenciasR _ []                 = 0
incidenciasR x (y:ys) | x == y    = 1 + incidenciasR x ys
                      | otherwise = incidenciasR x ys

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por comprensión, la función
--    incidenciasC :: Eq a => a -> [a] -> Int
-- tal que (incidenciasC x ys) es el número de veces que aparece el
-- elemento x en la lista ys. Por ejemplo,
--    incidenciasC 3 [7,3,5,3]  ==  2
```

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

incidenciasC :: Eq a => a -> [a] -> Int
incidenciasC x ys = length [y | y <- ys, y == x]
```

1.1.2. Examen 2 (12 de febrero de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (12 de febrero de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    diferenciasR :: Num a => [a] -> [a]
-- tal que (diferenciasR xs) es la lista de las diferencias entre los
-- elementos consecutivos de xs. Por ejemplo,
--    diferenciasR [5,3,8,7]  == [2,-5,1]
-- ---------------------------------------------------------------------

diferenciasR :: Num a => [a] -> [a]
diferenciasR []         = []
diferenciasR [_]        = []
diferenciasR (x1:x2:xs) = (x1-x2) : diferenciasR (x2:xs)

-- La definición anterior puede simplificarse
diferenciasR' :: Num a => [a] -> [a]
diferenciasR' (x1:x2:xs) = (x1-x2) : diferenciasR' (x2:xs)
diferenciasR' _          = []

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por comprensión, la función
--    diferenciasC :: Num a => [a] -> [a]
-- tal que (diferenciasC xs) es la lista de las diferencias entre los
-- elementos consecutivos de xs. Por ejemplo,
--    diferenciasC [5,3,8,7]  == [2,-5,1]
-- ---------------------------------------------------------------------

diferenciasC :: Num a => [a] -> [a]
diferenciasC xs = [a-b | (a,b) <- zip xs (tail xs)]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    producto :: [[a]] -> [[a]]
-- tal que (producto xss) es el producto cartesiano de los conjuntos
-- xss. Por ejemplo,
--    ghci> producto [[1,3],[2,5]]
--    [[1,2],[1,5],[3,2],[3,5]]
--    ghci> producto [[1,3],[2,5],[6,4]]
--    [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
--    ghci> producto [[1,3,5],[2,4]]
--    [[1,2],[1,4],[3,2],[3,4],[5,2],[5,4]]
--    ghci> producto []
--    [[]]
-- ---------------------------------------------------------------------

producto :: [[a]] -> [[a]]
producto []       = [[]]
producto (xs:xss) = [x:ys | x <- xs, ys <- producto xss]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir el predicado
--    comprueba :: [[Int]] -> Bool
-- tal que tal que (comprueba xss) se verifica si cada elemento de la
-- lista de listas xss contiene algún número par. Por ejemplo,
--    comprueba [[1,2],[3,4,5],[8]]  ==  True
--    comprueba [[1,2],[3,5]]        ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
comprueba :: [[Int]] -> Bool
comprueba xss = and [or [even x | x <- xs] | xs <- xss]

-- 2ª definición (por recursión):
compruebaR :: [[Int]] -> Bool
compruebaR [] = True
compruebaR (xs:xss) = tienePar xs && compruebaR xss

-- (tienePar xs) se verifica si xs contiene algún número par.
tienePar  :: [Int] -> Bool
```

```
tienePar []     = False
tienePar (x:xs) = even x || tienePar xs

-- 3ª definición (por plegado):
compruebaP :: [[Int]] -> Bool
compruebaP = foldr ((&&) . tienePar) True

-- (tieneParP xs) se verifica si xs contiene algún número par.
tieneParP  :: [Int] -> Bool
tieneParP = foldr ((||) . even) False

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    pertenece :: Ord a => a -> [a] -> Bool
-- tal que (pertenece x ys) se verifica si x pertenece a la lista
-- ordenada creciente, finita o infinita, ys. Por ejemplo,
--    pertenece 22 [1,3,22,34]  ==  True
--    pertenece 22 [1,3,34]     ==  False
--    pertenece 23 [1,3..]      ==  True
--    pertenece 22 [1,3..]      ==  False
-- ---------------------------------------------------------------------

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

-- La definición de pertenece puede simplificarse
pertenece' :: Ord a => a -> [a] -> Bool
pertenece' x ys = x `elem` takeWhile (<= x) ys
```

1.1.3. Examen 3 (15 de marzo de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 3º examen de evaluación continua (15 de marzo de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

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

```
-- Ejercicio 1.1.1. Definir, por recursión, la función
--    pares :: [Int] -> [Int]
-- tal que (pares xs) es la lista de los elementos pares de xs. Por
-- ejemplo,
--    pares [2,5,7,4,6,8,9]    ==  [2,4,6,8]
-- ---------------------------------------------------------------------

pares   :: [Int] -> [Int]
pares [] = []
pares (x:xs) | even x    = x : pares xs
             | otherwise = pares xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.1.2. Definir, por recursión, la función
--    impares :: [Int] -> [Int]
-- tal que (impares xs) es la lista de los elementos impares de xs. Por
-- ejemplo,
--    impares [2,5,7,4,6,8,9]  ==  [5,7,9]
-- ---------------------------------------------------------------------

impares   :: [Int] -> [Int]
impares [] = []
impares (x:xs) | odd x     = x : impares xs
               | otherwise = impares xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.1.3. Definir, por recursión, la función
--    suma :: [Int] -> Int
-- tal que (suma xs) es la suma de los elementos de xs. Por ejemplo,
--    suma [2,5,7,4,6,8,9]  ==  41
-- ---------------------------------------------------------------------

suma :: [Int] -> Int
suma []     = 0
suma (x:xs) = x + suma xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que la suma de la suma de
-- (pares xs) y la suma de (impares xs) es igual que la suma de xs.
-- ---------------------------------------------------------------------
```

```
-- La propiedad es
prop_pares :: [Int] -> Bool
prop_pares xs =
    suma (pares xs) + suma (impares xs) == suma xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3 Demostrar por inducción que que la suma de la suma de
-- (pares xs) y la suma de (impares xs) es igual que la suma de xs.
-- ---------------------------------------------------------------------

{-
 Demostración:
 La propiedad que hay que demostrar es
    suma (pares xs) + suma (impares xs) = suma xs

 Caso base: Hay que demostrar que
    suma (pares []) + suma (impares []) = suma []
 En efecto,
    suma (pares []) + suma (impares [])
    = suma [] + suma []                    [por pares.1 e impares.1]
    = 0 + 0                                [por suma.1]
    = 0                                    [por aritmética]
    = suma []                              [por suma.1]

 Paso de inducción: Se supone que la hipótesis de inducción
    suma (pares xs) + suma (impares xs) = suma xs
 Hay que demostrar que
    suma (pares (x:xs)) + suma (impares (x:xs)) = suma (x:xs)
 Lo demostraremos distinguiendo dos casos

 Caso 1: Supongamos que x es par. Entonces,
     suma (pares (x:xs)) + suma (impares (x:xs))
     = suma (x:pares xs) + suma (impares xs)       [por pares.2, impares.3]
     = x + suma (pares xs) + suma (impares xs)     [por suma.2]
     = x + suma xs                                 [por hip. de inducción]
```

```
     = suma (x:xs)                                    [por suma.2]

 Caso 1: Supongamos que x es impar. Entonces,
     suma (pares (x:xs)) + suma (impares (x:xs))
     = suma (pares xs) + suma (x:impares xs)          [por pares.3, impares.2]
     = suma (pares xs) + x + suma (impares xs)        [por suma.2]
     = x + suma xs                                    [por hip. de inducción]
     = suma (x:xs)                                    [por suma.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.1.1. Definir, por recursión, la función
--    duplica :: [a] -> [a]
-- tal que (duplica xs) es la lista obtenida duplicando los elementos de
-- xs. Por ejemplo,
--    duplica  [7,2,5]  ==  [7,7,2,2,5,5]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1.2. Definir, por recursión, la función
--    longitud :: [a] -> Int
-- tal que (longitud xs) es el número de elementos de xs. Por ejemplo,
--    longitud [7,2,5]  ==  3
-- ---------------------------------------------------------------------

longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que (longitud (duplica xs))
-- es el doble de (longitud xs), donde xs es una lista de números
-- enteros.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```
prop_duplica :: [Int] -> Bool
prop_duplica xs =
    longitud (duplica xs) == 2 * longitud xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Demostrar por inducción que la longitud de
-- (duplica xs) es el doble de la longitud de xs.
-- ---------------------------------------------------------------------

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

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    listasMayores :: [[Int]] -> [[Int]]
-- tal que (listasMayores xss) es la lista de las listas de xss de mayor
-- suma. Por ejemplo,
--    ghci> listasMayores [[1,3,5],[2,7],[1,1,2],[3],[5]]
--    [[1,3,5],[2,7]]
-- ---------------------------------------------------------------------

listasMayores :: [[Int]] -> [[Int]]
listasMayores xss = [xs | xs <- xss, sum xs == m]
    where m = maximum [sum xs | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que todas las listas de
-- (listasMayores xss) tienen la misma suma.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_listasMayores :: [[Int]] -> Bool
prop_listasMayores xss =
    iguales [sum xs | xs <- listasMayores xss]

-- (iguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    iguales [2,2,2]  ==  True
--    iguales [2,3,2]  ==  False
iguales :: Eq a => [a] -> Bool
iguales (x1:x2:xs) = x1 == x2 && iguales (x2:xs)
iguales _          = True

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

1.1.4. Examen 4 (12 de abril de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 4º examen de evaluación continua (12 de abril de 2010)
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. En los apartados de este ejercicio se usará el tipo de
-- árboles binarios definidos como sigue
--    data Arbol a = Hoja
--                 | Nodo a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- En los ejemplos se usará el siguiente árbol
--    ejArbol :: Arbol Int
--    ejArbol = Nodo 2
--                    (Nodo 5
--                          (Nodo 3 Hoja Hoja)
--                          (Nodo 7 Hoja Hoja))
--                    (Nodo 4 Hoja Hoja)
--
-- Definir por recursión la función
--    sumaArbol :: Num a => Arbol a -> a
-- tal (sumaArbol x) es la suma de los valores que hay en el árbol
-- x. Por ejemplo,
--    sumaArbol ejArbol  ==  21
-- ---------------------------------------------------------------------

data Arbol a = Hoja
             | Nodo a (Arbol a) (Arbol a)
             deriving (Show, Eq)

ejArbol :: Arbol Int
ejArbol = Nodo 2
               (Nodo 5
                     (Nodo 3 Hoja Hoja)
                     (Nodo 7 Hoja Hoja))
               (Nodo 4 Hoja Hoja)

sumaArbol :: Num a => Arbol a -> a
sumaArbol Hoja = 0
sumaArbol (Nodo x i d) = x + sumaArbol i + sumaArbol d

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

```
--    nodos :: Arbol a -> [a]
-- tal que (nodos x) es la lista de los nodos del árbol x. Por ejemplo.
--    nodos ejArbol  ==  [2,5,3,7,4]
-- ---------------------------------------------------------------------

nodos :: Arbol a -> [a]
nodos Hoja = []
nodos (Nodo x i d) = x : nodos i ++ nodos d

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Demostrar por inducción que para todo árbol a,
-- sumaArbol a = sum (nodos a).
-- Indicar la propiedad de sum que se usa en la demostración.
-- ---------------------------------------------------------------------

{-
 Caso base: Hay que demostrar que
    sumaArbol Hoja = sum (nodos Hoja)
 En efecto,
    sumaArbol Hoja
    = 0                   [por sumaArbol.1]
    = sum []              [por suma.1]
    = sum (nodos Hoja)    [por nodos.1]

 Caso inductivo: Se supone la hipótesis de inducción
    sumaArbol i = sum (nodos i)
    sumaArbol d = sum (nodos d)
 Hay que demostrar que
    sumaArbol (Nodo x i d) = sum (nodos (Nodo x i d))
 En efecto,
    sumaArbol (Nodo x i d)
    = x + sumaArbol i + sumaArbol d       [por sumaArbol.2]
    = x + sum (nodos i) + sum (nodos d)   [por hip. de inducción]
    = x + sum (nodos i ++ nodos d)        [por propiedad de sum]
    = sum(x:(nodos i)++(nodos d))         [por sum.2]
    = sum (Nodos x i d)                   [por nodos.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la constante
```

```
--    pares :: Int
-- tal que pares es la lista de todos los pares de números enteros
-- positivos ordenada según la suma de sus componentes y el valor de la
-- primera componente. Por ejemplo,
--    ghci> take 11 pares
--    [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1),(1,5)]
-- ---------------------------------------------------------------------

pares :: [(Integer,Integer)]
pares = [(x,z-x) | z <- [1..], x <- [1..z-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la constante
--    paresDestacados :: [(Integer,Integer)]
-- tal que paresDestadados es la lista de pares de números enteros (x,y)
-- tales que 11 divide a x+13y y 13 divide a x+11y.
-- ---------------------------------------------------------------------

paresDestacados :: [(Integer,Integer)]
paresDestacados = [(x,y) | (x,y) <- pares,
                           x+13*y `rem` 11 == 0,
                           x+11*y `rem` 13 == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la constante
--    parDestacadoConMenorSuma :: Integer
-- tal que parDestacadoConMenorSuma es el par destacado con menor suma y
-- calcular su valor y su posición en la lista pares.
-- ---------------------------------------------------------------------

-- La definición es
parDestacadoConMenorSuma :: (Integer,Integer)
parDestacadoConMenorSuma = head paresDestacados

-- El valor es
--   ghci> parDestacadoConMenorSuma
--   (23,5)

-- La posición es
--    ghci> 1 + length (takeWhile (/=parDestacadoConMenorSuma) pares)
```

```
--    374

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    limite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b
-- tal que (limite f a) es el valor de f en el primer término x tal que
-- para todo y entre x+1 y x+100, el valor absoluto de f(y)-f(x) es
-- menor que a. Por ejemplo,
--    limite (\n -> (2*n+1)/(n+5)) 0.001  ==  1.9900110987791344
--    limite (\n -> (1+1/n)**n) 0.001     ==  2.714072874546881
-- ---------------------------------------------------------------------

limite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b
limite f a =
    head [f x | x <- [1..],
                maximum [abs(f y - f x) | y <- [x+1..x+100]] < a]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    esLimite :: (Num a, Enum a, Num b, Ord b) =>
--                (a -> b) -> b -> b -> Bool
-- tal que (esLimite f b a) se verifica si existe un x tal que para todo
-- y entre x+1 y x+100, el valor absoluto de f(y)-b es menor que a. Por
-- ejemplo,
--    esLimite (\n -> (2*n+1)/(n+5)) 2 0.01     ==  True
--    esLimite (\n -> (1+1/n)**n) (exp 1) 0.01  ==  True
-- ---------------------------------------------------------------------

esLimite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b -> Bool
esLimite f b a =
    not (null [x | x <- [1..],
                   maximum [abs(f y - b) | y <- [x+1..x+100]] < a])
```

1.1.5. Examen 5 (17 de mayo de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 5º examen de evaluación continua (17 de mayo de 2010)
-- ---------------------------------------------------------------------

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

```
-- Ejercicio 1.1. Definir Haskell la función
--    primo :: Int -> Integer
-- tal que (primo n) es el n-ésimo número primo. Por ejemplo,
--    primo 5 = 11
-- ---------------------------------------------------------------------

primo :: Int -> Integer
primo n = primos !! (n-1)

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    sumaCifras :: Integer -> Integer
-- tal que (sumaCifras n) es la suma de las cifras del número n. Por
-- ejemplo,
--    sumaCifras 325 = 10
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    primosSumaPar :: Int -> [Integer]
-- tal que (primosSumaPar n) es el conjunto de elementos del conjunto de
-- los n primeros primos tales que la suma de sus cifras es par. Por
-- ejemplo,
--    primosSumaPar 10 = [2,11,13,17,19]
-- ---------------------------------------------------------------------
```

```
primosSumaPar :: Int -> [Integer]
primosSumaPar n =
   [x | x <- take n primos, even (sumaCifras x)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
--    numeroPrimosSumaPar :: Int -> Int
-- tal que (numeroPrimosSumaPar n) es la cantidad de elementos del
-- conjunto de los n primeros primos tales que la suma de sus cifras es
-- par. Por ejemplo,
--    numeroPrimosSumaPar 10 = 5
-- ---------------------------------------------------------------------

numeroPrimosSumaPar :: Int -> Int
numeroPrimosSumaPar = length . primosSumaPar

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Definir la función
--    puntos :: Int -> [(Int,Int)]
-- tal que (puntos n) es la lista de los puntos de la forma (x,y) donde x
-- toma los valores 0,10,20,...,10*n e y es la cantidad de elementos del
-- conjunto de los x primeros primos tales que la suma de sus cifras es
-- par. Por ejemplo,
--    puntos 5 = [(0,0),(10,5),(20,10),(30,17),(40,21),(50,23)]
-- ---------------------------------------------------------------------

puntos :: Int -> [(Int,Int)]
puntos n = [(i,numeroPrimosSumaPar i) | i <- [0,10..10*n]]
```

1.1.6. Examen 6 (21 de junio de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 6º examen de evaluación continua (21 de junio de 2010)
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    calculaPi :: Int -> Double
-- tal que (calculaPi n) es la aproximación del número pi calculada
```

```
-- mediante la expresión
--    4*(1 - 1/3 + 1/5 - 1/7 ... 1/(2*n+1))
-- Por ejemplo,
--    calculaPi 3    ==  2.8952380952380956
--    calculaPi 300  ==  3.1449149035588526
-- Indicación: La potencia es **, por ejemplo 2**3 es 8.0.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. En la Olimpiada de Matemática del 2010 se planteó el
-- siguiente problema:
--   Una sucesión pucelana es una sucesión creciente de 16 números
--   impares positivos consecutivos, cuya suma es un cubo perfecto.
--   ¿Cuántas sucesiones pucelanas tienen solamente números de tres
--   cifras?
--
-- Definir la función
--    pucelanas :: [[Int]]
-- tal que pucelanas es la lista de las sucesiones pucelanas. Por
-- ejemplo,
--    ghci> head pucelanas
--    [17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47]
-- ---------------------------------------------------------------------

pucelanas :: [[Int]]
pucelanas = [[x,x+2..x+30] | x <- [1..],
                             esCubo (sum [x,x+2..x+30])]

-- (esCubo n) se verifica si n es un cubo. Por ejemplo,
--    esCubo 27  ==  True
--    esCubo 28  ==  False
esCubo x = y^3== x
    where y = ceiling (fromIntegral x ** (1/3))

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    pucelanasConNcifras :: Int -> [[Int]]
```

```
-- tal que (pucelanasConNcifras n) es la lista de las sucesiones
-- pucelanas que tienen sólamente números de n cifras. Por ejemplo,
--    ghci> pucelanasConNcifras 2
--    [[17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47]]
-- ---------------------------------------------------------------------

pucelanasConNcifras :: Int -> [[Int]]
pucelanasConNcifras n = [[x,x+2..x+30] | x <- [10^(n-1)+1..10^n-31],
                                         esCubo (sum [x,x+2..x+30])]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Calcular cuántas sucesiones pucelanas tienen solamente
-- números de tres cifras.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> length (pucelanasConNcifras 3)
--    3

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    inflexion :: Ord a => [a] -> Maybe a
-- tal que (inflexion xs) es el primer elemento de la lista en donde se
-- cambia de creciente a decreciente o de decreciente a creciente y
-- Nothing si no se cambia. Por ejemplo,
--    inflexion [2,2,3,5,4,6]    ==  Just 4
--    inflexion [9,8,6,7,10,10]  ==  Just 7
--    inflexion [2,2,3,5]        ==  Nothing
--    inflexion [5,3,2,2]        ==  Nothing
-- ---------------------------------------------------------------------

inflexion :: Ord a => [a] -> Maybe a
inflexion (x:y:zs)
    | x <  y = decreciente (y:zs)
    | x == y = inflexion (y:zs)
    | x >  y = creciente (y:zs)
inflexion _  = Nothing

-- (creciente xs) es el segundo elemento de la primera parte creciente
-- de xs y Nothing, en caso contrario. Por ejemplo,
```

```
--    creciente [4,3,5,6]  ==  Just 5
--    creciente [4,3,5,2,7]  ==  Just 5
--    creciente [4,3,2]  ==  Nothing
creciente (x:y:zs)
    | x <  y    = Just y
    | otherwise = creciente (y:zs)
creciente _     = Nothing

-- (decreciente xs) es el segundo elemento de la primera parte
-- decreciente de xs y Nothing, en caso contrario. Por ejemplo,
--    decreciente [4,2,3,1,0]  ==  Just 2
--    decreciente [4,5,3,1,0]  ==  Just 3
--    decreciente [4,5,7]      ==  Nothing
decreciente (x:y:zs)
    | x >  y    = Just y
    | otherwise = decreciente (y:zs)
decreciente _   = Nothing
```

1.1.7. Examen 7 (5 de julio de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la 1ª convocatoria (5 de julio de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    numeroCerosFactorial :: Integer -> Integer
-- tal que (numeroCerosFactorial n) es el número de ceros con los que
-- termina el factorial de n. Por ejemplo,
--    numeroCerosFactorial 17 == 3
-- ---------------------------------------------------------------------

numeroCerosFactorial :: Integer -> Integer
numeroCerosFactorial n = numeroCeros (product [1..n])

-- (numeroCeros x) es el número de ceros con los que termina x. Por
-- ejemplo,
```

```
--    numeroCeros 35400  ==  2
numeroCeros :: Integer -> Integer
numeroCeros x | mod x 10 /= 0 = 0
              | otherwise     = 1 + numeroCeros (div x 10)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las matrices pueden representarse mediante una lista de
-- listas donde cada una de las lista representa una fila  de la
-- matriz. Por ejemplo, la matriz
--    |1 0 -2|
--    |0 3 -1|
-- puede representarse por [[1,0,-2],[0,3,-1]]. Definir la función
--    producto :: Num t => [[t]] -> [[t]] -> [[t]]
-- tal que (producto a b) es el producto de las matrices a y b. Por
-- ejemplo,
--    ghci> producto [[1,0,-2],[0,3,-1]] [[0,3],[-2,-1],[0,4]]
--    [[0,-5],[-6,-7]]
-- ---------------------------------------------------------------------

producto :: Num t => [[t]] -> [[t]] -> [[t]]
producto a b =
    [[sum [x*y | (x,y) <- zip fil col] | col <- transpose b] | fil <- a]

-- ---------------------------------------------------------------------
-- Ejercicio 3. El ejercicio 4 de la Olimpiada Matemáticas de 1993 es el
-- siguiente:
--    Demostrar que para todo número primo p distinto de 2 y de 5,
--    existen infinitos múltiplos de p de la forma 1111......1 (escrito
--    sólo con unos).
-- Definir la función
--    multiplosEspeciales :: Integer -> Int -> [Integer]
-- tal que (multiplosEspeciales p n) es una lista de n múltiplos p de la
-- forma 1111...1 (escrito sólo con unos), donde p es un número primo
-- distinto de 2 y 5. Por ejemplo,
--    multiplosEspeciales 7 2  ==  [111111,111111111111]
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    recorridos :: [a] -> [[a]]
-- tal que (recorridos xs) es la lista de todos los posibles recorridos
-- por el grafo cuyo conjunto de vértices es xs y cada vértice se
-- encuentra conectado con todos los otros y los recorridos pasan por
-- todos los vértices una vez y terminan en el vértice inicial. Por
-- ejemplo,
--    ghci> recorridos [2,5,3]
--    [[2,5,3,2],[5,2,3,5],[3,5,2,3],[5,3,2,5],[3,2,5,3],[2,3,5,2]]
-- Indicación: No importa el orden de los recorridos en la lista.
-- ---------------------------------------------------------------------

recorridos :: [a] -> [[a]]
recorridos xs = [(y:ys)++[y] | (y:ys) <- permutations xs]
```

1.1.8. Examen 8 (15 de septiembre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la 2ª convocatoria (15 de septiembre de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

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

```
-- Ejercicio 1.1. Definir la función
--    diagonal :: [[a]] -> [a]
-- tal que (diagonal m) es la diagonal de la matriz m. Por ejemplo,
--    diagonal [[3,5,2],[4,7,1],[6,9,0]]  ==  [3,7,0]
--    diagonal [[3,5,2],[4,7,1]]          ==  [3,7]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
diagonal :: [[a]] -> [a]
diagonal ((x1:_):xs) = x1 : diagonal [tail x | x <- xs]
diagonal _ = []

-- Segunda definición (sin recursión):
diagonal2 :: [[a]] -> [a]
diagonal2 = flip (zipWith (!!)) [0..]

-- ---------------------------------------------------------------------.
-- Ejercicio 1.2. Definir la función
--    matrizDiagonal ::  Num a => [a] -> [[a]]
-- tal que (matrizDiagonal xs) es la matriz cuadrada cuya diagonal es el
-- vector xs y los restantes elementos son iguales a cero. Por ejemplo,
--    matrizDiagonal [2,5,3]  ==  [[2,0,0],[0,5,0],[0,0,3]]
-- ---------------------------------------------------------------------

matrizDiagonal :: Num a => [a] -> [[a]]
matrizDiagonal [] = []
matrizDiagonal (x:xs) =
    (x: [0 | _ <- xs]) : [0:zs | zs <- ys]
    where ys = matrizDiagonal xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck si se verifican las
-- siguientes propiedades:
-- 1. Para cualquier lista xs, (diagonal (matrizDiagonal xs)) es igual a
--    xs.
-- 2. Para cualquier matriz m, (matrizDiagonal (diagonal m)) es igual a
--    m.
-- ---------------------------------------------------------------------

-- La primera propiedad es
```

```
prop_diagonal1 :: [Int] -> Bool
prop_diagonal1 xs =
    diagonal (matrizDiagonal xs) == xs

-- La comprobación es
--    ghci> quickCheck prop_diagonal1
--    +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_diagonal2 :: [[Int]] -> Bool
prop_diagonal2 m =
    matrizDiagonal (diagonal m) == m

-- La comprobación es
--    ghci> quickCheck prop_diagonal2
--    *** Failed! Falsifiable (after 4 tests and 5 shrinks):
--    [[0,0]]
-- lo que indica que la propiedad no se cumple y que [[0,0]] es un
-- contraejemplo,

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. El enunciado del problema 1 de la Fase nacional de la
-- Olimpiada Matemática Española del 2009 dice:
--    Hallar todas las sucesiones finitas de n números naturales
--    consecutivos a1, a2, ..., an, con n ≥ 3, tales que
--    a1 + a2 + ... + an = 2009.
--
-- En este ejercicio vamos a resolver el problema con Haskell.
--
-- Definir la función
--    sucesionesConSuma :: Int -> [[Int]]
-- tal que (sucesionesConSuma x) es la lista de las sucesiones finitas
-- de n números naturales consecutivos a1, a2, ..., an, con n ≥ 3, tales
-- que
--    a1 + a2 + ... + an = x.
-- Por ejemplo.
--    sucesionesConSuma 9   ==  [[2,3,4]]
--    sucesionesConSuma 15  ==  [[1,2,3,4,5],[4,5,6]]
-- ---------------------------------------------------------------------
```

```
-- 1ª definición:
sucesionesConSuma :: Int -> [[Int]]
sucesionesConSuma x =
    [[a..b] | a <- [1..x], b <- [a+2..x], sum [a..b] == x]

-- 2ª definición (con la fórmula de la suma de las progresiones
-- aritméticas):
sucesionesConSuma' :: Int -> [[Int]]
sucesionesConSuma' x =
    [[a..b] | a <- [1..x], b <- [a+2..x], (a+b)*(b-a+1) `div` 2 == x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Resolver el problema de la Olimpiada con la función
-- sucesionesConSuma.
-- ---------------------------------------------------------------------

-- Las soluciones se calculan con
--    ghci> sucesionesConSuma' 2009
--    [[17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
--      38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,
--      59,60,61,62,63,64,65],
--     [29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
--      50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69],
--     [137,138,139,140,141,142,143,144,145,146,147,148,149,150],
--     [284,285,286,287,288,289,290]]
-- Por tanto, hay 4 soluciones.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    sumasNcuadrados :: Int -> Int -> [[Int]]
-- tal que (sumasNcuadrados x n) es la lista de las descomposiciones de
-- x en sumas decrecientes de n cuadrados. Por ejemplo,
--    sumasNcuadrados 10 4  ==  [[3,1,0,0],[2,2,1,1]]
-- ---------------------------------------------------------------------

sumasNcuadrados :: Int -> Int -> [[Int]]
sumasNcuadrados x 1 | a^2 == x  = [[a]]
                    | otherwise = []
    where a = ceiling (sqrt (fromIntegral x))
sumasNcuadrados x n =
```

```
    [a:y:ys | a <- [x',x'-1..0],
              (y:ys) <- sumasNcuadrados (x-a^2) (n-1),
              y <= a]
    where x' = ceiling (sqrt (fromIntegral x))

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    numeroDeCuadrados :: Int -> Int
-- tal que (numeroDeCuadrados x) es el menor número de cuadrados que se
-- necesita para escribir x como una suma de cuadrados. Por ejemplo,
--    numeroDeCuadrados 6  ==  3
--    sumasNcuadrados 6 3  ==  [[2,1,1]]
-- ---------------------------------------------------------------------

numeroDeCuadrados :: Int -> Int
numeroDeCuadrados x = head [n | n <- [1..], sumasNcuadrados x n /= []]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Calcular el menor número n tal que todos los números
-- de 0 a 100 pueden expresarse como suma de n cuadrados.
-- ---------------------------------------------------------------------

-- El cálculo de n es
--    ghci> maximum [numeroDeCuadrados x | x <- [0..100]]
--    4

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Comprobar con QuickCheck si todos los números
-- positivos pueden expresarse como suma de n cuadrados (donde n es el
-- número calculado anteriormente).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_numeroDeCuadrados x =
    x >= 0 ==> numeroDeCuadrados x <= 4

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

1.1.9. Examen 9 (17 de diciembre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la 3ª convocatoria (17 de diciembre de 2010)
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    ullman :: (Num a, Ord a) => a -> Int -> [a] -> Bool
-- tal que (ullman t k xs) se verifica si xs tiene un subconjunto con k
-- elementos cuya suma sea menor que t. Por ejemplo,
--    ullman 9 3 [1..10] == True
--    ullman 5 3 [1..10] == False
-- ---------------------------------------------------------------------

-- 1ª solución (corta y eficiente)
ullman :: (Ord a, Num a) => a -> Int -> [a] -> Bool
ullman t k xs = sum (take k (sort xs)) < t

-- 2ª solución (larga e ineficiente)
ullman2 :: (Num a, Ord a) => a -> Int -> [a] -> Bool
ullman2 t k xs =
    [ys | ys <- subconjuntos xs, length ys == k, sum ys < t] /= []

-- (subconjuntos xs) es la lista de los subconjuntos de xs. Por
-- ejemplo,
--    subconjuntos "bc"  ==  ["","c","b","bc"]
--    subconjuntos "abc" ==  ["","c","b","bc","a","ac","ab","abc"]
subconjuntos :: [a] -> [[a]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = zss++[x:ys | ys <- zss]
    where zss = subconjuntos xs

-- Los siguientes ejemplos muestran la diferencia en la eficencia:
--    ghci> ullman 9 3 [1..20]
--    True
--    (0.02 secs, 528380 bytes)
--    ghci> ullman2 9 3 [1..20]
--    True
```

```
--    (4.08 secs, 135267904 bytes)
--    ghci> ullman 9 3 [1..100]
--    True
--    (0.02 secs, 526360 bytes)
--    ghci> ullman2 9 3 [1..100]
--      C-c C-cInterrupted.
--    Agotado

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumasDe2Cuadrados :: Integer -> [(Integer, Integer)]
-- tal que (sumasDe2Cuadrados n) es la lista de los pares de números
-- tales que la suma de sus cuadrados es n y el primer elemento del par
-- es mayor o igual que el segundo. Por ejemplo,
--    sumasDe2Cuadrados 25  ==  [(5,0),(4,3)]
-- ---------------------------------------------------------------------

-- 1ª definición:
sumasDe2Cuadrados_1 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_1 n =
    [(x,y) | x <- [n,n-1..0],
             y <- [0..x],
             x*x+y*y == n]

-- 2ª definición:
sumasDe2Cuadrados2 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrado_2 n =
    [(x,y) | x <- [a,a-1..0],
             y <- [0..x],
             x*x+y*y == n]
    where a = ceiling (sqrt (fromIntegral n))

-- 3ª definición:
sumasDe2Cuadrados3 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrado_3 n = aux (ceiling (sqrt (fromIntegral n))) 0
    where aux x y | x < y          = []
                  | x*x + y*y <  n = aux x (y+1)
                  | x*x + y*y == n = (x,y) : aux (x-1) (y+1)
                  | otherwise      = aux (x-1) y
```

```
-- Comparación
--    +----------+--------------+--------------+--------------+
--    | n        | 1ª definición | 2ª definición | 3ª definición |
--    +----------+--------------+--------------+--------------+
--    |      999 | 2.17 segs     |   0.02 segs   | 0.01 segs     |
--    | 48612265 |               | 140.38 segs   | 0.13 segs     |
--    +----------+--------------+--------------+--------------+

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios pueden representarse mediante el
-- tipo de datos Arbol definido por
--    data Arbol a = Nodo (Arbol a) (Arbol a)
--                 | Hoja a
--                 deriving Show
-- Por ejemplo, los árboles
--    árbol1          árbol2       árbol3     árbol4
--       o              o            o               o
--      / \            / \          / \             / \
--     1   o          o   3        o   3           o   1
--        / \        / \          / \             / \
--       2   3      1   2        1   4           2   3
-- se representan por
--    arbol1, arbol2, arbol3, arbol4 :: Arbol Int
--    arbol1 = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
--    arbol2 = Nodo (Nodo (Hoja 1) (Hoja 2)) (Hoja 3)
--    arbol3 = Nodo (Nodo (Hoja 1) (Hoja 4)) (Hoja 3)
--    arbol4 = Nodo (Nodo (Hoja 2) (Hoja 3)) (Hoja 1)
-- Definir la función
--    igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (igualBorde t1 t2) se verifica si los bordes de los árboles
-- t1 y t2 son iguales. Por ejemplo,
--    igualBorde arbol1 arbol2  ==  True
--    igualBorde arbol1 arbol3  ==  False
--    igualBorde arbol1 arbol4  ==  False
-- ---------------------------------------------------------------------

data Arbol a = Nodo (Arbol a) (Arbol a)
             | Hoja a
             deriving Show
```

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

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

-- (borde t) es el borde del árbol t; es decir, la lista de las hojas
-- del árbol t leídas de izquierda a derecha. Por ejemplo,
--    borde arbol4  ==  [2,3,1]
borde :: Arbol a -> [a]
borde (Nodo i d) = borde i ++ borde d
borde (Hoja x)   = [x]

-- ---------------------------------------------------------------------
-- Ejercicio 4. (Basado en el problema 145 del Proyecto Euler).
-- Se dice que un número n es reversible si su última cifra es
-- distinta de 0 y la suma de n y el número obtenido escribiendo las
-- cifras de n en orden inverso es un número que tiene todas sus cifras
-- impares. Por ejemplo,
-- * 36  es reversible porque 36+63=99 tiene todas sus cifras impares,
-- * 409 es reversible porque 409+904=1313 tiene todas sus cifras
--       impares,
-- * 243 no es reversible porque 243+342=585 no tiene todas sus cifras
--       impares.
--
-- Definir la función
--    reversiblesMenores :: Int -> Int
-- tal que (reversiblesMenores n) es la cantidad de números reversibles
-- menores que n. Por ejemplo,
--    reversiblesMenores 10   == 0
--    reversiblesMenores 100  == 20
--    reversiblesMenores 1000 == 120
-- ---------------------------------------------------------------------

-- (reversiblesMenores n) es la cantidad de números reversibles menores
-- que n. Por ejemplo,
--    reversiblesMenores 10   == 0
```

```
--    reversiblesMenores 100  == 20
--    reversiblesMenores 1000 == 120
reversiblesMenores :: Int -> Int
reversiblesMenores n = length [x | x <- [1..n-1], esReversible x]

-- (esReversible n) se verifica si n es reversible; es decir, si su
-- última cifra es distinta de 0 y la suma de n y el número obtenido
-- escribiendo las cifras de n en orden inverso es un número que tiene
-- todas sus cifras impares. Por ejemplo,
--    esReversible 36  == True
--    esReversible 409 == True
esReversible :: Int -> Bool
esReversible n = rem n 10 /= 0 && impares (cifras (n + (inverso n)))

-- (impares xs) se verifica si xs es una lista de números impares. Por
-- ejemplo,
--    impares [3,5,1] == True
--    impares [3,4,1] == False
impares :: [Int] -> Bool
impares xs = and [odd x | x <- xs]

-- (inverso n) es el número obtenido escribiendo las cifras de n en
-- orden inverso. Por ejemplo,
--    inverso 3034 == 4303
inverso :: Int -> Int
inverso n = read (reverse (show n))

-- (cifras n) es la lista de las cifras del número n. Por ejemplo,
--    cifras 3034 == [3,0,3,4]
cifras :: Int -> [Int]
cifras n = [read [x] | x <- show n]
```

1.2. Exámenes del grupo 3 (María J. Hidalgo)

1.2.1. Examen 1 (4 de diciembre de 2009)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (4 de diciembre de 2009)
-- ---------------------------------------------------------------------
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por composición, la función
--    insertaEnposicion :: a -> Int -> [a] -> [a]
-- tal que (insertaEnposicion x n xs) es la lista obtenida insertando x
-- en xs en la posición n. Por ejemplo,
--    insertaEnposicion 80 4 [1,2,3,4,5,6,7,8] == [1,2,3,80,4,5,6,7,8]
--    insertaEnposicion 'a' 1 "hola"             == "ahola"
-- ---------------------------------------------------------------------

insertaEnposicion :: a -> Int -> [a] -> [a]
insertaEnposicion x n xs = take (n-1) xs ++ [x] ++ drop (n-1) xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. El algoritmo de Euclides para calcular el máximo común
-- divisor de dos números naturales a y b es el siguiente:
--    Si b = 0, entonces mcd(a,b) = a
--    Si b > 0, entonces mcd(a,b) = mcd(b,c), donde c es el resto de
--              dividir a entre b
--
-- Definir la función
--    mcd :: Int -> Int -> Int
-- tal que (mcd a b) es el máximo común divisor de a y b calculado
-- usando el algoritmo de Euclides. Por ejemplo,
--    mcd 2 3      ==  1
--    mcd 12 30    ==  6
--    mcd 700 300  ==  100
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    esSubconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (esSubconjunto xs ys) se verifica si todos los elementos de
```

```
-- xs son también elementos de ys. Por ejemplo,
--    esSubconjunto [3,5,2,1,1,1,6,3] [1,2,3,5,6,7]   == True
--    esSubconjunto [3,2,1,1,1,6,3] [1,2,3,5,6,7]     == True
--    esSubconjunto [3,2,1,8,1,6,3] [1,2,3,5,6,7]     == False
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
esSubconjunto :: Eq a => [a] -> [a] -> Bool
esSubconjunto [] ys     = True
esSubconjunto (x:xs) ys = x `elem` ys && esSubconjunto xs ys

-- 2ª definición (por comprensión):
esSubconjunto2 :: Eq a => [a] -> [a] -> Bool
esSubconjunto2 xs ys = and [x `elem` ys | x <- xs]

-- 3ª definición (por plegado):
esSubconjunto3 :: Eq a => [a] -> [a] -> Bool
esSubconjunto3 xs ys = foldr (\ x -> (&&) (x `elem` ys)) True xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    igualConjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (igualConjunto xs ys) se verifica si xs e ys son iguales como
-- conjuntos. Por ejemplo,
--    igualConjunto [3,2,1,8,1,6,3] [1,2,3,5,6,7]     == False
--    igualConjunto [3,2,1,1,1,6,3] [1,2,3,5,6,7]     == False
--    igualConjunto [3,2,1,1,1,6,3] [1,2,3,5,6]       == False
--    igualConjunto [3,2,1,1,1,6,3,5] [1,2,3,5,6]     == True
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por comprensión la función
--    repiteC :: Int -> [a] -> [a]
-- tal que (repiteC n xs) es la lista que resulta de repetir cada
-- elemento de xs n veces. Por ejemplo,
--    repiteC 5 "Hola" == "HHHHHooooollllllaaaaa"
-- ---------------------------------------------------------------------
```

```
repiteC :: Int -> [a] -> [a]
repiteC n xs = [x | x <- xs, i <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir por recursión la función
--    repiteR :: Int -> [a] -> [a]
-- tal que (repiteR n xs) es la lista que resulta de repetir cada
-- elemento de xs n veces. Por ejemplo,
--    repiteR 5 "Hola" == "HHHHHooooollllllaaaaa"
-- ---------------------------------------------------------------------

repiteR :: Int -> [a] -> [a]
repiteR _ []     = []
repiteR n (x:xs) = replicate n x ++ repiteR n xs
```

1.2.2. Examen 2 (16 de marzo de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (16 de marzo de 2010)
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Probar por inducción que para toda lista xs:
--    length (reverse xs) = length xs
--
-- Nota: Las definiciones recursivas de length y reverse son:
--
--    length [] = 0                        -- length.1
--    length (x:xs) = 1 + length xs        -- length.2
--    reverse [] = []                      -- reverse.1
--    reverse (x:xs) = reverse xs ++ [x]   -- reverse.2
-- ---------------------------------------------------------------------

{-
La demostración es por inducción en xs.
```

```
Base: Supongamos que xs = []. Entonces,
   length (reverse xs)
   = length (reverse [])
   = length []                    [por reverse.1]
   = length xs.

Paso de inducción: Supongamos la hipótesis de inducción
   length (reverse xs) = length xs                 (H.I.)
y sea x un elemento cualquiera. Hay que demostrar que
   length (reverse (x:xs)) = length (x:xs)
En efecto,
   length (reverse (x:xs))
   = length (reverse xs ++ [x])          [por reverse.2]
   = length (reverse xs) + length [x]
   = length xs + 1                       [por H.I.]
   = length (x:xs)                       [por length.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por recursión, la función
--    sumaVectores :: [Int] -> [Int] -> [Int]
-- tal que (sumaVectores v w) es la lista obtenida sumando los elementos
-- de v y w que ocupan las mismas posiciones. Por ejemplo,
--    sumaVectores [1,2,5,-6] [0,3,-2,9]  ==  [1,5,3,3]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
sumaVectores :: [Int] -> [Int] -> [Int]
sumaVectores xs ys = [x+y | (x,y) <- zip xs ys]

-- 2ª definición (por recursión):
sumaVectores2 :: [Int] -> [Int] -> [Int]
sumaVectores2 [] _          = []
sumaVectores2 _ []          = []
sumaVectores2 (x:xs) (y:ys) = x+y : sumaVectores2 xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    multPorEscalar :: Int -> [Int] -> [Int]
-- tal que (multPorEscalar x v) es la lista que resulta de multiplicar
```

```
-- todos los elementos de v por x. Por ejemplo,
--    multPorEscalar 4 [1,2,5,-6]  ==  [4,8,20,-24]
-- ---------------------------------------------------------------------

multPorEscalar :: Int -> [Int] -> [Int]
multPorEscalar _ [] = []
multPorEscalar n (x:xs) = n*x : multPorEscalar n xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que las operaciones
-- anteriores verifican la propiedad distributiva de multPorEscalar con
-- respecto a sumaVectores.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_distributiva :: Int -> [Int] -> [Int] -> Bool
prop_distributiva n xs ys =
    multPorEscalar n (sumaVectores xs ys) ==
    sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Probar, por inducción, la propiedad anterior.
-- ---------------------------------------------------------------------

{-
La demostración es por inducción en xs.

Base: Supongamos que xs = []. Entonces,
   multPorEscalar n (sumaVectores xs ys)
   = multPorEscalar n (sumaVectores [] ys)
   = multPorEscalar n []
   = []
   = sumaVectores [] (multPorEscalar n ys)
   = sumaVectores (multPorEscalar n []) (multPorEscalar n ys)
   = sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)
```

```
Paso de inducción: Supongamos la hipótesis de inducción
    multPorEscalar n (sumaVectores xs ys)
    = sumaVectores (multPorEscalar n xs) (multPorEscalar n ys) (H.I. 1)
Hay que demostrar que
    multPorEscalar n (sumaVectores (x:xs) ys)
    = sumaVectores (multPorEscalar n (x:xs)) (multPorEscalar n ys)
Lo haremos por casos en ys.

Caso 1: Supongamos que ys = []. Entonces,
   multPorEscalar n (sumaVectores xs ys)
   = multPorEscalar n (sumaVectores xs [])
   = multPorEscalar n []
   = []
   = sumaVectores (multPorEscalar n xs) []
   = sumaVectores (multPorEscalar n xs) (multPorEscalar n [])
   = sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)

Caso 2: Para (y:ys). Entonces,
    multPorEscalar n (sumaVectores (x:xs) (y:ys))
    = multPorEscalar n (x+y : sumaVectores xs ys)
         [por multPorEscalar.2]
    = n*(x+y) : multPorEscalar n (sumaVectores xs ys)
         [por multPorEscalar.2]
    = n*x+n*y : sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)
         [por H.I. 1]
    = sumaVectores (n*x : multPorEscalar n xs) (n*y : multPorEscalar n ys)
         [por sumaVectores.2]
    = sumaVectores (multPorEscalar n (x:xs)) (multPorEscalar n (y:ys))
         [por multPorEscalar.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideremos los árboles binarios definidos como sigue
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
--
-- Definir la función
--    mapArbol :: (a -> b) -> Arbol a -> Arbol b
-- tal que (mapArbol f x) es el árbol que resulta de sustituir cada nodo
```

```
-- n del árbol x por (f n). Por ejemplo,
--    ghci> mapArbol (+1) (N 9 (N 3 (N 2 H H) (N 4 H H)) (N 7 H H))
--    N 10 (N 8 H H) (N 4 (N 5 H H) (N 3 H H))
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por comprensión, la función
--    mayorExpMenor :: Int -> Int -> Int
-- tal que (mayorExpMenor a b) es el menor n tal que a^n es mayor que
-- b. Por ejemplo,
--    mayorExpMenor 2 1000  ==  10
--    mayorExpMenor 9 7     ==   1
-- ---------------------------------------------------------------------

mayorExpMenor :: Int -> Int -> Int
mayorExpMenor a b =
    head [n | n <- [0..], a^n > b]
```

1.2.3. Examen 3 (5 de julio de 2010)

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

1.2.4. Examen 4 (15 de septiembre de 2010)

El examen es común con el del grupo 1 (ver página 29).

1.2.5. Examen 5 (17 de diciembre de 2010)

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

2

Exámenes del curso 2010–11

2.1. Exámenes del grupo 3 (María J. Hidalgo)

2.1.1. Examen 1 (29 de Octubre de 2010)

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función extremos tal que (extremos n xs) es la
-- lista formada por los n primeros elementos de xs y los n finales
-- elementos de xs. Por ejemplo,
--    extremos 3 [2,6,7,1,2,4,5,8,9,2,3]  ==  [2,6,7,9,2,3]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1.2. Examen 2 (26 de Noviembre de 2010)

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

import Test.QuickCheck

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

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

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

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

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

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

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

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

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

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

e = 2.71828459

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

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

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

```
-- abundante. Por ejemplo,
--    numeroAbundante 5   ==  False
--    numeroAbundante 12  ==  True
--    numeroAbundante 28  ==  False
--    numeroAbundante 30  ==  True
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

2.1.3. Examen 3 (17 de Diciembre de 2010)

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

import Data.List
import Test.QuickCheck

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

-- 1ª definición
grafoReducido1:: (Eq a, Eq b) => (a->b)->(a-> Bool) -> [a] -> [(a,b)]
grafoReducido1 f p xs = nub (map (\x -> (x,f x)) (filter p xs))
```

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

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

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

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

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

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

```
    where yss = sublistas xs

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

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

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

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

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

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

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

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

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

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

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

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

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

```
--    esFuncion r1 == False
--    esFuncion r2 == True
--    esFuncion r3 == True
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir, por composición, la función
```

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

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

2.1.4. Examen 4 (11 de Febrero de 2011)

El examen es común con el del grupo 1 (ver página 81).

2.1.5. Examen 5 (14 de Marzo de 2011)

El examen es común con el del grupo 1 (ver página 83).

2.1.6. Examen 6 (15 de abril de 2011)

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

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

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

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

```
                    | otherwise = aux (resto m) (max k (menor m))

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

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


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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1.  Consideremos los tipos de los vectores y las matrices
-- definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
-- y los siguientes ejemplos
--    p1, p2, p3 :: Matriz Double
--    p1 = listArray ((1,1),(3,3)) [1.0,2,3,1,2,4,1,2,5]
--    p2 = listArray ((1,1),(3,3)) [1.0,2,3,1,3,4,1,2,5]
--    p3 = listArray ((1,1),(3,3)) [1.0,2,1,0,4,7,0,0,5]
--
-- Definir la función
--    esTriangularS :: Num a => Matriz a -> Bool
-- tal que (esTriangularS p) se verifica si p es una matriz triangular
-- superior. Por ejemplo,
--    esTriangularS p1 == False
```

```
--    esTriangularS p3 == True
-- ---------------------------------------------------------------------

type Vector a = Array Int a

type Matriz a = Array (Int,Int) a

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

esTriangularS:: Num a => Matriz a -> Bool
esTriangularS p = and [p!(i,j) == 0 | i <-[1..m], j <-[1..n], i > j]
    where (_,(m,n)) = bounds p

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    determinante:: Matriz Double -> Double
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> determinante (listArray ((1,1),(3,3)) [2,0,0,0,3,0,0,0,1])
--    6.0
--    ghci> determinante (listArray ((1,1),(3,3)) [1..9])
--    0.0
--    ghci> determinante (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
--    -33.0
-- ---------------------------------------------------------------------

determinante:: Matriz Double -> Double
determinante p
    | (m,n) == (1,1) = p!(1,1)
    | otherwise =
        sum [((-1)^(i+1))*(p!(i,1))*determinante (submatriz i 1 p)
             | i <- [1..m]]
    where (_,(m,n)) = bounds p

-- (submatriz i j p) es la submatriz de p obtenida eliminado la fila i y
-- la columna j. Por ejemplo,
--    ghci> submatriz 2 3 (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
```

```
--    array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),5),((2,2),4)]
--    ghci> submatriz 2 3 (listArray ((1,1),(3,3)) [1..9])
--    array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),7),((2,2),8)]
submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
    array ((1,1), (m-1,n -1))
          [((k,l), p ! f k l) | k <- [1..m-1], l <- [1..n-1]]
    where (_,(m,n)) = bounds p
          f k l | k < i  && l < j  = (k,l)
                | k >= i && l < j  = (k+1,l)
                | k < i  && l >= j = (k,l+1)
                | otherwise        = (k+1,l+1)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El número 22940075 tiene una curiosa propiedad. Si lo
-- factorizamos, obtenemos 22940075 = 5^2×229×4007. Reordenando y
-- concatenando los factores primos (5, 229, 40079 podemos obtener el
-- número original: 22940075.
--
-- Diremos que un número es especial si tiene esta propiedad.
--
-- Definir la función
--    esEspecial :: Integer -> Bool
-- tal que (esEspecial n) se verifica si n es un número especial. Por
-- ejemplo,
--    esEspecial 22940075  ==  True
--    esEspecial 22940076  ==  False
-- ---------------------------------------------------------------------

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

-- (factorizacion n) es la lista de los factors de n. Por ejemplo,
--    factorizacion 22940075  ==  [5,5,229,4007]
factorizacion :: Integer -> [Integer]
factorizacion n | n == 1    = []
                | otherwise = x : factorizacion (div n x)
    where x = menorFactor n
```

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

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

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

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

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

2.1.7. Examen 7 (27 de mayo de 2011)

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

import Data.List
import Data.Array
import Test.QuickCheck
-- import GrafoConMatrizDeAdyacencia
import GrafoConVectorDeAdyacencia

-- ---------------------------------------------------------------------
-- Ejercicio 1. En los distintos apartados de este ejercicio
```

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

type RB a = [(a,a)]

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

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

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

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

```
--    reflexiva r1 == True
--    reflexiva r2 == False
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

-- 4ª solución:
transitiva4 :: RB Int -> Bool
transitiva4 = not . noTransitiva

-- (noTransitiva r) se verifica si r no es transitiva; es decir, si
-- existe un (x,y), (y,z) en r tales que (x,z) no está en r.
noTransitiva :: RB Int -> Bool
```

```
noTransitiva r =
    not (null [(x,y,z) | (x,y,z) <- ls,
                         (x,y) `elem` r , (y,z) `elem` r,
                         (x,z) `notElem` r])
    where l = universo r
          ls = [(x,y,z) | x <- l, y <- l, z <- l, x/=y, y /= z]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Consideremos un grafo G = (V,E), donde V es un
-- conjunto finito de nodos ordenados y E es un conjunto de arcos. En un
-- grafo, la anchura de un nodo es el máximo de los valores absolutos de
-- la diferencia entre el valor del nodo y los de sus adyacentes; y la
-- anchura del grafo es la máxima anchura de sus nodos. Por ejemplo, en
-- el grafo
--    g :: Grafo Int Int
--    g = creaGrafo ND (1,5) [(1,2,1),(1,3,1),(1,5,1),
--                            (2,4,1),(2,5,1),
--                            (3,4,1),(3,5,1),
--                            (4,5,1)]
-- su anchura es 4 y el nodo de máxima anchura es el 5.
--
-- Definir la función
--    anchura :: Grafo Int Int -> Int
-- tal que (anchuraG g) es la anchura del grafo g. Por ejemplo,
--    anchura g  ==  4
-- ---------------------------------------------------------------------

g :: Grafo Int Int
g = creaGrafo ND (1,5) [(1,2,1),(1,3,1),(1,5,1),
                        (2,4,1),(2,5,1),
                        (3,4,1),(3,5,1),
                        (4,5,1)]

anchura :: Grafo Int Int -> Int
anchura g = maximum [anchuraN g x | x <- nodos g]

-- (anchuraN g x) es la anchura del nodo x en el grafo g. Por ejemplo,
--    anchuraN g 1  ==  4
--    anchuraN g 2  ==  3
--    anchuraN g 4  ==  2
```

```
--    anchuraN g 5  ==  4
anchuraN :: Grafo Int Int -> Int -> Int
anchuraN g x = maximum (0 : [abs (x-v) | v <- adyacentes g x])

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar experimentalmente que la anchura del grafo
-- grafo cíclico de orden n es n-1.
-- ---------------------------------------------------------------------

-- La conjetura
conjetura :: Int -> Bool
conjetura n = anchura (grafoCiclo n) == n-1

-- (grafoCiclo n) es el grafo cíclico de orden n. Por ejemplo,
--    ghci> grafoCiclo 4
--    G ND (array (1,4) [(1,[(4,0),(2,0)]),(2,[(1,0),(3,0)]),
--                       (3,[(2,0),(4,0)]),(4,[(3,0),(1,0)])])
grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n = creaGrafo ND (1,n) xs
    where xs = [(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)]

-- La comprobación es
--    ghci> and [conjetura n | n <- [2..10]]
--    True

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
```

```
--    prodB :: Bool -> Bool -> Bool
-- tal que (prodB x y) es verdadero si y sólo si ambos argumentos son
-- verdaderos.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. En los siguientes apartados usaremos los tipos
-- definidos a continuación:
-- * Los vectores son tablas cuyos índices son números naturales.
--      type Vector a = Array Int a
-- * Las matrices son tablas cuyos índices son pares de números
--   naturales.
--      type Matriz a = Array (Int,Int) a
-- En los ejemplos se usarán las siguientes matrices:
--    m1, m2 :: Matriz Bool
--    m1 = array ((1,1),(3,3)) [((1,1),True), ((1,2),False),((1,3),True),
--                              ((2,1),False),((2,2),False),((2,3),False),
--                              ((3,1),True), ((3,2),False),((3,3),True)]
--    m2 = array ((1,1),(3,3)) [((1,1),False),((1,2),False),((1,3),True),
--                              ((2,1),False),((2,2),False),((2,3),False),
--                              ((3,1),True), ((3,2),False),((3,3),False)]
--
-- También se usan las siguientes funciones definidas en las relaciones
-- de ejercicios.
--    numFilas :: Matriz a -> Int
--    numFilas = fst . snd . bounds
--
--    numColumnas:: Matriz a -> Int
--    numColumnas = snd . snd . bounds
--
--    filaMat :: Int -> Matriz a -> Vector a
--    filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
--        where n = numColumnas p
--
--    columnaMat :: Int -> Matriz a -> Vector a
--    columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]
--        where m = numFilas p
```

```
--
-- Definir la función
--    prodMatricesB :: Matriz Bool -> Matriz Bool -> Matriz Bool
-- tal que (prodMatricesB p q) es el producto de las matrices booleanas
-- p y q, usando la suma y el producto de booleanos, definidos
-- previamente. Por ejemplo,
--    ghci> prodMatricesB m1 m2
--    array ((1,1),(3,3)) [((1,1),True), ((1,2),False),((1,3),True),
--                         ((2,1),False),((2,2),False),((2,3),False),
--                         ((3,1),True), ((3,2),False),((3,3),True)]
-- ---------------------------------------------------------------------

type Vector a = Array Int a

type Matriz a = Array (Int,Int) a

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

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

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

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

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

prodMatricesB:: Matriz Bool -> Matriz Bool -> Matriz Bool
prodMatricesB p q =
```

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Dada una relación r sobre un conjunto de números
-- naturales mayores que 0, la matriz asociada a r es una matriz
-- booleana p, tal que p_ij = True si y sólo si i está relacionado con j
-- mediante la relación r. Definir la función
--    matrizRB:: RB Int -> Matriz Bool
-- tal que (matrizRB r) es la matriz booleana asociada a r. Por ejemplo,
--    ghci> matrizRB r1
--    array ((1,1),(3,3)) [((1,1),True),((1,2),False),((1,3),True),
```

```
--                          ((2,1),False),((2,2),False),((2,3),False),
--                          ((3,1),True),((3,2),False),((3,3),True)]
--    ghci> matrizRB r2
--    array ((1,1),(3,3)) [((1,1),False),((1,2),False),((1,3),True),
--                          ((2,1),False),((2,2),False),((2,3),False),
--                          ((3,1),True),((3,2),False),((3,3),False)]
--
-- Nota: Construir una matriz booleana cuadrada, de dimensión nxn,
-- siendo n el máximo de los elementos del universo de r.
-- ---------------------------------------------------------------------

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

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

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

2.1.8. Examen 8 (24 de Junio de 2011)

El examen es común con el del grupo 1 (ver página 94).

2.1.9. Examen 9 (8 de Julio de 2011)

El examen es común con el del grupo 1 (ver página 100).

2.1.10. Examen 10 (16 de Septiembre de 2011)

El examen es común con el del grupo 1 (ver página 107).

2.1.11. Examen 11 (22 de Noviembre de 2011)

El examen es común con el del grupo 1 (ver página 117).

2.2. Exámenes del grupo 4 (José A. Alonso y Agustín Riscos)

2.2.1. Examen 1 (25 de Octubre de 2010)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función finales tal que (finales n xs) es la
-- lista formada por los n finales elementos de xs. Por ejemplo,
--    finales 3 [2,5,4,7,9,6]  ==  [7,9,6]
-- ---------------------------------------------------------------------

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

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

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

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

```
--    mediano 2 6 6  ==  6
-- ---------------------------------------------------------------------

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

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

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

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

2.2.2. Examen 2 (22 de Noviembre de 2010)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. El doble factorial de un número n se define por
--    n!! = n*(n-2)* ... * 3 * 1, si n es impar
--    n!! = n*(n-2)* ... * 4 * 2, si n es par
--    1!! = 1
--    0!! = 1
-- Por ejemplo,
--    8!! = 8*6*4*2   = 384
--    9!! = 9*7*5*3*1 = 945
--
-- Definir, por recursión, la función
--    dobleFactorial :: Integer -> Integer
-- tal que (dobleFactorial n) es el doble factorial de n. Por ejemplo,
```

```
--    dobleFactorial 8  ==  384
--    dobleFactorial 9  ==  945
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

2.2.3. Examen 3 (20 de Diciembre de 2010)

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

import Test.QuickCheck

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. El enunciado del problema 1 de la Olimpiada
-- Iberoamericana de Matemática Universitaria del 2006 es el siguiente:
--    Sean m y n números enteros mayores que 1. Se definen los conjuntos
--    P(m) = {1/m, 2/m,..., (m-1)/m} y P(n) = {1/n, 2/n,..., (n-1)/n}.
--    La distancia entre P(m) y P(n) es
--    mín {|a - b| : a en P(m), b en P(n)}.
--
-- Definir la función
--    distancia :: Float -> Float -> Float
-- tal que (distancia m n) es la distancia entre P(m) y P(n). Por
-- ejemplo,
--    distancia 2 7 == 7.142857e-2
--    distancia 2 8 == 0.0
-- ---------------------------------------------------------------------
```

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

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

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

-- Se puede hacer una definición sin búsqueda, ya que por la fórmula de
-- la suma de progresiones aritméticas, la expresión
--    sum [y..y+n-1] == x
-- se reduce a
--    (y+(y+n-1))n/2 = x
-- De donde se puede despejar la y, ya que
--    2yn+n^2-n = 2x
--    y = (2x-n^2+n)/2n
-- De la anterior anterior se obtiene la siguiente definición de
-- consecutivosConSuma que no utiliza búsqueda.

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

```
    | otherwise                  = []
    where z = 2*x-n^2+n
          y = div z (2*n)

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

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

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

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

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

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

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

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

2.2.4. Examen 4 (11 de Febrero de 2011)

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se consideran los árboles binarios
-- definidos por
--    data Arbol = H Int
--               | N Arbol Int Arbol
--               deriving (Show, Eq)
-- Por ejemplo, el árbol
--         5
--        / \
--       /   \
--      9     7
--     / \   / \
--    1   4 6   8
-- se representa por
--    N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8))
-- Definir la función
--    maximoArbol :: Arbol -> Int
-- tal que (maximoArbol a) es el máximo valor en el árbol a. Por
-- ejemplo,
--    maximoArbol (N (N (H 1) 9 (H 4)) 5 (N (H 6) 7 (H 8)))  ==  9
-- ---------------------------------------------------------------------

data Arbol = H Int
```

```
           | N Arbol Int Arbol
           deriving (Show, Eq)

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

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

segmentos _ [] = []
segmentos p xs =
    takeWhile p xs : segmentos p (dropWhile (not.p) (dropWhile p xs))
```

2.2.5. Examen 5 (14 de Marzo de 2011)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Se consideran las funciones
--    duplica  :: [a] -> [a]
--    longitud :: [a] -> Int
-- tales que
--    (duplica xs) es la lista obtenida duplicando los elementos de xs y
--    (longitud xs) es el número de elementos de xs.
-- Por ejemplo,
--    duplica  [7,2,5]  ==  [7,7,2,2,5,5]
--    longitud [7,2,5]  ==  3
--
-- Las definiciones correspondientes son
--    duplica [] = []                    -- duplica.1
--    duplica (x:xs) = x:x:duplica xs    -- duplica.2
--
--    longitud [] = 0                    -- longitud.1
```

```
--    longitud (x:xs) = 1 + longitud xs   -- longitud.2
-- Demostrar por inducción que
--    longitud (duplica xs) = 2 * longitud xs
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

-- fibs es la sucesión de Fibonacci. Por ejemplo,
--    take 10 fibs  ==  [0,1,1,2,3,5,8,13,21,34]
fibs :: [Int]
fibs = 0:1:[x+y | (x,y) <- zip fibs (tail fibs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4 El ejercicio 4 de la Olimpiada Matemáticas de 1993 es el
-- siguiente:
--    Demostrar que para todo número primo p distinto de 2 y de 5,
--    existen infinitos múltiplos de p de la forma 1111......1 (escrito
--    sólo con unos).
--
```

```
-- Definir la función
--    multiplosEspeciales :: Integer -> Int -> [Integer]
-- tal que (multiplosEspeciales p n) es una lista de n múltiplos p de la
-- forma 1111...1 (escrito sólo con unos), donde p es un número primo
-- distinto de 2 y 5. Por ejemplo,
--    multiplosEspeciales 7 2  ==  [111111,111111111111]
-- ---------------------------------------------------------------------

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

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

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

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

2.2.6. Examen 6 (11 de Abril de 2011)

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

import Data.List
import Data.Array
import Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 1. Las expresiones aritméticas pueden representarse usando
-- el siguiente tipo de datos
```

```
--    data Expr = N Int | V Char | S Expr Expr | P Expr Expr
--              deriving Show
-- Por ejemplo, la expresión 2*(a+5) se representa por
--    P (N 2) (S (V 'a') (N 5))
--
-- Definir la función
--    valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- decir, el valor de la expresión donde las variables de x se sustituyen
-- por los valores según se indican en el entorno e). Por ejemplo,
--    ghci> valor (P (N 2) (S (V 'a') (V 'b'))) [('a',2),('b',5)]
--    14
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    ocurrencias :: Ord a => a -> Monticulo a -> Int
-- tal que (ocurrencias x m) es el número de veces que ocurre el
-- elemento x en el montículo m. Por ejemplo,
--    ocurrencias 7 (foldr inserta vacio [6,1,7,8,7,5,7])  ==  3
-- ---------------------------------------------------------------------

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

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

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

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

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

```
-- tal que (digitosDeFactorizacion n) es el conjunto de los dígitos que
-- aparecen en la factorización de n. Por ejemplo,
--    digitosDeFactorizacion (factorial 6)   ==  [1,2,3,4,5]
--    digitosDeFactorizacion (factorial 12)  ==  [0,1,2,3,5,7]
-- Usando la función anterior, calcular la solución del problema.
-- ---------------------------------------------------------------------

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

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

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

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

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

-- (menorFactor n) es el menor factor primo de n. Por ejemplo,
```

```
--    menorFactor 15  ==  3
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

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

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

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

2.2.7. Examen 7 (23 de Mayo de 2011)

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

import Data.Array
import GrafoConVectorDeAdyacencia
import RecorridoEnAnchura

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

```
type Punto = (Double,Double)

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un grafo no dirigido G se dice conexo, si para cualquier
-- par de vértices u y v en G, existe al menos una trayectoria (una
-- sucesión de vértices adyacentes) de u a v.
--
-- Definirla función
--    conexo :: (Ix a, Num p) => Grafo a p -> Bool
-- tal que (conexo g) se verifica si el grafo g es conexo. Por ejemplo,
```

```
--    conexo (creaGrafo False (1,3) [(1,2,0),(3,2,0)])  ==  True
--    conexo (creaGrafo False (1,4) [(1,2,0),(3,4,0)])  ==  False
-- ---------------------------------------------------------------------

conexo :: (Ix a, Num p) => Grafo a p -> Bool
conexo g = length (recorridoEnAnchura i g) == n
    where xs = nodos g
          i  = head xs
          n  = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los tipos de los vectores y de las
-- matrices definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
-- y, como ejemplo, la matriz q definida por
--    q :: Matriz Int
--    q = array ((1,1),(2,2)) [((1,1),1),((1,2),1),((2,1),1),((2,2),0)]
--
-- Definir la función
--    potencia :: Num a => Matriz a -> Int -> Matriz a
-- tal que (potencia p n) es la potencia n-ésima de la matriz cuadrada
-- p. Por ejemplo,
--    ghci> potencia q 2
--    array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),1),((2,2),1)]
--    ghci> potencia q 3
--    array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),2),((2,2),1)]
--    ghci> potencia q 4
--    array ((1,1),(2,2)) [((1,1),5),((1,2),3),((2,1),3),((2,2),2)]
-- ¿Qué relación hay entre las potencias de la matriz q y la sucesión de
-- Fibonacci?
-- ---------------------------------------------------------------------

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

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

potencia :: Num a => Matriz a -> Int -> Matriz a
```

```
potencia p 0 = identidad (numFilas p)
potencia p (n+1) = prodMatrices p (potencia p n)

-- (identidad n) es la matriz identidad de orden n. Por ejemplo,
--    ghci> identidad 3
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
--                         ((2,1),0),((2,2),1),((2,3),0),
--                         ((3,1),0),((3,2),0),((3,3),1)]
identidad :: Num a => Int -> Matriz a
identidad n =
    array ((1,1),(n,n))
          [((i,j),f i j) | i <- [1..n], j <- [1..n]]
    where f i j | i == j    = 1
                | otherwise = 0

-- (prodEscalar v1 v2) es el producto escalar de los vectores v1
-- y v2. Por ejemplo,
--    ghci> prodEscalar (listArray (1,3) [3,2,5]) (listArray (1,3) [4,1,2])
--    24
prodEscalar :: Num a => Vector a -> Vector a -> a
prodEscalar v1 v2 =
    sum [i*j | (i,j) <- zip (elems v1) (elems v2)]

-- (filaMat i p) es el vector correspondiente a la fila i-ésima
-- de la matriz p. Por ejemplo,
--    filaMat 2 q  ==  array (1,2) [(1,1),(2,0)]
filaMat :: Num a => Int -> Matriz a -> Vector a
filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
    where n = numColumnas p

-- (columnaMat j p) es el vector correspondiente a la columna
-- j-ésima de la matriz p. Por ejemplo,
--    columnaMat 2 q  ==  array (1,2) [(1,1),(2,0)]
columnaMat :: Num a => Int -> Matriz a -> Vector a
columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]
    where m = numFilas p

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

```
numFilas = fst . snd . bounds

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

-- (prodMatrices p q) es el producto de las matrices p y q. Por ejemplo,
--    ghci> prodMatrices q q
--    array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),1),((2,2),1)]
prodMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
prodMatrices p q =
    array ((1,1),(m,n))
          [((i,j), prodEscalar (filaMat i p) (columnaMat j q)) |
           i <- [1..m], j <- [1..n]]
    where m = numFilas p
          n = numColumnas q

-- Los sucesión de Fibonacci es 0,1,1,2,3,5,8,13,... Se observa que los
-- elementos de (potencia q n) son los términos de la sucesión en los
-- lugares n+1, n, n y n-1.
```

2.2.8. Examen 8 (24 de Junio de 2011)

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

import Test.QuickCheck
import Data.Array

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se consideran el tipo de las matrices definido por
--    type Matriz a = Array (Int,Int) a
-- y, como ejemplo, la matriz q definida por
--    q :: Matriz Int
--    q = array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),3),((2,2),1)]
--
-- Definir la función
--    indicesMaximo :: (Num a, Ord a) => Matriz a -> [(Int,Int)]
-- tal que (indicesMaximo p) es la lista de los índices del elemento
-- máximo de la matriz p. Por ejemplo,
--    indicesMaximo q  ==  [(1,1),(2,1)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

q :: Matriz Int
q = array ((1,1),(2,2)) [((1,1),3),((1,2),2),((2,1),3),((2,2),1)]

indicesMaximo :: (Num a, Ord a) => Matriz a -> [(Int,Int)]
indicesMaximo p = [(i,j) | (i,j) <- indices p, p!(i,j) == m]
    where m = maximum (elems p)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los montículo se pueden representar mediante el
-- siguiente tipo de dato algebraico.
--    data Monticulo = Vacio
--                   | M Int Monticulo Monticulo
--                   deriving Show
-- Por ejemplo, el montículo
--         1
--        / \
```

```
--       /   \
--      5     4
--     / \   /
--    7   6 8
-- se representa por
--    m1, m2, m3 :: Monticulo
--    m1 = M 1 m2 m3
--    m2 = M 5 (M 7 Vacio Vacio) (M 6 Vacio Vacio)
--    m3 = M 4 (M 8 Vacio Vacio) Vacio
--
-- Definir las funciones
--    ramaDerecha :: Monticulo -> [Int]
--    rango :: Monticulo -> Int
-- tales que (ramaDerecha m) es la rama derecha del montículo m. Por ejemplo,
--    ramaDerecha m1  ==  [1,4]
--    ramaDerecha m2  ==  [5,6]
--    ramaDerecha m3  ==  [4]
-- y (rango m) es el rango del montículo m; es decir, la menor distancia
-- desde la raíz de m a un montículo vacío. Por ejemplo,
--    rango m1        ==  2
--    rango m2        ==  2
--    rango m3        ==  1
-- ---------------------------------------------------------------------

data Monticulo = Vacio
               | M Int Monticulo Monticulo
               deriving Show

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.9. Examen 9 (8 de Julio de 2011)

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

import Data.Array
```

```
import Data.Char
import Test.QuickCheck
import GrafoConVectorDeAdyacencia

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

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

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

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

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

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

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

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

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

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

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

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

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

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

type Matriz a = Array (Int,Int) a

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

esMatrizPermutacion :: (Num a, Eq a) => Matriz a -> Bool
```

```
esMatrizPermutacion p =
    and [esListaUnitaria [p!(i,j) | i <- [1..n]] | j <- [1..n]] &&
    and [esListaUnitaria [p!(i,j) | j <- [1..n]] | i <- [1..n]]
    where ((1,1),(n,_)) = bounds p

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

-- ---------------------------------------------------------------------
--  Ejercicio 4. Un mapa se puede representar mediante un grafo donde
--  los vértices son las regiones del mapa y hay una arista entre dos
--  vértices si las correspondientes regiones son vecinas. Por ejemplo,
--  el mapa siguiente
--        +----------+----------+
--        |    1     |     2    |
--        +----+-----+-----+----+
--        |    |           |    |
--        | 3  |     4     | 5  |
--        |    |           |    |
--        +----+-----+-----+----+
--        |    6     |     7    |
--        +----------+----------+
-- se pueden representar por
--    mapa :: Grafo Int Int
--    mapa = creaGrafo False (1,7)
--                       [(1,2,0),(1,3,0),(1,4,0),(2,4,0),(2,5,0),(3,4,0),
--                        (3,6,0),(4,5,0),(4,6,0),(4,7,0),(5,7,0),(6,7,0)]
-- Para colorear el mapa se dispone de 4 colores definidos por
--    data Color = A | B | C | D deriving (Eq, Show)
--
-- Definir la función
--    correcta :: [(Int,Color)] -> Grafo Int Int -> Bool
-- tal que (correcta ncs m) se verifica si ncs es una coloración del
-- mapa m tal que todos las regiones vecinas tienen colores distintos.
```

```
-- Por ejemplo,
--    correcta [(1,A),(2,B),(3,B),(4,C),(5,A),(6,A),(7,B)] mapa == True
--    correcta [(1,A),(2,B),(3,A),(4,C),(5,A),(6,A),(7,B)] mapa == False
-- ---------------------------------------------------------------------

mapa :: Grafo Int Int
mapa = creaGrafo ND (1,7)
                 [(1,2,0),(1,3,0),(1,4,0),(2,4,0),(2,5,0),(3,4,0),
                  (3,6,0),(4,5,0),(4,6,0),(4,7,0),(5,7,0),(6,7,0)]

data Color = A | B | C | D deriving (Eq, Show)

correcta :: [(Int,Color)] -> Grafo Int Int -> Bool
correcta ncs g =
    and [and [color x /= color y | y <- adyacentes g x] | x <- nodos g]
    where color x = head [c | (y,c) <- ncs, y == x]

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. La expansión decimal de un número racional puede
-- representarse mediante una lista cuyo primer elemento es la parte
-- entera y el resto está formado por los dígitos de su parte decimal.
--
-- Definir la función
--    expansionDec :: Integer -> Integer -> [Integer]
-- tal que (expansionDec x y) es la expansión decimal de x/y. Por
-- ejemplo,
--    take 10 (expansionDec 1 4)    ==  [0,2,5]
--    take 10 (expansionDec 1 7)    ==  [0,1,4,2,8,5,7,1,4,2]
--    take 12 (expansionDec 90 7)   ==  [12,8,5,7,1,4,2,8,5,7,1,4]
--    take 12 (expansionDec 23 14)  ==  [1,6,4,2,8,5,7,1,4,2,8,5]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. La parte decimal de las expansiones decimales se puede
```

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

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

-- (decimales xs) es el par formado por la parte decimal pura y la parte
-- decimal periódica de la lista de decimales xs. Por ejemplo,
--    decimales [3,1,4]            ==  ([3,1,4],[])
--    decimales [3,1,6,7,5,6,7,5]  ==  ([3,1],[6,7,5])
decimales :: [Integer] -> ([Integer],[Integer])
decimales xs = decimales' xs []
    where decimales' [] ys = (reverse ys, [])
          decimales' (x:xs) ys
              | x `elem` ys = splitAt k ys'
              | otherwise   = decimales' xs (x:ys)
              where ys' = reverse ys
                    k   = posicion x ys'

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

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

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

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

2.2.10. Examen 10 (16 de Septiembre de 2011)

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

import Data.List
import Test.QuickCheck

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El problema del laberinto numérico consiste en, dados
-- un par de números, encontrar la longitud del camino más corto entre
-- ellos usando sólo las siguientes operaciones:
--    * multiplicar por 2,
--    * dividir por 2 (sólo para los pares) y
--    * sumar 2.
-- Por ejemplo,
--    longitudCaminoMinimo 3 12  ==  2
--    longitudCaminoMinimo 12 3  ==  2
--    longitudCaminoMinimo 9 2   ==  8
--    longitudCaminoMinimo 2 9   ==  5
-- Unos caminos mínimos correspondientes a los ejemplos anteriores son
-- [3,6,12], [12,6,3], [9,18,20,10,12,6,8,4,2] y [2,4,8,16,18,9].
--
-- Definir la función
--    orbita :: Int -> [Int] -> [Int]
-- tal que (orbita n xs) es el conjunto de números que se pueden obtener
-- aplicando como máximo n veces las operaciones a los elementos de
-- xs. Por ejemplo,
--    orbita 0 [12]  ==  [12]
--    orbita 1 [12]  ==  [6,12,14,24]
--    orbita 2 [12]  ==  [3,6,7,8,12,14,16,24,26,28,48]
-- ---------------------------------------------------------------------

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

```
                    | otherwise = [2*x, x `div` 2, x+2]

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. En este ejercicio se estudia las relaciones entre los
-- valores de polinomios y los de sus correspondientes expresiones
-- aritméticas.
--
-- Las expresiones aritméticas construidas con una variables, los
-- números enteros y las operaciones de sumar y multiplicar se pueden
-- representar mediante el tipo de datos Exp definido por
--    data Exp = Var | Const Int | Sum Exp Exp | Mul Exp  Exp
--               deriving Show
-- Por ejemplo, la expresión 3+5x^2 se puede representar por
--    exp1 :: Exp
--    exp1 = Sum (Const 3) (Mul Var (Mul Var (Const 5)))
--
-- Definir la función
--    valorE :: Exp -> Int -> Int
-- tal que (valorE e n) es el valor de la expresión e cuando se
-- sustituye su variable por n. Por ejemplo,
--    valorE exp1 2  ==  23
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

2.2.11. Examen 11 (22 de Noviembre de 2011)

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

import Data.Array
import Test.QuickCheck

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    barajaR :: [a] -> [a] -> [a]
-- tal que (barajaR xs ys) es la lista obtenida intercalando los
-- elementos de las listas xs e ys. Por ejemplo,
```

```
--    barajaR [1,6,2] [3,7]        ==  [1,3,6,7]
--    barajaR [1,6,2] [3,7,4,9,0]  ==  [1,3,6,7,2,4]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

```
--    divide 0 0  ==  True
divide :: Int -> Int -> Bool
divide 0 y = y == 0
divide x y = y `rem` x == 0

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

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

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

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

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

prop_refactorizable :: Int -> Bool
```

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un árbol binario de búsqueda (ABB) es un árbol binario
-- tal que el de cada nodo es mayor que los valores de su subárbol
-- izquierdo y es menor que los valores de su subárbol derecho y,
-- además, ambos subárboles son árboles binarios de búsqueda. Por
-- ejemplo, al almacenar los valores de [8,4,2,6,3] en un ABB se puede
-- obtener el siguiente ABB:
--
--       5
--      / \
--     /   \
--    2     6
--         / \
--        4   8
--
-- Los ABB se pueden representar como tipo de dato algebraico:
--    data ABB = V
--             | N Int ABB ABB
--             deriving (Eq, Show)
-- Por ejemplo, la definición del ABB anteriore es
--    ej :: ABB
--    ej = N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
-- Definir la función
```

```
--    inserta :: Int -> ABB -> ABB
-- tal que (inserta v a) es el árbol obtenido añadiendo el valor v al
-- ABB a, si no es uno de sus valores. Por ejemplo,
--    ghci>  inserta 5 ej
--    N 3 (N 2 V V) (N 6 (N 4 V (N 5 V V)) (N 8 V V))
--    ghci>  inserta 1 ej
--    N 3 (N 2 (N 1 V V) V) (N 6 (N 4 V V) (N 8 V V))
--    ghci>  inserta 2 ej
--    N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los tipos de los vectores y de las
-- matrices definidos por
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
--
-- Definir la función
--    antidiagonal :: (Num a, Eq a) => Matriz a -> Bool
-- tal que (antidiagonal m) se verifica si es cuadrada y todos los
-- elementos de m que no están en su diagonal secundaria son nulos. Por
-- ejemplo,  si m1 y m2 son las matrices definidas por
--    m1, m2 :: Matriz Int
--    m1 = array ((1,1),(3,3)) [((1,1),7),((1,2),0),((1,3),4),
--                              ((2,1),0),((2,2),6),((2,3),0),
--                              ((3,1),0),((3,2),0),((3,3),5)]
```

```
--    m2 = array ((1,1),(3,3)) [((1,1),0),((1,2),0),((1,3),4),
--                              ((2,1),0),((2,2),6),((2,3),0),
--                              ((3,1),0),((3,2),0),((3,3),0)]
-- entonces
--    antidiagonal m1  ==  False
--    antidiagonal m2  ==  True
-- ---------------------------------------------------------------------

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

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

antidiagonal :: (Num a, Eq a) => Matriz a -> Bool
antidiagonal p =
    m == n && nula [p!(i,j) | i <- [1..n], j <- [1..n], j /= n+1-i]
    where (m,n) = snd (bounds p)

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

3

Exámenes del curso 2011–12

3.1. Exámenes del grupo 1 (José A. Alonso y Agustín Riscos)

3.1.1. Examen 1 (26 de Octubre de 2011)

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

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

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

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

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

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

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

interior xs = tail (init xs)
```

3.1.2. Examen 2 (30 de Noviembre de 2011)

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. En este ejercicio vamos a comprobar que la ecuación
```

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

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

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

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

3.1.3. Examen 3 (25 de Enero de 2012)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [2 puntos] Un número es muy compuesto si tiene más
-- divisores que sus anteriores. Por ejemplo, 12 es muy compuesto porque
-- tiene 6 divisores (1, 2, 3, 4, 6, 12) y todos los números del 1 al 11
```

```
-- tienen menos de 6 divisores.
--
-- Definir la función
--    esMuyCompuesto :: Int -> Bool
-- tal que (esMuyCompuesto x) se verifica si x es un número muy
-- compuesto. Por ejemplo,
--    esMuyCompuesto 24  ==  True
--    esMuyCompuesto 25  ==  False
-- Calcular  el menor número muy compuesto de 4 cifras.
-- ---------------------------------------------------------------------

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. [1 punto] Definir la función
--    muyCompuesto :: Int -> Int
-- tal que (muyCompuesto n) es el n-ésimo número muy compuesto. Por
-- ejemplo,
--    muyCompuesto 10  ==  180
```

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. [1.5 puntos] Definir la función
--    sumaPrimosTruncables :: Int -> Int
-- tal que (sumaPrimosTruncables n) es la suma de los n primeros primos
```

```
-- truncables. Por ejemplo,
--    sumaPrimosTruncables 10  ==  249
-- Calcular la suma de los 20 primos truncables.
-- ---------------------------------------------------------------------

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. [1.5 puntos] Definir la función
--    posicion :: Int -> Int
-- tal que (posicion x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
--    posicion 2  ==  4
-- ---------------------------------------------------------------------
```

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

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

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

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

3.1.4. Examen 4 (29 de Febrero de 2012)

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

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

primitivo :: Integer -> Integer
```

```
primitivo n | n < 10    = n
            | otherwise = primitivo (producto n)

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Los árboles binarios se pueden representar
-- mediante el siguiente tipo de datos
--    data Arbol = H
--               | N Int Arbol Arbol
-- Por ejemplo, el árbol
--            9
--           / \
--          /   \
--         3     7
```

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

data Arbol = H
           | N Int Arbol Arbol

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

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

primoPermutable :: Integer -> Bool
primoPermutable x = and [primo y | y <- permutacionesN x]

-- (permutacionesN x) es la lista de los números obtenidos permutando
```

```
-- las cifras de x. Por ejemplo,
permutacionesN :: Integer -> [Integer]
permutacionesN x = [read ys | ys <- permutaciones (show x)]

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

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

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

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

3.1.5. Examen 5 (21 de Marzo de 2012)

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

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

```
-- tal que (equivalentes x y) se verifica si los números x e y son
-- equivalentes. Por ejemplo,
--    equivalentes 3205 41  ==  True
--    equivalentes 3205 25  ==  False
-- ---------------------------------------------------------------------

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Definir la función
--    primosEquivalentes :: Int -> [[Int]]
```

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

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

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

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

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

3.1.6. Examen 6 (2 de Mayo de 2012)

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

```
import Data.Array
import Data.List

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

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

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

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

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

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

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

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

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

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

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

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

sucGolomb :: [Int]
sucGolomb = subSucGolomb 1

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

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

-- Nota: La sucesión de Golomb puede definirse de forma más compacta
```

```
-- como se muestra a continuación.
sucGolomb' :: [Int]
sucGolomb' = 1 : 2 : 2 : g 3
    where g x      = replicate (golomb x) x ++ g (x+1)
          golomb n = sucGolomb !! (n-1)
```

3.1.7. Examen 7 (25 de Junio de 2012)

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    superpar :: Int -> Bool
-- tal que (superpar n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
--    superpar 426  ==  True
--    superpar 456  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión)
superpar :: Int -> Bool
```

```
superpar n | n < 10    = even n
           | otherwise = even n && superpar (n `div` 10)

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    potenciaFunc :: Int -> (a -> a) -> a -> a
-- tal que (potenciaFunc n f x) es el resultado de aplicar n veces la
-- función f a x. Por ejemplo,
--    potenciaFunc 3 (*10) 5  ==  5000
--    potenciaFunc 4 (+10) 5  ==  45
-- ---------------------------------------------------------------------

potenciaFunc :: Int -> (a -> a) -> a -> a
potenciaFunc 0 _ x = x
```

```
potenciaFunc n f x = potenciaFunc (n-1) f (f x)

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


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

data Expr = Num Int
          | Suma Expr Expr
          | X

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] Cuentan que Alan Turing tenía una bicicleta
-- vieja, que tenía una cadena con un eslabón débil y además uno de los
-- radios de la rueda estaba doblado. Cuando el radio doblado coincidía
-- con el eslabón débil, entonces la cadena se rompía.
--
```

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

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

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

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

3.1.8. Examen 8 (29 de Junio de 2012)

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

```
import Data.List
import Data.Array

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    sumaDeDos :: Int -> [Int] -> Maybe (Int,Int)
-- tal que (sumaDeDos x ys) decide si x puede expresarse como suma de
-- dos elementos de ys y, en su caso, devuelve un par de elementos de ys
-- cuya suma es x. Por ejemplo,
--    sumaDeDos 9 [7,4,6,2,5]  ==  Just (7,2)
--    sumaDeDos 5 [7,4,6,2,5]  ==  Nothing
-- ---------------------------------------------------------------------
```

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] La expresiones aritméticas se pueden
-- representar mediante el siguiente tipo
--    data Expr = V Char
```

```
--              | N Int
--              | S Expr Expr
--              | P Expr Expr
--              deriving Show
-- por ejemplo, representa la expresión "z*(3+x)" se representa por
-- (P (V 'z') (S (N 3) (V 'x'))).
--
-- Definir la función
--    sustitucion :: Expr -> [(Char, Int)] -> Expr
-- tal que (sustitucion e s) es la expresión obtenida sustituyendo las
-- variables de la expresión e según se indica en la sustitución s. Por
-- ejemplo,
--    ghci> sustitucion (P (V 'z') (S (N 3) (V 'x'))) [('x',7),('z',9)]
--    P (N 9) (S (N 3) (N 7))
--    ghci> sustitucion (P (V 'z') (S (N 3) (V 'y'))) [('x',7),('z',9)]
--    P (N 9) (S (N 3) (V 'y'))
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] (Problema 345 del proyecto Euler) Las
-- matrices puede representarse mediante tablas cuyos índices son pares
-- de números naturales:
--    type Matriz = Array (Int,Int) Int
-- Definir la función
--    maximaSuma :: Matriz -> Int
-- tal que (maximaSuma p) es el máximo de las sumas de las listas de
```

```
-- elementos de la matriz p tales que cada elemento pertenece sólo a una
-- fila y a una columna. Por ejemplo,
--    ghci> maximaSuma (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
--    17
-- ya que las selecciones, y sus sumas, de la matriz
--    |1 2 3|
--    |8 4 9|
--    |5 6 7|
-- son
--    [1,4,7] --> 12
--    [1,9,6] --> 16
--    [2,8,7] --> 17
--    [2,9,5] --> 16
--    [3,8,6] --> 17
--    [3,4,5] --> 12
-- Hay dos selecciones con máxima suma: [2,8,7] y [3,8,6].
-- ------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

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

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

-- Nota: En la anterior definición se ha usado la función pernutations
-- de Data.List. También se puede definir mediante
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat [intercala x ys | ys <- permutaciones xs]
```

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

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

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

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

3.1.9. Examen 9 (9 de Septiembre de 2012)

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

import Data.Array
```

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

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

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

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

-- La 2ª definición es más eficiente
--    ghci> :set +s
--    ghci> sucesionesConSuma 500
--    [(8,32),(59,66),(98,102),(500,500)]
--    (1.47 secs, 1452551760 bytes)
--    ghci> sucesionesConSuma2 500
--    [(8,32),(59,66),(98,102),(500,500)]
--    (0.31 secs, 31791148 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2 [1.7 puntos] Definir la función
```

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

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

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

paso :: [Int] -> [Int]
paso [] = []
paso [x] = [x]
paso (x:y:zs) | x == -y   = paso zs
              | otherwise = x : paso (y:zs)

reducida :: [Int] -> [Int]
reducida xs | xs == ys  = xs
```

```
            | otherwise = reducida ys
            where ys = paso xs

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. [1.6 puntos] Un árbol ordenado es un árbol binario tal
-- que para cada nodo, los elementos de su subárbol izquierdo son
-- menores y los de su subárbol derecho son mayores. Por ejemplo,
--         5
--        / \
```

```
--       /   \
--      3     7
--     / \   / \
--    1   4 6   9
-- El tipo de los árboles binarios se define por
--    data Arbol = H Int
--               | N Int Arbol Arbol
-- con lo que el ejemplo anterior se define por
--    ejArbol = N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9))
-- Definir la función
--    ancestroMasProximo :: Int -> Int -> Int
-- tal que (ancestroMasProximo x y a) es el ancestro más próximo de los
-- nodos x e y en el árbol a. Por ejemplo,
--    ancestroMasProximo 4 1 ejArbol  == 3
--    ancestroMasProximo 4 6 ejArbol  == 5
-- ---------------------------------------------------------------------

data Arbol = H Int
           | N Int Arbol Arbol

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. [1.6 puntos] Las matrices puede representarse mediante
-- tablas cuyos índices son pares de números naturales:
--    type Matriz = Array (Int,Int) Int
-- Definir la función
--    maximos :: Matriz -> [Int]
-- tal que (maximos p) es la lista de los máximos locales de la matriz
-- p; es decir de los elementos de p que son mayores que todos sus
-- vecinos. Por ejemplo,
--    ghci> maximos (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,0,2,5,4])
--    [9,7]
```

```
-- ya que los máximos locales de la matriz
--    |9 4 6 5|
--    |8 1 7 3|
--    |0 2 5 4|
-- son 9 y 7.
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

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

3.1.10. Examen 10 (10 de Diciembre de 2012)

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

import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir una función verificanP
--    verificanP :: Int -> Integer -> (Integer -> Bool) -> Bool
-- tal que (verificanP k n p) se cumple si los primeros k dígitos del
-- número n verifican la propiedad p y el (k+1)-ésimo no la verifica.
-- Por ejemplo,
--   verificanP 3 224119 even == True
--   verificanP 3 265119 even == False
--   verificanP 3 224619 even == False
-- ---------------------------------------------------------------------

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

verificanP :: Int -> Integer -> (Integer -> Bool) -> Bool
```

```
verificanP k n p = length (takeWhile p (digitos n)) == k

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideremos el tipo de los árboles binarios definido
-- por
--    data Arbol = H Int
```

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las matrices puede representarse mediante tablas cuyos
-- índices son pares de números naturales:
--    type Matriz = Array (Int,Int) Int
-- Definir la función
--    algunMenor :: Matriz -> [Int]
-- tal que (algunMenor p) es la lista de los elementos de p que tienen
```

```
-- algún vecino menor que él. Por ejemplo,
--    algunMenor (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,4,2,5,4])
--    [9,4,6,5,8,7,4,2,5,4]
-- pues sólo el 1 y el 3 no tienen ningún vecino menor en la matriz
--    |9 4 6 5|
--    |8 1 7 3|
--    |4 2 5 4|
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

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

3.2. Exámenes del grupo 2 (María J. Hidalgo)

3.2.1. Examen 1 (27 de Octubre de 2011)

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función pCardinales tal que
-- (pCardinales (x,y) d) es la lista formada por los cuatro puntos
-- situados al norte, sur este y oeste, a una distancia d de (x,y).
-- Por ejemplo,
--    pCardinales (0,0) 2  == [(0,2),(0,-2),(-2,0),(2,0)]
--    pCardinales (-1,3) 4 == [(-1,7),(-1,-1),(-5,3),(3,3)]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

3.2.2. Examen 2 (1 de Diciembre de 2011)

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

import Test.QuickCheck
import Data.List

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

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

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

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

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

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

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

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

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

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

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

```
prop_intercalaCero xs =
    length (intercalaCero xs) >= length xs

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. (Problema 358 del proyecto Euler) Un número x con n
-- cifras se denomina número circular si tiene la siguiente propiedad:
-- si se multiplica por 1, 2, 3, 4, ..., n, todos los números que
-- resultan tienen exactamente las mismas cifras que x, pero en distinto
-- orden. Por ejemplo, el número 142857 es circular, ya que
--    142857 * 1 = 142857
--    142857 * 2 = 285714
--    142857 * 3 = 428571
--    142857 * 4 = 571428
--    142857 * 5 = 714285
--    142857 * 6 = 857142
--
-- Definir la función
--    esCircular :: Int -> Bool
-- tal que (esCircular x) se verifica si x es circular. Por ejemplo,
--    esCircular 142857 == True
--    esCircular 14285  == False
-- ---------------------------------------------------------------------

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

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

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

-- (esPermutacion xs ys) se verifica si xs es una permutación de ys. Por
-- ejemplo,
```

```
--    esPermutacion [2,5,3] [3,5,2]    ==  True
--    esPermutacion [2,5,3] [2,3,5,2]  ==  False
esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     [] = True
esPermutacion []     _  = False
esPermutacion (x:xs) ys = elem x ys && esPermutacion xs (delete x ys)

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

3.2.3. Examen 3 (26 de Enero de 2012)

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

import Test.QuickCheck
import Data.List

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

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

-- 2ª definición (por recursión):
sumatorioR :: (Integer -> Integer) -> Integer -> Integer -> Integer
sumatorioR f m n = aux m 0
```

```
    where aux k ac | k > n     = ac
                   | otherwise = aux (k+1) (ac + f k)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    sumaPred :: Num a => (a -> Bool) -> [a] -> a
-- tal que (sumaPred p xs) es la suma de los elementos de xs que
-- verifican el predicado p. Por ejemplo:
--    sumaPred even [1..1000]   == 250500
--    sumaPred even [1..100000] == 2500050000
-- ---------------------------------------------------------------------

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

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

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

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Representamos una relación binaria sobre un conjunto
-- como un par formado por:
--   * una lista, que representa al conjunto, y
--   * una lista de pares, que forman la relación
-- En los ejemplos usaremos las siguientes relaciones
--    r1, r2, r3,r4 :: ([Int],[(Int, Int)])
--    r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
--    r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
--    r3 = ([1..9],[(1,3), (2,6), (6,2), (3,1), (4,4)])
--    r4 = ([1..3],[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)])
--
-- Definir la función
--    reflexiva :: Eq a => ([a],[(a,a)]) -> Bool
-- tal que (reflexiva r) se verifica si r es una relación reflexiva. Por
-- ejemplo,
--    reflexiva r1 == False
--    reflexiva r4 == True
-- ---------------------------------------------------------------------

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

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

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

-- 1ª definición (por comprensión):
simetricaC :: Eq a => ([a],[(a,a)]) -> Bool
```

```
simetricaC (x,r) =
    null [(x,y) | (x,y) <- r, (y,x) `notElem` r]

-- 2ª definición (por recursión):
simetrica :: Eq a => ([a],[(a,a)]) -> Bool
simetrica r = aux (snd r)
    where aux [] = True
          aux ((x,y):s) | x == y    = aux s
                        | otherwise = elem (y,x) s && aux (delete (y,x) s)

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

-- 1ª solución
-- ===========

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

```
-- (divisoresPrimos n) es la lista de los divisores primos de x. Por
-- ejemplo,
--    divisoresPrimos 180  ==  [2,3,5]
divisoresPrimos :: Int -> [Int]
divisoresPrimos n = [x | x <- [1..n], rem n x == 0, esPrimo x]

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

-- 2ª solución:
-- ============

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Dado un número n, calculamos la suma de sus divisores
-- propios reiteradamente hasta que quede un número primo. Por ejemplo,
--
--    n  | divisores propios                    | suma de div. propios
```

```
-- ------+--------------------------------------+---------------------
--   30  | [1,2,3,5,6,10,15]                    |  42
--   42  | [1,2,3,6,7,14,21]                    |  54
--   54  | [1,2,3,6,9,18,27]                    |  66
--   66  | [1,2,3,6,11,22,33]                   |  78
--   78  | [1,2,3,6,13,26,39]                   |  90
--   90  | [1,2,3,5,6,9,10,15,18,30,45]         | 144
--  144  | [1,2,3,4,6,8,9,12,16,18,24,36,48,72] | 259
--  259  | [1,7,37]                             |  45
--   45  | [1,3,5,9,15]                         |  33
--   33  | [1,3,11]                             |  15
--   15  | [1,3,5]                              |   9
--    9  | [1,3]                                |   4
--    4  | [1,2]                                |   3
--    3 (es primo)
--
-- Definir una función
--    sumaDivReiterada :: Int -> Int
-- tal que (sumaDivReiterada n) calcule reiteradamente la suma de los
-- divisores propios hasta que se llegue a un número primo. Por ejemplo,
--    sumaDivReiterada 30   == 3
--    sumaDivReiterada 52   == 3
--    sumaDivReiterada 5289 == 43
--    sumaDivReiterada 1024 == 7
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. ¿Hay números naturales para los que la función
-- anterior no termina? Si crees que los hay, explica por qué y
```

```
-- encuentra los tres primeros  números para los que la función anterior
-- no terminaría. En caso contrario, justifica por qué termina siempre.
-- ..............................................................

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

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

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

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

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

3.2.4. Examen 4 (1 de Marzo de 2012)

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

import Test.QuickCheck
import Data.List

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

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

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

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

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

data Arbol = H Int
           | N Arbol Int Arbol
```

```
           deriving (Show, Eq)

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

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

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

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

```
-- Para este ejercicio, decimos que un número poligonal de razón n es
-- un número del tipo
--    1 + (1+n) + (1+2*n)+...
-- Es decir, los números triángulares son números poligonales de razón
-- 1, los números cuadrados son números poligonales de razón 2, los
-- pentagonales de razón 3, etc.
--
-- Definir la constante
--    triangulares :: [Integer]
-- tal que es la lista de todos los números triángulares. Por ejemplo,
--    ghci> take 20 triangulares
--    [1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171,190,210]
-- ---------------------------------------------------------------------

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

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

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

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Definir la función
--    esPoligonalN :: Integer -> Integer -> Bool
-- tal que (esPoligonalN x n) se verifica si x es poligonal de razón n.
-- Por ejemplo,
--    esPoligonalN 12 3  ==  True
--    esPoligonalN 12 1  ==  False
-- ---------------------------------------------------------------------

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.7. Definir la función
--    descomposicionTriangular :: Integer -> (Integer, Integer, Integer)
-- tal que que (descomposicionTriangular n) es la descomposición de un
-- número natural en la suma de, a lo sumo 3 números triángulares. Por
```

```
-- ejemplo,
--    descomposicionTriangular 20  == (0,10,10)
--    descomposicionTriangular 206 == (1,15,190)
--    descomposicionTriangular 6   == (0,0,6)
--    descomposicionTriangular 679 == (1,300,378)
-- ---------------------------------------------------------------------

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

3.2.5. Examen 5 (22 de Marzo de 2012)

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

import Test.QuickCheck
import Data.List
import PolOperaciones

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recurción, la función
--    interseccionR :: Eq a => [[a]] -> [a]
```

```
-- tal que (interseccionR xss) es la lista con los elementos comunes a
-- todas las listas de xss. Por ejemplo,
--    interseccionR [[1,2],[3]] == []
--    interseccionR [[1,2],[3,2], [2,4,5,6,1]] == [2]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

```
--    ejPol = consPol 3 1
--                    (consPol 2 3
--                             (consPol 1 (-1)
--                                      (consPol 0 (-2) polCero)))
--
-- Definir la función
--    integral :: Polinomio Float -> Polinomio Float
-- tal que (integral p) es la integral del polinomio p. Por ejemplo,
--    integral ejPol == 0.25*x^4 + x^3 + -0.5*x^2 -2.0*x
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. El método de la bisección para calcular un cero de una
-- función en el intervalo [a,b] se basa en el teorema de Bolzano:
--    "Si f(x) es una función continua en el intervalo [a, b], y si,
```

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

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

3.2.6. Examen 6 (3 de Mayo de 2012)

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

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

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

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

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

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

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

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
--    verificaET :: (a -> Bool) -> [[a]] -> Bool
-- tal que (verificaET p xss) se cumple si todos los elementos de alguna
-- lista de xss verifican el predicado p. Por ejemplo,
--    verificaET odd [[1,3], [4,8], [10]] == True
--    verificaET odd [[4,2], [4,8], [10]] == False
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    raicesApol :: Fractional t => [(t, Int)] -> Polinomio t
-- tal que (raicesApol rs) es el polinomio correspondiente si rs es la
-- lista de raices, con sus respectivas multiplicidades, rs; es decir,
```

```
-- raicesApol [(r1,n1),...,(rk,nk)] es (x-r1)^n1...(x-rk)^nk. Por
-- ejemplo,
--   raicesApol [(2,1),(-1,3)] == x^4 + x^3 + -3.0*x^2 + -5.0*x + -2.0
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

-- (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
```

```
--    ghci> diagonalPral (listArray ((1,1),(3,3)) [5,0,0,1,6,0,0,2,4])
--    array (1,3) [(1,5),(2,6),(3,4)]
diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    determinante:: Matriz Double -> Double
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> determinante (listArray ((1,1),(3,3)) [2,0,0,0,3,0,0,0,1])
--    6.0
--    ghci> determinante (listArray ((1,1),(3,3)) [1..9])
--    0.0
--    ghci> determinante (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
--    -33.0
-- ---------------------------------------------------------------------

determinante:: Matriz Double -> Double
determinante p
    | dimension p == (1,1) = p!(1,1)
    | otherwise =
        sum [((-1)^(i+1))*(p!(i,1))*determinante (submatriz i 1 p)
             | i <- [1..numFilas p]]

-- (dimension p) es la dimensión de la matriz p. Por ejemplo,
--    dimension (listArray ((1,1),(2,3)) [5,0,0,1,6,0])  ==  (2,3)
dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas p)
```

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

3.2.7. Examen 7 (24 de Junio de 2012)

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

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

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

```
--    duplicaElemento 7 [2,7,3,7,7,5]  ==  [2,7,7,3,7,7,7,7,5]
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    menorP :: (Int -> Bool) -> Int
-- tal que (menorP p) es el menor número natural que verifica el
-- predicado p. Por ejemplo,
--    menorP (>7)   ==  8
-- ---------------------------------------------------------------------

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    polNumero :: Int -> Polinomio Int
```

```
-- tal que (polNumero n) es el polinomio cuyos coeficientes son las
-- cifras de n. Por ejemplo,
--   polNumero 5703 == 5x^3 + 7x^2 + 3
-- ---------------------------------------------------------------------

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

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

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

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

type Vector = Array Int Float

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Las matrices se definen por
--    type Matriz = Array (Int,Int) Float
--
-- Una matriz se demonina estocástica si sus columnas son vectores
-- estocásticos.
--
-- Definir la función
--    matrizEstocastica :: Matriz -> Bool
-- tal que (matrizEstocastico p) se verifica si p es estocástica. Por
-- ejemplo,
--    matrizEstocastica (listArray ((1,1),(2,2)) [0.1,0.2,0.9,0.8]) == True
--    matrizEstocastica (listArray ((1,1),(2,2)) [0.1,0.2,0.3,0.8]) == False
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Float

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. Consideremos un grafo G = (V,E), donde V es un conjunto
-- finito de nodos ordenados y E es un conjunto de arcos. En un grafo,
-- la anchura de un nodo es el número de nodos adyacentes; y la anchura
-- del grafo es la máxima anchura de sus nodos. Por ejemplo, en el grafo
--    g :: Grafo Int Int
```

```
--    g = creaGrafo ND (1,5) [(1,2,1),(1,3,1),(1,5,1),
--                            (2,4,1),(2,5,1),
--                            (3,4,1),(3,5,1),
--                            (4,5,1)]
-- su anchura es 4 y el nodo de máxima anchura es el 5.
--
-- Definir la función
--    anchura :: Grafo Int Int -> Int
-- tal que (anchuraG g) es la anchura del grafo g. Por ejemplo,
--    anchura g  ==  4
-- ---------------------------------------------------------------------

g :: Grafo Int Int
g = creaGrafo ND (1,5) [(1,2,1),(1,3,1),(1,5,1),
                        (2,4,1),(2,5,1),
                        (3,4,1),(3,5,1),
                        (4,5,1)]

anchura :: Grafo Int Int -> Int
anchura g = maximum [anchuraN g x | x <- nodos g]

-- (anchuraN g x) es la anchura del nodo x en el grafo g. Por ejemplo,
--    anchuraN g 1  ==  3
--    anchuraN g 2  ==  3
--    anchuraN g 3  ==  3
--    anchuraN g 4  ==  3
--    anchuraN g 5  ==  4
anchuraN :: Grafo Int Int -> Int -> Int
anchuraN g x = length (adyacentes g x)

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

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

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

-- (esAdmisible n) se verifica si n es admisible. Por ejemplo,
--    esAdmisible 32  ==  True
--    esAdmisible 48  ==  True
--    esAdmisible 15  ==  False
--    esAdmisible 10  ==  False
esAdmisible :: Integer -> Bool
esAdmisible n =
    even n &&
    (esPotenciaDeDos n || primosConsecutivos (nub (factorizacion n)))

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

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

-- (menorFactor n) es el menor factor primo de n. Por ejemplo,
--    menorFactor 15  ==  3
--    menorFactor 16  ==  2
--    menorFactor 17  == 17
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- (primosConsecutivos xs) se verifica si xs es una lista de primos
-- consecutivos. Por ejemplo,
--    primosConsecutivos [17,19,23]  ==  True
```

```
--    primosConsecutivos [17,19,29]  ==  False
--    primosConsecutivos [17,19,20]  ==  False
primosConsecutivos :: [Integer] -> Bool
primosConsecutivos [] = True
primosConsecutivos (x:xs) =
    take (1 + length xs) (dropWhile (<x) primos) == x:xs

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

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

3.2.8. Examen 8 (29 de Junio de 2012)

El examen es común con el del grupo 1 (ver página 138).

3.2.9. Examen 9 (9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 143).

3.2.10. Examen 10 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 148).

3.3. Exámenes del grupo 3 (Antonia M. Chávez)

3.3.1. Examen 1 (14 de Noviembre de 2011)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función posicion tal que (posicion x ys) es
-- la primera posición de x en ys. Por ejemplo,
```

```
--    posicion 5 [3,4,5,6,5]  ==  2
-- ---------------------------------------------------------------------

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

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

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

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

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

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

separaPorParidad xs = (pares xs, impares xs)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función eliminaElemento tal que
```

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir por comprensión, usando la lista [1..10], las
-- siguientes listas
--    l1 = [2,4,6,8,10]
--    l2 = [[1],[3],[5],[7],[9]]
--    l3 = [(1,2),(2,3),(3,4),(4,5),(5,6)]
-- ---------------------------------------------------------------------

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

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

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

3.3.2. Examen 2 (12 de Diciembre de 2011)

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por comprensión, la función
--    filtraAplicaC :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    sinRepetidos :: Eq a => [a] -> [a]
-- tal que (sinRepetidos xs) es la lista que resulta de quitar todos los
-- elementos repetidos de xs. Por ejemplo,
--    sinRepetidos [2,1,3,2,1,3,1]   ==  [2,3,1]
-- ---------------------------------------------------------------------

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

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

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

```
                  | otherwise   =  repetidos xs

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

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

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

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

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

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

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

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

3.3.3. Examen 7 (29 de Junio de 2012)

El examen es común con el del grupo 1 (ver página 138).

3.3.4. Examen 8 (09 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 143).

3.3.5. Examen 9 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 148).

3.4. Exámenes del grupo 4 (José F. Quesada)

3.4.1. Examen 1 (7 de Noviembre de 2011)

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

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

```
--    conjuntosIguales "arroz" "zorra"               == True
--    conjuntosIguales [1,2,2,3,2,1] [1,3,3,2,1,3,2] == True
--    conjuntosIguales [1,2,2,1] [1,2,3,2,1]         == False
--    conjuntosIguales [(1,2)] [(2,1)]               == False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    puntoInterior :: (Float,Float) -> Float -> (Float,Float) -> Bool
-- tal que (puntoInterior c r p) se verifica si p es un punto interior
-- del círculo de centro c y radio r. Por ejemplo,
--    puntoInterior (0,0) 1 (1,0)   == True
--    puntoInterior (0,0) 1 (1,1)   == False
--    puntoInterior (0,0) 2 (-1,-1) == True
-- ---------------------------------------------------------------------

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

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

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

```
--    tripletes 2 == []
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Las bases de datos de alumnos matriculados por
-- provincia y por especialidad se pueden representar como sigue
--    matriculas :: [(String,String,Int)]
--    matriculas = [("Almeria","Matematicas",27),
--                  ("Sevilla","Informatica",325),
--                  ("Granada","Informatica",296),
--                  ("Huelva","Matematicas",41),
--                  ("Sevilla","Matematicas",122),
--                  ("Granada","Matematicas",131),
--                  ("Malaga","Informatica",314)]
-- Es decir, se indica que por ejemplo en Almería hay 27 alumnos
-- matriculados en Matemáticas.
--
-- Definir la función
--    totalAlumnos :: [(String,String,Int)] -> Int
-- tal que (totalAlumnos bd) es el total de alumnos matriculados,
-- incluyendo todas las provincias y todas las especialidades, en la
-- base de datos bd. Por ejemplo,
--    totalAlumnos matriculas == 1256
-- ---------------------------------------------------------------------

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

```
              ("Malaga","Informatica",314)]

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

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

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

3.4.2. Examen 2 (30 de Noviembre de 2011)

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

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

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

-- (repunit n) se verifica si n es un repunit. Por ejemplo,
--    repunit 1111  ==  True
```

```
--    repunit 1121  ==  False
repunit :: Integer -> Bool
repunit n = and [x == 1 | x <- cifras n]

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las bases de datos con el nombre, profesión, año de
-- nacimiento y año de defunción de una serie de personas se puede
-- representar como sigue
--    personas :: [(String,String,Int,Int)]
--    personas = [("Cervantes","Literatura",1547,1616),
--                ("Velazquez","Pintura",1599,1660),
--                ("Picasso","Pintura",1881,1973),
--                ("Beethoven","Musica",1770,1823),
--                ("Poincare","Ciencia",1854,1912),
--                ("Quevedo","Literatura",1580,1654),
--                ("Goya","Pintura",1746,1828),
--                ("Einstein","Ciencia",1879,1955),
--                ("Mozart","Musica",1756,1791),
--                ("Botticelli","Pintura",1445,1510),
--                ("Borromini","Arquitectura",1599,1667),
--                ("Bach","Musica",1685,1750)]
-- Es decir, se indica que por ejemplo Mozart se dedicó a la Música y
-- vivió entre 1756 y 1791.
--
-- Definir la función
--    coetaneos :: [(String,String,Int,Int)] -> String -> [String]
-- tal que (coetaneos bd p) es la lista de nombres de personas que
-- fueron coetáneos con la persona p; es decir que al menos alguno
-- de los años vividos por ambos coincidan. Se considera que una persona
-- no es coetanea a sí misma. Por ejemplo,
--    coetaneos personas "Einstein"   == ["Picasso", "Poincare"]
--    coetaneos personas "Botticelli" == []
-- ---------------------------------------------------------------------
```

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

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

-- 2ª solución:

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

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

3.4.3. Examen 3 (16 de Enero de 2012)

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

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Dada una lista de números enteros, definiremos el
-- mayor salto como el mayor valor de las diferencias (en valor
-- absoluto) entre números consecutivos de la lista. Por ejemplo, dada
-- la lista [2,5,-3] las distancias son
--    3 (valor absoluto de la resta 2 - 5) y
--    8 (valor absoluto de la resta de 5 y (-3))
-- Por tanto, su mayor salto es 8. No está definido el mayor salto para
-- listas con menos de 2 elementos
--
-- Definir, por compresión, la función
--    mayorSaltoC :: [Integer] -> Integer
-- tal que (mayorSaltoC xs) es el mayor salto de la lista xs. Por
-- ejemplo,
--    mayorSaltoC [1,5]              == 4
--    mayorSaltoC [10,-10,1,4,20,-2] == 22
-- ---------------------------------------------------------------------

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

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

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

```
mayorSaltoR (x:y:ys) = max (abs (x-y)) (mayorSaltoR (y:ys))

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

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

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

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

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

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

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

```
          aux (x:xs) (y:ys) = aux xs (x+y:y:ys)

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, usando map, la función
--    porcentajesS :: [Float] -> [Float]
-- tal que (porcentajesS xs) es la lista de porcentaje de xs. Por
-- ejemplo,
```

```
--    porcentajesS [1,2,3,4] == [10.0,20.0,30.0,40.0]
--    porcentajesS [1,7,8,4]  ==  [5.0,35.0,40.0,20.0]
-- ---------------------------------------------------------------------

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

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

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

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

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

-- 2ª definición (por recursión)
equivalentes2 :: [Float] -> [Float] -> Bool
equivalentes2 [] []         = True
equivalentes2 _ []          = False
```

```
equivalentes2 [] _          = False
equivalentes2 (x:xs) (y:ys) = abs (x-y) <= 0.001 && equivalentes2 xs ys

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

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

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

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

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

3.4.4. Examen 4 (7 de Marzo de 2012)

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

import Test.QuickCheck
import Data.List

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    autodivisibleR :: Integer -> Bool
-- tal que (autodivisibleR n) se verifica si n es autodivisible. Por
-- ejemplo,
--    autodivisibleR 0       == True
--    autodivisibleR 25      == False
--    autodivisibleR 1234    == False
--    autodivisibleR 1234608 == True
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

```
--    cuentaArbol arbol1 2    = 0
--    cuentaArbol (N 5 H H) 5 = 1
--    cuentaArbol H 5         = 0
-----------------------------------------------

data Arbol = H
           | N Int Arbol Arbol
           deriving Show

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

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

3.4.5. Examen 5 (28 de Marzo de 2012)

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

import Data.List
import Test.QuickCheck
import PolRepTDA

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

```
--    moda []                    == []
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

```
        m <- ys]
    where ys = moda xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Representaremos un recorrido como una secuencia de
-- puntos en el espacio de dos dimensiones. Para ello utilizaremos la
-- siguiente definición
--    data Recorrido = Nodo Double Double Recorrido
--                   | Fin
--                     deriving Show
-- De esta forma, el recorrido que parte del punto (0,0) pasa por el
-- punto (1,2) y termina en el (2,4) se representará como
--    rec0 :: Recorrido
--    rec0 = Nodo 0 0 (Nodo 1 2 (Nodo 2 4 Fin))
--    A continuación se muestran otros ejemplos definidos
--    rec1, rec2, rec3, rec4 :: Recorrido
--    rec1 = Nodo 0 0 (Nodo 1 1 Fin)
--    rec2 = Fin
--    rec3 = Nodo 1 (-1) (Nodo 2 3 (Nodo 5 (-2) (Nodo 1 0 Fin)))
--    rec4 = Nodo 0 0
--             (Nodo 0 2
--               (Nodo 2 0
--                 (Nodo 0 0
--                   (Nodo 2 2
--                     (Nodo 2 0
--                       (Nodo 0 0 Fin))))))
--
-- Definir la función
--    distanciaRecorrido :: Recorrido -> Double
-- tal que (distanciaRecorrido ps) esla suma de las distancias de todos
-- los segmentos de un recorrido ps. Por ejemplo,
--    distanciaRecorrido rec0     ==  4.4721359549995
--    distanciaRecorrido rec1     ==  1.4142135623730951
--    distanciaRecorrido rec2     ==  0.0
-- ---------------------------------------------------------------------
```

```
data Recorrido = Nodo Double Double Recorrido
               | Fin
                 deriving Show

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

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

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


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

nodosDuplicados :: Recorrido -> Int
```

```
nodosDuplicados Fin = 0
nodosDuplicados (Nodo x y r)
    | existeNodo r x y = 1 + nodosDuplicados (eliminaNodo r x y)
    | otherwise        = nodosDuplicados r

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

-- (eliminaNodo r x y) es el recorrido obtenido eliminando en r las
-- ocurrencias del nodo (x,y). Por ejemplo,
--    ghci> rec3
--    Nodo 1.0 (-1.0) (Nodo 2.0 3.0 (Nodo 5.0 (-2.0) (Nodo 1.0 0.0 Fin)))
--    ghci> eliminaNodo rec3 2 3
--    Nodo 1.0 (-1.0) (Nodo 5.0 (-2.0) (Nodo 1.0 0.0 Fin))
eliminaNodo :: Recorrido -> Double -> Double -> Recorrido
eliminaNodo Fin _ _ = Fin
eliminaNodo (Nodo x y r) x' y'
            | x == x' && y == y' = eliminaNodo r x' y'
            | otherwise          = Nodo x y (eliminaNodo r x' y')

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se dice que un polinomio es completo si todos los
-- coeficientes desde el término nulo hasta el término de mayor grado
-- son distintos de cero.
--
-- Para hacer este ejercicio se utilizará algunas de las
-- implementaciones del tipo abstracto de datos de polinomio definidas
-- en el tema 21 y los siguientes ejemplos,
--    pol1, pol2, pol3 :: Polinomio Int
--    pol1 = polCero
--    pol2 = consPol 5 2 (consPol 3 1 (consPol 0 (-1) polCero))
--    pol3 = consPol 3 1 (consPol 2 2 (consPol 1 3 (consPol 0 4 polCero)))
--
```

```
-- Definir la función
--    polinomioCompleto :: Num a => Polinomio a -> Bool
-- tal que (polinomioCompleto p) se verifica si p es un polinomio
-- completo. Por ejemplo,
--    polinomioCompleto pol1  == False
--    polinomioCompleto pol2  == False
--    polinomioCompleto pol3  == True
-------------------------------------------------

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

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

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

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

3.4.6. Examen 6 (9 de Mayo de 2012)

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    aplicaT :: (Ix a, Num b) => Array a b -> (b -> c) -> Array a c
-- tal que (aplicaT t f) es la tabla obtenida aplicado la función f a
-- los elementos de la tabla t. Por ejemplo,
--    ghci> aplicaT (array (1,5) [(1,6),(2,3),(3,-1),(4,9),(5,20)]) (+1)
--    array (1,5) [(1,7),(2,4),(3,0),(4,10),(5,21)]
--    ghci> :{
--    *Main| aplicaT (array ((1,1),(2,3)) [((1,1),3),((1,2),-1),((1,3),0),
--    *Main|                               ((2,1),0),((2,2),0),((2,3),-1)])
--    *Main|         (*2)
--    *Main| :}
--    array ((1,1),(2,3)) [((1,1),6),((1,2),-2),((1,3),0),
--                         ((2,1),0),((2,2),0),((2,3),-2)]
-- ---------------------------------------------------------------------

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

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

```
-- polinomio. Por ejemplo,
--    ghci> polTabla (consPol 5 2 (consPol 3 (-1) polCero))
--    Tbl (array (0,5) [(0,0),(1,0),(2,0),(3,-1),(4,0),(5,2)])
-- -- ------------------------------------------------------------------

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

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

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

```
--                              ((2,1),2),((2,2),3),((2,3),4),
--                              ((3,1),3),((3,2),4),((3,3),5)]
--    p2 = array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
--                              ((2,1),2),((2,2),1),((2,3),4),
--                              ((3,1),3),((3,2),4),((3,3),5)]
-- ---------------------------------------------------------------------

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

matrizCreciente :: (Num a,Ord a) =>  Array (Int,Int) a -> Bool
matrizCreciente p =
    and ([p!(i,j) <= p!(i,j+1)   | i <- [1..m],   j <- [1..n-1]] ++
         [p!(i,j) <= p!(i+1,j)   | i <- [1..m-1], j <- [1..n]] ++
         [p!(i,j) <= p!(i+1,j+1) | i <- [1..m-1], j <- [1..n-1]])
    where (m,n) = snd (bounds p)


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

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

data Arbol = H
           | N Int Arbol Arbol
           deriving Show

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

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

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

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

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

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

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

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

3.4.7. Examen 7 (11 de Junio de 2012)

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

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

```
import GrafoConVectorDeAdyacencia
import ConjuntoConListasOrdenadasSinDuplicados

-- ---------------------------------------------------------------------
-- Ejercicio 1. Diremos que una lista de números es una reducción
-- general de un número entero N si el sumatorio de L es igual a N y,
-- además, L es una lista de números enteros consecutivos y ascendente,
-- con más de un elemento. Por ejemplo, las listas [1,2,3,4,5], [4,5,6]
-- y [7,8] son reducciones generales de 15
--
-- Definir, por comprensión, la función
--    reduccionesBasicas :: Integer -> [[Integer]]
-- tal que (reduccionesBasicas n) es la lista de reducciones de n cuya
-- longitud (número de elementos) sea menor o igual que la raíz cuadrada
-- de n. Por ejemplo,
--    reduccionesBasicas 15   ==  [[4,5,6],[7,8]]
--    reduccionesBasicas 232  ==  []
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Dada una matriz numérica A de dimensiones (m,n) y una
-- matriz booleana B de las mismas dimensiones, y dos funciones f y g,
-- la transformada de A respecto de B, f y g es la matriz C (de las
-- mismas dimensiones), tal que, para cada celda (i,j):
--    C(i,j) = f(A(i,j)) si B(i,j) es verdadero
--    C(i,j) = f(A(i,j)) si B(i,j) es falso
-- Por ejemplo, si A y B son las matrices
--    |1 2|   |True  False|
```

```
--    |3 4|   |False True |
-- respectivamente, y f y g son dos funciones tales que f(x) = x+1 y
-- g(x) = 2*x, entonces la transformada de A respecto de B, f y g es
--    |2 4|
--    |6 5|
--
-- En Haskell,
--    a :: Array (Int,Int) Int
--    a = array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),3),((2,2),4)]
--
--    b :: Array (Int,Int) Bool
--    b = array ((1,1),(2,2)) [((1,1),True),((1,2),False),((2,1),False),((2,2),True)]
--
-- Definir la función
--    transformada :: Array (Int,Int) a -> Array (Int,Int) Bool ->
--                    (a -> b) -> (a -> b) -> Array (Int,Int) b
-- tal que (transformada a b f g) es la transformada de A respecto de B,
-- f y g. Por ejemplo,
--    ghci> transformada a b (+1) (*2)
--    array ((1,1),(2,2)) [((1,1),2),((1,2),4),((2,1),6),((2,2),5)]
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Dado un grafo dirigido G, diremos que un nodo está
-- aislado si o bien de dicho nodo no sale ninguna arista o bien no
-- llega al nodo ninguna arista. Por ejemplo, en el siguiente grafo
```

```
-- (Tema 22, pag. 31)
--    g = creaGrafo D (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
--                           (5,4,0),(6,2,0),(6,5,0)]
-- podemos ver que del nodo 1 salen 3 aristas pero no llega ninguna, por
-- lo que lo consideramos aislado. Así mismo, a los nodos 2 y 4 llegan
-- aristas pero no sale ninguna, por tanto también estarán aislados.
--
-- Definir la función
--    aislados :: (Ix v, Num p) => Grafo v p -> [v]
-- tal que (aislados g) es la lista de nodos aislados del grafo g. Por
-- ejemplo,
--    aislados g == [1,2,4]
-- ---------------------------------------------------------------------

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

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

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

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

```
--    p2 = consPol 7 (-2) (consPol 5 1 (consPol 4 5 (consPol 2 1 polCero)))
--    p3 = polCero
--    p4 = consPol 4 (-1) (consPol 3 2 (consPol 1 1 polCero))
-- se tiene que
--    ghci> gradosCoeficientes [p1,p2,p3,p4]
--    [(1,{0,1}),(2,{0,1}),(3,{-1,0,2}),(4,{-1,0,5}),(5,{0,1,2}),
--     (6,{0}),(7,{-2,0})]
-- ---------------------------------------------------------------------

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

gradosCoeficientes :: (Ord a, Num a) => [Polinomio a] -> [(Int,Conj a)]
gradosCoeficientes ps =
    [(k,foldr (inserta . coeficiente k) vacio ps) | k <- [1..m]]
    where m = maximum (map grado ps)

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

3.4.8. Examen 8 (29 de Junio de 2012)

El examen es común con el del grupo 1 (ver página 138).

3.4.9. Examen 9 (9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 143).

3.4.10. Examen 10 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 148).

4

Exámenes del curso 2012–13

4.1. Exámenes del grupo 1 (Antonia M. Chávez)

4.1.1. 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]
```

```
--    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))
```

4.1.2. 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
-- 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.

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

```
-- 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 ..]]

-- ---------------------------------------------------------------------
-- 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]
```

4.1.3. Examen 3 (6 de febrero de 2013)

El examen es común con el del grupo 2 (ver página 249).

4.1.4. 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

-- ---------------------------------------------------------------------
-- 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
-- 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

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
-- ---------------------------------------------------------------------

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.
```

4.1.5. Examen 5 (15 de mayo de 2013)

```
-- 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]
-- 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?
-- ---------------------------------------------------------------------

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
```

4.1.6. 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.
--
-- 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]
-- tal que (nodosInternos a) es la lista de los nodos internos del
-- arbol 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]]
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
-- 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)]
```

4.1.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 2 (ver página 264).

4.1.8. Examen 8 (13 de septiembre de 2013)

El examen es común con el del grupo 2 (ver página 271).

4.1.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 2 (ver página 275).

4.2. Exámenes del grupo 2 (José A. Alonso y Miguel A. Martínez)

4.2.1. 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)
-- 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]
```

4.2.2. Examen 2 (20 de diciembre de 2012)

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

-- ---------------------------------------------------------------------
-- 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]

-- (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 número
-- 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
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
```

4.2.3. Examen 3 (6 de febrero de 2013)

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

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

-- (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
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]
```

4.2.4. Examen 4 (21 de marzo de 2013)

```
-- 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
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]
```

4.2.5. 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
--    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)]
-- 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)

-- (borraCol j1 p) es la matriz obtenida borrando la columna j1 de la
-- matriz p. Por ejemplo,
--    ghci> let p = listArray ((1,1),(2,4)) [1..8]
--    ghci> borraCol 2 p
--    array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),4),((2,1),5),((2,2),7),((2,3),8
--    ghci> borraCol 3 p
--    array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),4),((2,1),5),((2,2),6),((2,3),8
borraCol :: 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
```

4.2.6. Examen 6 (13 de junio de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (13 de junio de 2013)
-- ---------------------------------------------------------------------

import Data.Array

-- ---------------------------------------------------------------------
-- 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 [] = []
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
-- 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. [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]

-- ---------------------------------------------------------------------
-- 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)]
```

4.2.7. Examen 7 (3 de julio de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 7º examen de evaluación continua (3 de julio de 2013)
-- ---------------------------------------------------------------------

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)])

-- 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
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
-- 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]

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
```

4.2.8. Examen 8 (13 de septiembre de 2013)

```
-- 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)
--    99999999999999999999999999999999999999999999999999999999999999999

-- ---------------------------------------------------------------------
-- 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]]
-- 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

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]

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)
```

4.2.9. Examen 9 (20 de noviembre de 2013)

```
-- 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
-- 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
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
-- 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
-- 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
```

4.3. Exámenes del grupo 3 (María J. Hidalgo)

4.3.1. Examen 1 (16 de noviembre de 2012)

```
-- 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]

-- ---------------------------------------------------------------------
```

```
-- 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]
```

4.3.2. 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,
--   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
-- ---------------------------------------------------------------------

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,
--    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
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)

-- ---------------------------------------------------------------------
-- 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
-- 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
--    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)
```

4.3.3. Examen 3 (6 de febrero de 2013)

El examen es común con el del grupo 2 (ver página 249).

4.3.4. Examen 4 (22 de marzo de 2013)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 4º examen de evaluación continua (22 de marzo de 2013)
-- ---------------------------------------------------------------------

import Data.List
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

-- (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.

-- ---------------------------------------------------------------------
-- 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]]

-- 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
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))
--
-- 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
```

4.3.5. Examen 5 (10 de mayo de 2013)

```
-- 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
-- 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
-- ---------------------------------------------------------------------

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)]
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]
```

4.3.6. 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,
--    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)

-- 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]

-- 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)]
-- ---------------------------------------------------------------------

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)
```

4.3.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 2 (ver página 264).

4.3.8. Examen 8 (13 de septiembre de 2013)

El examen es común con el del grupo 2 (ver página 271).

4.3.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 2 (ver página 275).

4.4. Exámenes del grupo 4 (Andrés Cordón e Ignacio Pérez)

4.4.1. 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

-- ---------------------------------------------------------------------
-- 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
```

4.4.2. 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
-- ---------------------------------------------------------------------

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
```

4.4.3. Examen 3 (6 de febrero de 2013)

El examen es común con el del grupo 2 (ver página 249).

4.4.4. Examen 4 (18 de marzo de 2013)

```
-- 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]

-- ---------------------------------------------------------------------
-- 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]
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

-- (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)]
    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]
```

4.4.5. 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]
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"
-- ---------------------------------------------------------------------

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]
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
--    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)]
```

4.4.6. Examen 6 (13 de junio de 2013)

El examen es común con el del grupo 1 (ver página 238).

4.4.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 2 (ver página 264).

4.4.8. Examen 8 (13 de septiembre de 2013)

El examen es común con el del grupo 2 (ver página 271).

4.4.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 2 (ver página 275).

5

Exámenes del curso 2013–14

5.1. Exámenes del grupo 1 (María J. Hidalgo)

5.1.1. Examen 1 (7 de Noviembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (7 de noviembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1 [Del problema 21 del Proyecto Euler]. Sea d(n) la suma de
-- los divisores propios de n. Si d(a) = b y d(b) = a, siendo a ≠ b,
-- decimos que a y b son un par de números amigos. Por ejemplo, los
-- divisores propios de 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y
-- 110; por tanto, d(220) = 284. Los divisores propios de 284 son 1, 2,
-- 4, 71 y 142; por tanto,  d(284) = 220.  Luego, 220 y 284 son dos
-- números amigos.
--
-- Definir la función amigos tal que (amigos a b) se verifica si a y b
-- son números amigos. Por ejemplo,
--    amigos 6 6       == False
--    amigos 220 248   == False
--    amigos 220 284   == True
--    amigos 100 200   == False
--    amigos 1184 1210 == True
-- ---------------------------------------------------------------------

amigos a b = sumaDivisores a == b && sumaDivisores b == a
    where sumaDivisores n = sum [x | x<-[1..n-1], n `rem` x == 0]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Una representación de 20 en base 2 es [0,0,1,0,1] pues
-- 20 = 1*2^2 + 1*2^4. Y una representación de 46 en base 3 es [1,0,2,1]
-- pues 46 = 1*3^0 + 0*3^1 + 2*3^2 + 1*3^3.
--
-- Definir la función deBaseABase10 tal que (deBaseABase10 b xs) es el
-- número n tal que su representación en base b es xs. Por ejemplo,
--    deBaseABase10 2 [0,0,1,0,1]      == 20
--    deBaseABase10 2 [1,1,0,1]        == 11
--    deBaseABase10 3 [1,0,2,1]        == 46
--    deBaseABase10 5 [0,2,1,3,1,4,1]  == 2916
-- ---------------------------------------------------------------------

deBaseABase10 b xs = sum [y*b^n | (y,n) <- zip xs [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [De la IMO-1996-S-21]. Una sucesión [a(0),a(1),...,a(n)]
-- se denomina cuadrática si para cada i ∈ {1, 2,..., n} se cumple que
--    |a(i) − a(i−1)| = i^2 .
-- Definir una función esCuadratica tal que (esCuadratica xs) se
-- verifica si xs cuadrática.  Por ejemplo,
--    esCuadratica [2,1,-3,6]                     == True
--    esCuadratica [2,1,3,5]                      == False
--    esCuadratica [3,4,8,17,33,58,94,45,-19,-100] == True
-- ---------------------------------------------------------------------

esCuadratica xs =
    and [abs (y-x) == i^2 | ((x,y),i) <- zip (adyacentes xs) [1..]]

adyacentes xs = zip xs (tail xs)

-- ---------------------------------------------------------------
-- Ejercicio 4.1. Sea t una lista de pares de la forma
--    (nombre, [(asig1, nota1),...,(asigk, notak)])
-- Por ejemplo,
--    t1 = [("Ana",[("Algebra",1),("Calculo",3),("Informatica",8),("Fisica",2)]),
--          ("Juan",[("Algebra",5),("Calculo",1),("Informatica",2),("Fisica",9)]),
--          ("Alba",[("Algebra",5),("Calculo",6),("Informatica",6),("Fisica",5)]),
--          ("Pedro",[("Algebra",9),("Calculo",5),("Informatica",3),("Fisica",1)])]
```

```
-- Definir la función calificaciones tal que (calificaciones t p) es la
-- lista de las calificaciones de la persona p en la lista t. Por
-- ejemplo,
--    ghci> calificaciones t1 "Pedro"
--    [("Algebra",9),("Calculo",5),("Informatica",3),("Fisica",1)]
-- ---------------------------------------------------------------------

t1 = [("Ana",[("Algebra",1),("Calculo",3),("Informatica",8),("Fisica",2)]),
      ("Juan",[("Algebra",5),("Calculo",1),("Informatica",2),("Fisica",9)]),
      ("Alba",[("Algebra",5),("Calculo",6),("Informatica",6),("Fisica",5)]),
      ("Pedro",[("Algebra",9),("Calculo",5),("Informatica",3),("Fisica",1)])]

calificaciones t p = head [xs | (x,xs) <-t, x == p]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función todasAprobadas tal que
-- (todasAprobadas t p) se cumple si en la lista t, p tiene todas las
-- asignaturas aprobadas. Por ejemplo,
--    todasAprobadas t1 "Alba"  ==  True
--    todasAprobadas t1 "Pedro" ==  False
-- ---------------------------------------------------------------------

todasAprobadas t p = numeroAprobados t p == numeroAsignaturas t p

numeroAprobados t p = length [n | (_,n) <- calificaciones t p, n >= 5]

numeroAsignaturas t p = length (calificaciones t p)

apruebanTodo t = [p | (p,_) <- t, todasAprobadas t p]
```

5.1.2. Examen 2 (19 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (19 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
```

```
--     bocata :: Eq a => a -> a -> [a] -> [a]
-- tal que (bocata a b xs) es la lista obtenida colocando b delante y
-- detrás de todos los elementos de xs que coinciden con a. Por ejemplo,
--     bocata "chorizo" "pan" ["jamon", "chorizo", "queso", "chorizo"]
--     ["jamon","pan","chorizo","pan","queso","pan","chorizo","pan"]
--     bocata "chorizo" "pan" ["jamon", "queso", "atun"]
--     ["jamon","queso","atun"]
-- ---------------------------------------------------------------------

bocata :: Eq a => a -> a -> [a] -> [a]
bocata _ _ []     = []
bocata a b (x:xs) | x == a    = b : a : b : bocata a b xs
                  | otherwise = x : bocata a b xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que el número de elementos de
-- (bocata a b xs) es el número de elementos de xs más el doble del
-- número de elementos de xs que coinciden con a.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_bocata :: String -> String -> [String] -> Bool
prop_bocata a b xs =
    length (bocata a b xs) == length xs + 2 * length (filter (==a) xs)

-- La comprobación es
--    ghci> quickCheck prop_bocata
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    mezclaDigitos :: Integer -> Integer -> Integer
-- tal que (mezclaDigitos n m) es el número formado intercalando los
-- dígitos de n y m, empezando por los de n. Por ejemplo,
--    mezclaDigitos 12583 4519       == 142551893
--    mezclaDigitos 12583 4519091256 == 142551893091256
-- ---------------------------------------------------------------------

mezclaDigitos :: Integer -> Integer -> Integer
mezclaDigitos n m =
```

```
    read (intercala (show n) (show m))

-- (intercala xs ys) es la lista obtenida intercalando los elementos de
-- xs e ys. Por ejemplo,
--    intercala [2,5,3] [4,7,9,6,0]  ==  [2,4,5,7,3,9,6,0]
--    intercala [4,7,9,6,0] [2,5,3]   ==  [4,2,7,5,9,3,6,0]
intercala :: [a] -> [a] -> [a]
intercala [] ys = ys
intercala xs [] = xs
intercala (x:xs) (y:ys) = x : y : intercala xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Problema 211 del proyecto Euler) Dado un entero
-- positivo n, consideremos la suma de los cuadrados de sus divisores,
-- Por ejemplo,
--    f(10) = 1 + 4 + 25 + 100 = 130
--    f(42) = 1 + 4 +  9 +  36 + 49 + 196 + 441 + 1764 = 2500
-- Decimos que n es especial si f(n) es un cuadrado perfecto. En los
-- ejemplos anteriores, 42 es especial y 10 no lo es.
--
-- Definir la función
--    especial:: Int -> Bool
-- tal que (especial x) se verifica si x es un número es especial. Por
-- ejemplo,
--    especial 42  ==  True
--    especial 10  ==  False
-- Calcular todos los números especiales de tres cifras.
-- ---------------------------------------------------------------------

especial:: Int -> Bool
especial n = esCuadrado (sum (map (^2) (divisores n)))

-- (esCuadrado n) se verifica si n es un cuadrado perfecto. Por ejemplo,
--    esCuadrado 36  ==  True
--    esCuadrado 40  ==  False
esCuadrado :: Int -> Bool
esCuadrado n = y^2 == n
  where y = floor (sqrt (fromIntegral n))

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
```

```
--    divisores 36  ==  [1,2,3,4,6,9,12,18,36]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], rem n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    verificaMax :: (a -> Bool) -> [[a]] -> [a]
-- tal que (verificaMax p xss) es la lista de xss con mayor número de
-- elementos que verifican la propiedad p. Por ejemplo,
--    ghci> verificaMax even [[1..5], [2,4..20], [3,2,1,4,8]]
--    [2,4,6,8,10,12,14,16,18,20]
--    ghci> verificaMax even [[1,2,3], [6,8], [3,2,10], [3]]
--    [6,8]
-- Nota: En caso de que haya más de una lista, obtener la primera.
-- ---------------------------------------------------------------------

verificaMax :: (a -> Bool) -> [[a]] -> [a]
verificaMax p xss = head [xs | xs <- xss, test xs]
    where f xs    = length [x | x <- xs, p x]
          m       = maximum [f xs | xs <-xss]
          test xs = f xs == m
```

5.2. Exámenes del grupo 3 (José A. Alonso y Luis Valencia)

5.2.1. Examen 1 (5 de Noviembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (5 de noviembre de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Definir la función
--    divisoresPrimos :: Integer -> [Integer]
-- tal que (divisoresPrimos x) es la lista de los divisores primos de x.
-- Por ejemplo,
--    divisoresPrimos 40  ==  [2,5]
--    divisoresPrimos 70  ==  [2,5,7]
```

```
-- ---------------------------------------------------------------------

divisoresPrimos :: Integer -> [Integer]
divisoresPrimos x = [n | n <- divisores x, primo n]

-- (divisores n) es la lista de los divisores del número n. Por ejemplo,
--    divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores :: Integer -> [Integer]
divisores n = [x | x <- [1..n], n `mod` x == 0]

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
primo :: Integer -> Bool
primo n = divisores n == [1, n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] La multiplicidad de x en y es la mayor
-- potencia de x que divide a y. Por ejemplo, la multiplicidad de 2 en
-- 40 es 3 porque 40 es divisible por 2^3 y no lo es por 2^4. Además, la
-- multiplicidad de 1 en cualquier número se supone igual a 1.
--
-- Definir la función
--    multiplicidad :: Integer -> Integer -> Integer
-- tal que (multiplicidad x y) es la
-- multiplicidad de x en y. Por ejemplo,
--    multiplicidad 2 40  ==  3
--    multiplicidad 5 40  ==  1
--    multiplicidad 3 40  ==  0
--    multiplicidad 1 40  ==  1
-- ---------------------------------------------------------------------

multiplicidad :: Integer -> Integer -> Integer
multiplicidad 1 _ = 1
multiplicidad x y =
    head [n | n <- [0..], y `rem` (x^n) == 0, y `rem` (x^(n+1)) /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Un número es libre de cuadrados si no es
-- divisible el cuadrado de ningún entero mayor que 1. Por ejemplo, 70
```

```
-- es libre de cuadrado porque sólo es divisible por 1, 2, 5, 7 y 70; en
-- cambio, 40 no es libre de cuadrados porque es divisible por 2^2.
--
-- Definir la función
--    libreDeCuadrados :: Integer -> Bool
-- tal que (libreDeCuadrados x) se verifica si x es libre de cuadrados.
-- Por ejemplo,
--    libreDeCuadrados 70  ==  True
--    libreDeCuadrados 40  ==  False
-- Calcular los 10 primeros números libres de cuadrado de 3 cifras.
-- ---------------------------------------------------------------------

-- 1ª definición:
libreDeCuadrados :: Integer -> Bool
libreDeCuadrados x = x == product (divisoresPrimos x)

-- NOTA: La función primo está definida en el ejercicio 1.

-- 2ª definición
libreDeCuadrados2 :: Integer -> Bool
libreDeCuadrados2 x =
    and [multiplicidad n x == 1 | n <- divisores x]

-- 3ª definición
libreDeCuadrados3 :: Integer -> Bool
libreDeCuadrados3 n =
    null [x | x <- [2..n], rem n (x^2) == 0]


-- El cálculo es
--    ghci> take 10 [n | n <- [100..], libreDeCuadrados n]
--    [101,102,103,105,106,107,109,110,111,113]

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] La distancia entre dos números es el valor
-- absoluto de su diferencia. Por ejemplo, la distancia entre 2 y 5 es
-- 3.
--
-- Definir la función
--    cercanos :: [Int] -> [Int] -> [(Int,Int)]
```

```
-- tal que (cercanos xs ys) es la lista de pares de elementos de xs e ys
-- cuya distancia es mínima. Por ejemplo,
--    cercanos [3,7,2,1] [5,11,9]  ==  [(3,5),(7,5),(7,9)]
-- ---------------------------------------------------------------------

cercanos :: [Int] -> [Int] -> [(Int,Int)]
cercanos xs ys =
    [(x,y) | (x,y) <- pares, abs (x-y) == m]
    where pares = [(x,y) | x <- xs, y <- ys]
          m = minimum [abs (x-y) | (x,y) <- pares]
```

5.2.2. Examen 2 (17 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (17 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List (sort)

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Definir la función
--    expandida :: [Int] -> [Int]
-- tal que (expandida xs) es la lista obtenida duplicando cada uno de
-- los elementos pares de xs. Por ejemplo,
--    expandida [3,5,4,6,6,1,0]  ==  [3,5,4,4,6,6,6,6,1,0,0]
--    expandida [3,5,4,6,8,1,0]  ==  [3,5,4,4,6,6,8,8,1,0,0]
-- ---------------------------------------------------------------------

expandida :: [Int] -> [Int]
expandida [] = []
expandida (x:xs) | even x    = x : x : expandida xs
                 | otherwise = x : expandida xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Comprobar con QuickCheck que el número de
-- elementos de (expandida xs) es el del número de elementos de xs más
-- el número de elementos pares de xs.
-- ---------------------------------------------------------------------

prop_expandida :: [Int] -> Bool
```

```
prop_expandida xs =
    length (expandida xs) == length xs + length (filter even xs)

-- La comprobación es
--    ghci> quickCheck prop_expandida
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Definir la función
--    digitosOrdenados :: Integer -> Integer
-- tal que (digitosOrdenados n) es el número obtenido ordenando los
-- dígitos de n de mayor a menor. Por ejemplo,
--    digitosOrdenados 325724237  ==  775433222
-- ---------------------------------------------------------------------

digitosOrdenados :: Integer -> Integer
digitosOrdenados n = read (ordenados (show n))

ordenados :: Ord a => [a] -> [a]
ordenados [] = []
ordenados (x:xs) =
    ordenados mayores ++ [x] ++ ordenados menores
    where mayores = [y | y <- xs, y > x]
          menores = [y | y <- xs, y <= x]

-- Nota: La función digitosOrdenados puede definirse por composición
digitosOrdenados2 :: Integer -> Integer
digitosOrdenados2 = read . ordenados . show

-- Nota: La función digitosOrdenados puede definirse por composición y
-- también usando sort en lugar de ordenados
digitosOrdenados3 :: Integer -> Integer
digitosOrdenados3 = read . reverse . sort . show

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] Sea f la siguiente función, aplicable a
-- cualquier número entero positivo:
-- * Si el número es par, se divide entre 2.
-- * Si el número es impar, se multiplica por 3 y se suma 1.
--
```

```
-- La carrera de Collatz consiste en, dada una lista de números ns,
-- sustituir cada número n de ns por f(n) hasta que alguno sea igual a
-- 1. Por ejemplo, la siguiente sucesión es una carrera de Collatz
--    [ 3, 6,20, 49, 73]
--    [10, 3,10,148,220]
--    [ 5,10, 5, 74,110]
--    [16, 5,16, 37, 55]
--    [ 8,16, 8,112,166]
--    [ 4, 8, 4, 56, 83]
--    [ 2, 4, 2, 28,250]
--    [ 1, 2, 1, 14,125]
-- En esta carrera, los ganadores son 3 y 20.
--
-- Definir la función
--    ganadores :: [Int] -> [Int]
--    ganadores [3,6,20,49,73]  ==  [3,20]
-- ---------------------------------------------------------------------

ganadores :: [Int] -> [Int]
ganadores xs = selecciona xs (final xs)

-- (final xs) es el estado final de la carrera de Collatz a partir de
-- xs. Por ejemplo,
--    final [3,6,20,49,73]  ==  [1,2,1,14,125]
final :: [Int] -> [Int]
final xs | elem 1 xs = xs
         | otherwise = final [siguiente x | x <- xs]

-- (siguiente x) es el siguiente de x en la carrera de Collatz. Por
-- ejemplo,
--    siguiente 3  ==  10
--    siguiente 6  ==  3
siguiente :: Int -> Int
siguiente x | even x    = x `div` 2
            | otherwise = 3*x+1

-- (selecciona xs ys) es la lista de los elementos de xs cuyos tales que
-- los elementos de ys en la misma posición son iguales a 1. Por ejemplo,
--    selecciona [3,6,20,49,73] [1,2,1,14,125]  ==  [3,20]
selecciona :: [Int] -> [Int] -> [Int]
```

```
selecciona xs ys =
    [x | (x,y) <- zip xs ys, y == 1]
```

5.3. Exámenes del grupo 4 (Francisco J. Martín)

5.3.1. Examen 1 (5 de Noviembre de 2013)

```
-- Informática (1º del Grado en Matemáticas y en Estadística)
-- 1º examen de evaluación continua (11 de noviembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función listaIgualParidad tal que al
-- evaluarla sobre una lista de números naturales devuelva la lista de
-- todos los elementos con la misma paridad que la posición que ocupan
-- (contada desde 0); es decir, todos los pares en una posición par y
-- todos los impares en una posición impar. Por ejemplo,
--    listaIgualParidad [1,3,5,7]  ==  [3,7]
--    listaIgualParidad [2,4,6,8]  ==  [2,6]
--    listaIgualParidad [1..10]    ==  []
--    listaIgualParidad [0..10]    ==  [0,1,2,3,4,5,6,7,8,9,10]
--    listaIgualParidad []         ==  []
-- ---------------------------------------------------------------------

listaIgualParidad xs = [x | (x,i) <- zip xs [0..], even x == even i]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Decimos que una lista está equilibrada si el número de
-- elementos de la lista que son menores que la media es igual al número
-- de elementos de la lista que son mayores.
--
-- Definir la función listaEquilibrada que comprueba dicha propiedad
-- para una lista. Por ejemplo,
--    listaEquilibrada [1,7,1,6,2]  ==  False
--    listaEquilibrada [1,7,4,6,2]  ==  True
--    listaEquilibrada [8,7,4,6,2]  ==  False
--    listaEquilibrada []           ==  True
-- ---------------------------------------------------------------------

listaEquilibrada xs =
```

```
    length [y | y <- xs, y < media xs] ==
    length [y | y <- xs, y > media xs]

-- (media xs) es la media de xs. Por ejemplo,
--    media [6, 3, 9]  ==  6.0
media xs = sum xs / fromIntegral (length xs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. El trozo inicial de los elementos de una lista que
-- cumplen una propiedad es la secuencia de elementos de dicha lista
-- desde la posición 0 hasta el primer elemento que no cumple la
-- propiedad, sin incluirlo.
--
-- Definirla función trozoInicialPares que devuelve el trozo inicial de
-- los elementos de una lista que son pares. Por ejemplo,
--    trozoInicialPares []         ==  []
--    trozoInicialPares [1,2,3,4]  ==  []
--    trozoInicialPares [2,4,3,2]  ==  [2,4]
--    trozoInicialPares [2,4,6,8]  ==  [2,4,6,8]
-- ---------------------------------------------------------------------

trozoInicialPares xs = take (posicionPrimerImpar xs) xs

-- (posicionPrimerImpar xs) es la posición del primer elemento impar de
-- la lista xs o su longitud si no hay ninguno. Por ejemplo,
--    posicionPrimerImpar [2,4,3,2]  ==  2
--    posicionPrimerImpar [2,4,6,2]  ==  4
posicionPrimerImpar xs =
    head ([i | (x,i) <- zip xs [0..], odd x] ++ [length xs])

-- La función anterior se puede definir por recursión
posicionPrimerImpar2 [] = 0
posicionPrimerImpar2 (x:xs)
    | odd x     = 0
    | otherwise = 1 + posicionPrimerImpar2 xs

-- 2ª definición (por recursión).
trozoInicialPares2 [] = []
trozoInicialPares2 (x:xs) | odd x     = []
                          | otherwise = x : trozoInicialPares2 xs
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El registro de entradas vendidas de un cine se
-- almacena en una lista en la que cada elemento tiene el título de una
-- película, a continuación el número de entradas vendidas sin promoción
-- a 6 euros y por último el número de entradas vendidas con alguna de las
-- promociones del cine (menores de 4 años, mayores de 60, estudiantes
-- con carnet) a 4 euros. Por ejemplo,
--    entradas = [("Gravity",22,13),("Séptimo",18,6), ("Turbo",19,0),
--                ("Gravity",10,2), ("Séptimo",22,10),("Turbo",32,10),
--                ("Gravity",18,8), ("Séptimo",20,14),("Turbo",18,10)]
--
-- Definir la función ingresos tal que (ingresos bd) sea el total de
-- ingresos obtenidos según la información sobre entradas vendidas
-- almacenada en la lista 'bd'. Por ejemplo,
--    ingressos entradas  =  1366
-- ---------------------------------------------------------------------

entradas = [("Gravity",22,13),("Séptimo",18,6), ("Turbo",19,0),
            ("Gravity",10,2), ("Séptimo",22,10),("Turbo",32,10),
            ("Gravity",18,8), ("Séptimo",20,14),("Turbo",18,10)]

ingresos bd = sum [6*y+4*z | (_,y,z) <- bd]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función ingresosPelicula tal que
-- (ingresos bd p) sea el total de ingresos obtenidos en las distintas
-- sesiones de la película p según la información sobre entradas
-- vendidas almacenada en la lista bd. Por ejemplo,
--    ingresosPelicula entradas "Gravity"  ==  392
--    ingresosPelicula entradas "Séptimo"  ==  480
--    ingresosPelicula entradas "Turbo"    ==  494
-- ---------------------------------------------------------------------

ingresosPelicula bd p = sum [6*y+4*z | (x,y,z) <- bd, x == p]

-- =====================================================================
```

5.3.2. Examen 2 (16 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas y en Estadística)
-- 2º examen de evaluación continua (16 de diciembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
-- numeroConsecutivosC tal que (numeroConsecutivosC xs) es la cantidad
-- de números consecutivos que aparecen al comienzo de la lista xs. Por
-- ejemplo,
--    numeroConsecutivosC []               ==  0
--    numeroConsecutivosC [1,3,5,7,9]      ==  1
--    numeroConsecutivosC [1,2,3,4,5,7,9]  ==  5
-- ---------------------------------------------------------------------

numeroConsecutivosC [] = 0
numeroConsecutivosC xs =
    1 + length [x | (x,y) <- zip xs (tail xs), y == x+1]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función numeroConsecutivosR
-- tal que (numeroConsecutivosR xs) es la cantidad de números
-- consecutivos que aparecen al comienzo de la lista xs. Por ejemplo,
--    numeroConsecutivosR []               ==  0
--    numeroConsecutivosR [1,3,5,7,9]      ==  1
--    numeroConsecutivosR [1,2,3,4,5,7,9]  ==  5
-- ---------------------------------------------------------------------

numeroConsecutivosR []  = 0
numeroConsecutivosR [x] = 1
numeroConsecutivosR (x:y:ys)
    | y == x+1  = 1 + numeroConsecutivosR (y:ys)
    | otherwise = 1

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Una sustitución es una lista de parejas
-- [(x1,y1),...,(xn,yn)] que se usa para indicar que hay que reemplazar
-- cualquier ocurrencia de cada uno de los xi, por el correspondiente
-- yi. Por ejemplo,
--    sustitucion = [('1','a'),('2','n'),('3','v'),('4','i'),('5','d')]
```

```
-- es la sustitución que reemplaza '1' por 'a', '2' por 'n', ...
--
-- Definir, por comprensión, la función sustitucionEltC tal que
-- (sustitucionEltC xs z) es el resultado de aplicar la sustitución xs
-- al elemento z. Por ejemplo,
--    sustitucionEltC sustitucion '4'  ==  'i'
--    sustitucionEltC sustitucion '2'  ==  'n'
--    sustitucionEltC sustitucion '0'  ==  '0'
-- ---------------------------------------------------------------------

sustitucion = [('1','a'),('2','n'),('3','v'),('4','i'),('5','d')]

sustitucionEltC xs z = head [y | (x,y) <- xs, x == z] ++ [z]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función sustitucionEltR tal
-- que (sustitucionEltR xs z) es el resultado de aplicar la sustitución
-- xs al elemento z. Por ejemplo,
--    sustitucionEltR sustitucion '4'  ==  'i'
--    sustitucionEltR sustitucion '2'  ==  'n'
--    sustitucionEltR sustitucion '0'  ==  '0'
-- ---------------------------------------------------------------------

sustitucionEltR [] z = z
sustitucionEltR ((x,y):xs) z
    | x == z    = y
    | otherwise = sustitucionEltR xs z

-- ---------------------------------------------------------------------
-- Ejercicio 2.3, Definir, por comprensión, la función sustitucionLstC
-- tal que (sustitucionLstC xs zs) es el resultado de aplicar la
-- sustitución xs a los elementos de la lista zs. Por ejemplo,
--    sustitucionLstC sustitucion "2151"      ==  "nada"
--    sustitucionLstC sustitucion "3451"     ==  "vida"
--    sustitucionLstC sustitucion "2134515"  ==  "navidad"
-- ---------------------------------------------------------------------

sustitucionLstC xs zs = [sustitucionEltC xs z | z <- zs]

-- ---------------------------------------------------------------------
```

```
-- Ejercicio 2.4. Definir, por recursión, la función sustitucionLstR tal
-- que (sustitucionLstR xs zs) es el resultado de aplicar la sustitución
-- xs a los elementos de la lista zs. Por ejemplo,
--    sustitucionLstR sustitucion "2151"     ==  "nada"
--    sustitucionLstR sustitucion "3451"     ==  "vida"
--    sustitucionLstR sustitucion "2134515"  ==  "navidad"
-- ------------------------------------------------------------------

sustitucionLstR xs []     = []
sustitucionLstR xs (z:zs) =
     sustitucionEltR xs z : sustitucionLstR xs zs

-- ------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión, la función sublista tal que
-- (sublista xs ys) se verifica si todos los elementos de xs aparecen en
-- ys en el mismo orden aunque no necesariamente consecutivos. Por ejemplo,
--    sublista "meta"   "matematicas"  ==  True
--    sublista "temas"  "matematicas"  ==  True
--    sublista "mitica" "matematicas"  ==  False
-- ------------------------------------------------------------------

sublista []     ys = True
sublista (x:xs) [] = False
sublista (x:xs) (y:ys)
    | x == y           = sublista xs ys
    | otherwise        = sublista (x:xs) ys

-- ------------------------------------------------------------------
-- Ejercicio 4. Definir, por recursión, la función numeroDigitosPares
-- tal que (numeroDigitosPares n) es la cantidad de dígitos pares que
-- hay en el número natural n. Por ejemplo,
--    numeroDigitosPares      0  ==  1
--    numeroDigitosPares      1  ==  0
--    numeroDigitosPares    246  ==  3
--    numeroDigitosPares    135  ==  0
--    numeroDigitosPares 123456  ==  3
-- ------------------------------------------------------------------


numeroDigitosPares2 n
```

```
    | n < 10    = aux n
    | otherwise = (aux n `rem` 10) + numeroDigitosPares2 (n `div` 10)
    where aux n | even n    = 1
                | otherwise = 0
```

5.4. Exámenes del grupo 5 (Andrés Cordón y Miguel A. Martínez)

5.4.1. Examen 1 (5 de Noviembre de 2013)

```
-- Informática (1º del Grado en Matemáticas y en Física)
-- 1º examen de evaluación continua (4 de noviembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un año es bisiesto si, o bien es divisible por 4 pero no
-- por 100, o bien es divisible por 400. En cualquier otro caso, no lo
-- es.
--
-- Definir el predicado
--    bisiesto :: Int -> Bool
-- tal que (bisiesto a) se verifica si a es un año bisiesto. Por ejemplo:
--    bisiesto 2013 == False        bisiesto 2012 == True
--    bisiesto 1700 == False        bisiesto 1600 == True
-- ---------------------------------------------------------------------

-- 1ª definición:
bisiesto :: Int -> Bool
bisiesto x = (mod x 4 == 0 && mod x 100 /= 0) || mod x 400 == 0

-- 2ª definición (con guardas):
bisiesto2 :: Int -> Bool
bisiesto2 x | mod x 4 == 0 && mod x 100 /= 0 = True
            | mod x 400 == 0                 = True
            | otherwise                      = False

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    entre :: Int -> Int -> [Int]
-- tal que (entre a b) devuelve la lista de todos los bisiestos entre
```

```
-- los años a y b. Por ejemplo:
--    entre 2000 2019 == [2000,2004,2008,2012,2016]
-- ---------------------------------------------------------------------

entre :: Int -> Int -> [Int]
entre a b = [x | x <- [a..b], bisiesto x]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir el predicado
--    coprimos :: (Int,Int) -> Bool
-- tal que (coprimos (a,b)) se verifica si a y b son primos entre sí;
-- es decir, no tienen ningún factor primo en común. Por ejemplo,
--    coprimos (12,25) == True
--    coprimos (6,21)  == False
--    coprimos (1,5)   == True
-- Calcular todos los números de dos cifras coprimos con 30.
-- ---------------------------------------------------------------------

-- 1ª definición
coprimos :: (Int,Int) -> Bool
coprimos (a,b) = and [mod a x /= 0 | x <- factores b]
    where factores x = [z | z <- [2..x], mod x z == 0]

-- 2º definición
coprimos2 :: (Int,Int) -> Bool
coprimos2 (a,b) = gcd a b == 1

-- El cálculo es
--    ghci> [x | x <- [10..99], coprimos (x,30)]
--    [11,13,17,19,23,29,31,37,41,43,47,49,53,59,61,67,71,73,77,79,83,89,91,97]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    longCamino :: [(Float,Float)] -> Float
-- tal que (longCamino xs) es la longitud del camino determinado por los
-- puntos del plano listados en xs. Por ejemplo,
--    longCamino [(0,0),(1,0),(2,1),(2,0)] == 3.4142137
-- ---------------------------------------------------------------------

longCamino :: [(Float,Float)] -> Float
```

```
longCamino xs =
    sum [sqrt ((a-c)^2+(b-d)^2)| ((a,b),(c,d)) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Se quiere poner en marcha un nuevo servicio de correo
-- electrónico. Se requieren las siguientes condiciones para las
-- contraseñas: deben contener un mínimo de 8 caracteres, al menos deben
-- contener dos números, y al menos deben contener una letra
-- mayúscula. Se asume que el resto de caracteres son letras del
-- abecedario sin tildes.
--
-- Definir la función
--    claveValida :: String -> Bool
-- tal que (claveValida xs) indica si la contraseña es válida. Por
-- ejemplo,
--    claveValida "EstoNoVale" == False
--    claveValida "Tampoco7"   == False
--    claveValida "SiVale23"   == True
-- ---------------------------------------------------------------------

claveValida :: String -> Bool
claveValida xs =
    length xs >= 8 &&
    length [x | x <- xs, x `elem` ['0'..'9']] > 1 &&
    [x | x <- xs, x `elem` ['A'..'Z']] /= []

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    media :: [String] -> Float
-- tal que (media xs) es la media de las longitudes de las contraseñas
-- válidas de xs. Por ejemplo,
--    media ["EstoNoVale","Tampoco7","SiVale23","grAnada1982"] == 9.5
-- Indicación: Usar fromIntegral.
-- ---------------------------------------------------------------------

media :: [String] -> Float
media xss =
    fromIntegral (sum [length xs | xs <- validas]) / fromIntegral (length validas)
    where validas = [xs | xs <- xss, claveValida xs]
```

5.4.2. Examen 2 (16 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas y en Física)
-- 2º examen de evaluación continua (16 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Data.Char

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir las funciones
--    ultima, primera :: Int -> Int
-- que devuelven, respectivamente, la última y la primera cifra de un
-- entero positivo, Por ejemplo:
--    ultima  711 = 1
--    primera 711 = 7
-- ---------------------------------------------------------------------

ultima, primera :: Int -> Int
ultima n  = n `rem` 10
primera n = read [head (show n)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, el predicado
--    encadenadoR :: [Int] -> Bool
-- tal que (encadenadoR xs) se verifica si xs es una lista de
-- enteros positivos encadenados (es decir, la última cifra de cada
-- número coincide con la primera del siguiente en la lista). Por ejemplo:
--   encadenadoR [711,1024,413,367]  ==  True
--   encadenadoR [711,1024,213,367]  ==  False
-- ---------------------------------------------------------------------

encadenadoR :: [Int] -> Bool
encadenadoR (x:y:zs) = ultima x == primera y && encadenadoR (y:zs)
encadenadoR _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por comprensión, el predicado
--    encadenadoC :: [Int] -> Bool
-- tal que (encadenadoC xs) se verifica si xs es una lista de
-- enteros positivos encadenados (es decir, la última cifra de cada
-- número coincide con la primera del siguiente en la lista). Por ejemplo:
```

```
--   encadenadoC [711,1024,413,367]  ==  True
--   encadenadoC [711,1024,213,367]  ==  False
-- ---------------------------------------------------------------------

encadenadoC :: [Int] -> Bool
encadenadoC xs = and [ultima x == primera y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Un entero positivo se dirá semiperfecto si puede
-- obtenerse como la suma de un subconjunto de sus divisores propios (no
-- necesariamente todos). Por ejemplo, 18 es semiperfecto, pues sus
-- divisores propios son 1, 2, 3, 6 y 9 y además 18 = 1+2+6+9.
--
-- Define el predicado
--    semiperfecto :: Int -> Bool
-- tal que (semiperfecto x) se verifica si x es semiperfecto. Por
-- ejemplo,
--   semiperfecto 18  ==  True
--   semiperfecto 15  ==  False
-- ---------------------------------------------------------------------

semiperfecto :: Int -> Bool
semiperfecto n = not (null (sublistasConSuma (divisores n) n))

-- (divisores x) es la lista de los divisores propios de x. Por ejemplo,
--   divisores 18  ==  [1,2,3,6,9]
divisores :: Int -> [Int]
divisores x = [y | y <- [1..x-1], x `rem` y == 0]

-- (sublistasConSuma xs n) es la lista de las sublistas de la lista de
-- números naturales xs que suman n. Por ejemplo,
--   sublistasConSuma [1,2,3,6,9] 18  ==  [[1,2,6,9],[3,6,9]]
sublistasConSuma :: [Int] -> Int -> [[Int]]
sublistasConSuma [] 0 = [[]]
sublistasConSuma [] _ = []
sublistasConSuma (x:xs) n
    | x > n = sublistasConSuma xs n
    | otherwise = [x:ys | ys <- sublistasConSuma xs (n-x)] ++
                  sublistasConSuma xs n
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    semiperfectos :: Int -> [Int]
-- tal que (semiperfectos n) es la lista de los n primeros números
-- semiperfectos. Por ejemplo:
--    semiperfectos 10 == [6,12,18,20,24,28,30,36,40,42]
-- ---------------------------------------------------------------------

semiperfectos :: Int -> [Int]
semiperfectos n = take n [x | x <- [1..], semiperfecto x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Formatear una cadena de texto consiste en:
--    1. Eliminar los espacios en blanco iniciales.
--    2. Eliminar los espacios en blanco finales.
--    3. Reducir a 1 los espacios en blanco entre palabras.
--    4. Escribir la primera letra en mayúsculas, si no lo estuviera.
--
-- Definir la función
--    formateada :: String -> String
-- tal que (formateada cs) es la cadena cs formateada. Por ejemplo,
--    formateada "   la   palabra   precisa "  ==  "La palabra precisa"
-- ---------------------------------------------------------------------

formateada :: String -> String
formateada cs = toUpper x : xs
    where (x:xs) = unwords (words cs)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El centro de masas de un sistema discreto es el punto
-- geométrico que dinámicamente se comporta como si en él estuviera
-- aplicada la resultante de las fuerzas externas al sistema.
--
-- Representamos un conjunto de n masas en el plano mediante una lista
-- de n pares de la forma ((ai,bi),mi) donde (ai,bi) es la posición y mi
-- la masa puntual. Las coordenadas del centro de masas (a,b) se
-- calculan por
--    a = (a1*m1+a2*m2+ ... an*mn)/(m1+m2+...mn)
--    b = (b1*m1+b2*m2+ ... bn*mn)/(m1+m2+...mn)
--
```

```
-- Definir la función
--    masaTotal ::  [((Float,Float),Float)] -> Float
-- tal que (masaTotal xs) es la masa total de sistema xs. Por ejemplo,
--    masaTotal  [((-1,3),2),((0,0),5),((1,3),3)]  ==  10
-- ---------------------------------------------------------------------

masaTotal ::  [((Float,Float),Float)] -> Float
masaTotal xs = sum [m | (_,m) <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    centrodeMasas :: [((Float,Float),Float)] -> (Float,Float)
-- tal que (centrodeMasas xs) es las coordenadas del centro
-- de masas del sistema discreto xs. Por ejemplo:
--    centrodeMasas [((-1,3),2),((0,0),5),((1,3),3)] == (0.1,1.5)
-- ---------------------------------------------------------------------

centrodeMasas :: [((Float,Float),Float)] -> (Float,Float)
centrodeMasas xs =
    (sum [a*m | ((a,_),m) <- xs] / mt,
     sum [b*m | ((_,b),m) <- xs] / mt)
    where mt = masaTotal xs
```

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*.