Exámenes de "Programación funcional con Haskell" (2009–2015)

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, 25 de julio de 2016

Índice general

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 [1]. Sus implementaciones se encuentra en la página de los códigos[3]

El libro consta de 6 capítulos correspondientes a los 6 cursos en los que 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, en esta versión del libro hay 129 exámenes con un total de 833 ejercicios.

[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] https://github.com/jaalonso/I1M

José A. Alonso
Sevilla, 14 de septiembre de 2015

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" == "HHHHHooooolllllaaaaa"
-- ---------------------------------------------------------------------
```

```
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" == "HHHHHooooolllllaaaaa"
-- ---------------------------------------------------------------------

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 31).

1.2.4. Examen 4 (15 de septiembre de 2010)

El examen es común con el del grupo 1 (ver página 33).

1.2.5. Examen 5 (17 de diciembre de 2010)

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

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
-- estríctamente 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 85).

2.1.5. Examen 5 (14 de Marzo de 2011)

El examen es común con el del grupo 1 (ver página 87).

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 factores 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 98).

2.1.9. Examen 9 (8 de Julio de 2011)

El examen es común con el del grupo 1 (ver página 104).

2.1.10. Examen 10 (16 de Septiembre de 2011)

El examen es común con el del grupo 1 (ver página 111).

2.1.11. Examen 11 (22 de Noviembre de 2011)

El examen es común con el del grupo 1 (ver página 121).

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
```

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

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

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

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

data Expr = Num Int
          | Suma Expr Expr
          | X

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

-- ---------------------------------------------------------------------
-- 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 como suma de números
--    naturales consecutivos.
-- Definir la función
--    sucesionesConSuma :: Int -> [(Int,Int)]
-- tal que (sucesionesConSuma n) es la lista de las sucesiones de
-- números naturales consecutivos con suma n. Por ejemplo,
--    sucesionesConSuma 15  == [(1,5),(4,6),(7,8),(15,15)]
-- ya que 15 = 1+2+3+4+5 = 4+5+6 = 7+8 = 15.
--
-- Calcular la solución del problema usando sucesionesConSuma.
-- ---------------------------------------------------------------------

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

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

-- Otra defínción, 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)
```

```haskell
                     | 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 143).

3.2.9. Examen 9 (9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 148).

3.2.10. Examen 10 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 152).

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 143).

3.3.4. Examen 8 (9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 148).

3.3.5. Examen 9 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 152).

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 143).

3.4.9. Examen 9 (9 de Septiembre de 2012)

El examen es común con el del grupo 1 (ver página 148).

3.4.10. Examen 10 (10 de Diciembre de 2012)

El examen es común con el del grupo 1 (ver página 152).

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 253).

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
```

```haskell
-- 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
-- árbol a. Por ejemplo,
--    nodosInternos ej1  ==  [9,8,6]
```

```
--    nodosInternos ej2  ==  [9,7,9]
-- .....................................................................

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

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

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

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

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

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

-- (ramas a) es la lista de las ramas del árbol a. Por ejemplo,
--    ramas ej1  ==  [[9,8,3],[9,8,2],[9,6,4],[9,6,5]]
--    ramas ej2  ==  [[9,7,3],[9,7,2],[9,9,9],[9,9,7]]
ramas :: Arbol a -> [[a]]
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 268).

4.1.8. Examen 8 (13 de septiembre de 2013)

El examen es común con el del grupo 2 (ver página 275).

4.1.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 2 (ver página 279).

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)
--    999999999999999999999999999999999999999999999999999999999999

-- ---------------------------------------------------------------------
-- 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 253).

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 268).

4.3.8. Examen 8 (13 de septiembre de 2013)

El examen es común con el del grupo 2 (ver página 275).

4.3.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 2 (ver página 279).

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 253).

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]
```

```haskell
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 242).

4.4.7. Examen 7 (3 de julio de 2013)

El examen es común con el del grupo 2 (ver página 268).

4.4.8. Examen 8 (13 de septiembre de 2013)

El examen es común con el del grupo 2 (ver página 275).

4.4.9. Examen 9 (20 de noviembre de 2013)

El examen es común con el del grupo 2 (ver página 279).

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]  == 29160
-- ---------------------------------------------------------------------

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 x y zs) es la lista obtenida colocando y delante y
-- detrás de todos los elementos de zs que coinciden con x. 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 x y (z:zs) | z == x    = y : z : y : bocata x y zs
                  | otherwise = z : bocata x y zs

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

5.1.3. Examen 3 (23 de Enero de 2014)

```
-- Informática: 3º examen de evaluación continua (23 de enero de 2014)
-- ---------------------------------------------------------------------

-- Puntuación: Cada uno de los 4 ejercicios vale 2.5 puntos.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    divisoresConUno :: Integer -> Bool
-- tal que (divisoresConUno n) se verifica si todos sus divisores
```

```
-- contienen el dígito 1. Por ejemplo,
--    divisoresConUno 671  ==  True
--    divisoresConUno 100  ==  False
-- ya que los divisores de 671 son 1, 11, 61 y 671 y todos contienen el
-- número 1; en cambio, 25 es un divisor de 100 que no contiene el
-- dígito 1.
-- ---------------------------------------------------------------------

divisoresConUno :: Integer -> Bool
divisoresConUno n = all contieneUno (divisores n)

-- 2ª definición (sin all)
divisoresConUno2 :: Integer -> Bool
divisoresConUno2 n = and [contieneUno x | x <- divisores n]

-- 3ª definición (por recursión)
divisoresConUno3 :: Integer -> Bool
divisoresConUno3 n = aux (divisores n)
    where aux []     = True
          aux (x:xs) = contieneUno x && aux xs

-- 4ª definición (por plegado)
divisoresConUno4 :: Integer -> Bool
divisoresConUno4 n = foldr f True (divisores n)
    where f x y = contieneUno x && y

-- 5ª definición (por plegado y lambda)
divisoresConUno5 :: Integer -> Bool
divisoresConUno5 n =
    foldr (\x y -> contieneUno x && y) True (divisores n)

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

-- (contienUno n) se verifica si n contiene el dígito 1. Por ejemplo,
--    contieneUno 214  ==  True
--    contieneUno 234  ==  False
contieneUno :: Integer -> Bool
```

```
contieneUno n = elem '1' (show n)

-- 2ª definición (por recursión sin show)
contieneUno2 :: Integer -> Bool
contieneUno2 1 = True
contieneUno2 n | n < 10          = False
               | n `rem` 10 == 1 = True
               | otherwise       = contieneUno2 (n `div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. ¿Cuál será el próximo año en el que todos sus divisores
-- contienen el dígito 1? ¿y el anterior?
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [2014..], divisoresConUno n]
--    2017
--    ghci> head [n | n <- [2014,2013..], divisoresConUno n]
--    2011

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Un elemento de una lista es permanente si ninguno de
-- los siguientes es mayor que él.
--
-- Definir, por recursión, la función
--    permanentesR :: [Int] -> [Int]
-- tal que (permanentesR xs) es la lista de los elementos permanentes de
-- xs. Por ejemplo,
--    permanentesR [80,1,7,8,4]  ==  [80,8,4]
-- ---------------------------------------------------------------------

-- 1ª definición:
permanentesR :: [Int] -> [Int]
permanentesR [] = []
permanentesR (x:xs) | x == maximum (x:xs) = x:permanentesR xs
                    | otherwise           = permanentesR xs

-- 2ª definición (sin usar maximum):
permanentesR2 :: [Int] -> [Int]
permanentesR2 [] = []
```

```
permanentesR2 (x:xs) | and [x>=y|y<-xs] = x:permanentesR2 xs
                     | otherwise        = permanentesR2 xs

-- Nota: Comparación de eficiencia
--    ghci> let xs = [1..1000] in last (permanentesR (xs ++ reverse xs))
--    1
--    (0.22 secs, 41105812 bytes)
--    ghci> let xs = [1..1000] in last (permanentesR2 (xs ++ reverse xs))
--    1
--    (0.96 secs, 31421308 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por plegado, la función
--    permanentesP :: [Int] -> [Int]
-- tal que (permanentesP xs) es la lista de los elementos permanentes de
-- xs. Por ejemplo,
--    permanentesP [80,1,7,8,4]  ==  [80,8,4]
-- ---------------------------------------------------------------------

-- 1ª definición:
permanentesP :: [Int] -> [Int]
permanentesP = foldr f []
    where f x ys | x == maximum (x:ys) = x:ys
                 | otherwise           = ys

-- 2ª definición:
permanentesP2 :: [Int] -> [Int]
permanentesP2 xs = foldl f [] (reverse xs)
    where f ac x | x == maximum (x:ac) = x:ac
                 | otherwise           = ac

-- Nota: Comparación de eficiencia
--    ghci> let xs = [1..1000] in last (permanentesP (xs ++ reverse xs))
--    1
--    (0.22 secs, 52622056 bytes)
--    ghci> let xs = [1..1000] in last (permanentesP2 (xs ++ reverse xs))
--    1
--    (0.23 secs, 52918324 bytes)

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

```
-- Ejercicio 3. Definir la función
--    especial :: Int -> [[Int]] -> Bool
-- tal que (especial k xss) se verifica si cada uno de los diez dígitos
-- 0, 1, 2,..., 9 aparece k veces entre todas las listas de xss. Por
-- ejemplo,
--    especial 1 [[12,40],[5,79,86,3]]                          == True
--    especial 2 [[107,32,89],[58,76,94],[63,120,45]]           == True
--    especial 3 [[1329,276,996],[534,867,1200],[738,1458,405]] == True
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
especial :: Int -> [[Int]] -> Bool
especial k xss =
    sort (concat [show n | xs <- xss, n <- xs])
    == concat [replicate k d | d <- ['0'..'9']]

-- 2ª definición (con map)
especial2 :: Int -> [[Int]] -> Bool
especial2 k xss =
    sort (concat (concat (map (map cifras) xss)))
    == concat [replicate k d | d <- [0..9]]

cifras:: Int -> [Int]
cifras n = [read [x] | x <-show n]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    primosConsecutivosConIgualFinal :: Int -> [Integer]
-- tal que (primosConsecutivosConIgualFinal n) es la lista de los
-- primeros n primos consecutivos que terminan en el  mismo dígito. Por
-- ejemplo,
--    primosConsecutivosConIgualFinal 2 == [139, 149]
--    primosConsecutivosConIgualFinal 3 == [1627, 1637, 1657]
-- ---------------------------------------------------------------------

primosConsecutivosConIgualFinal :: Int -> [Integer]
primosConsecutivosConIgualFinal n = consecutivosConPropiedad p n primos
    where p []     = True
          p (x:xs) = and [r == rem y 10 | y <- xs]
              where r = rem x 10
```

```
-- (consecutivosConPropiedad p n xs) es la lista con los n primeros
-- elementos consecutivos de zs que verifican la propiedad p. Por
-- ejemplo,
--    ghci> consecutivosConPropiedad (\xs -> sum xs > 20) 2 [5,2,1,17,4,25]
--    [17,4]
consecutivosConPropiedad :: ([a] -> Bool) -> Int -> [a] -> [a]
consecutivosConPropiedad p n zs =
    head [xs | xs <- [take n ys | ys <- tails zs], p xs]

-- primos es la lista de los números primos. Por ejemplo,
--    ghci> take 20 primos
--    [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
primos :: [Integer]
primos = [n | n <- 2:[3,5..], primo n]

-- (primo n) se verifica si n es un número primo. Por ejemplo,
--    primo 7  ==  True
--    primo 8  ==  False
primo :: Integer -> Bool
primo n = [x | x <- [1..n], rem n x == 0] == [1,n]
```

5.1.4. Examen 4 (20 de Marzo de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (20 de marzo de 2014)
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Consideremos la sucesión siguiente
--     a0 = 1
--     a1 = 1
--     an = 7*a(n-1) - a(n-2) - 2
--
```

```
-- Definir, por recursión, la función
--     suc :: Integer -> Integer
-- tal que (suc n) es el n-ésimo término de la sucesión anterior. Por
-- ejemplo,
--     suc 1 == 1
--     suc 4 == 169
--     suc 8 == 372100
-- Por ejemplo,
--    ghci> [suc n | n <-[0..10]]
--    [1,1,4,25,169,1156,7921,54289,372100,2550409,17480761]
-- ---------------------------------------------------------------------

suc :: Integer -> Integer
suc 0 = 1
suc 1 = 1
suc n = 7*(suc (n-1)) - (suc (n-2)) - 2

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, usando evaluación perezosa. la función
--     suc' :: Integer -> Integer
-- tal que (suc n) es el n-ésimo término de la sucesión anterior. Por
-- ejemplo,
--     suc 1 == 1
--     suc 4 == 169
--     suc 8 == 372100
-- Por ejemplo,
--    ghci> [suc n | n <-[0..10]]
--    [1,1,4,25,169,1156,7921,54289,372100,2550409,17480761]
--    ghci> suc' 30
--    915317035111995882133681
-- ---------------------------------------------------------------------

-- La sucesión es
sucesion:: [Integer]
sucesion = 1:1:zipWith f (tail sucesion) sucesion
    where f x y = 7*x-y-2

-- Por ejemplo, el cálculo de los 4 primeros términos es
--    take 4 sucesion
--    = take 4 (1:1:zipWith f (tail sucesion) sucesion)
```

```
--    = 1:take 3 (1:zipWith f (tail sucesion) sucesion)
--    = 1:1:take 2 (zipWith f (tail sucesion) sucesion)
--    = 1:1:take 2 (zipWith f (1:R2) (1:1:R2))
--    = 1:1:take 2 (4:zipWith f R2 (1:R2))
--    = 1:1:4:take 1 (zipWith f (4:R3) (1:4:R3))
--    = 1:1:4:take 1 (25:zipWith f R3 (4:R3))
--    = 1:1:4:25:take 0 (25:zipWith f R3 (4:R3))
--    = 1:1:4:25:[]
--    = [1,1,4,25]

suc' :: Integer -> Integer
suc' n = sucesion `genericIndex` n

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Calcular el término 100 de la sucesión anterior.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> suc' 100
--    300684343490825938802118062949475967529205466257891810825055230703212868070465

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Comprobar que los primeros 30 términos de la sucesión
-- son cuadrados perfectos.
-- ---------------------------------------------------------------------

esCuadrado :: (Integral a) => a -> Bool
esCuadrado n = y*y == n
    where y = floor (sqrt (fromIntegral n))

-- La comprobación es
--    ghci> and [esCuadrado (suc' n) | n <-[0..30]]
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 2. Consideremos los árboles binarios definidos por el tipo
-- siguiente:
--    data Arbol t = Hoja t
--                 | Nodo (Arbol t) t (Arbol t)
--                 deriving (Show, Eq)
```

```
-- y el siguiente ejemplo de árbol
--    ejArbol :: Arbol Int
--    ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
--                   5
--                   (Nodo (Hoja 6) 7 (Hoja 9))
--
-- Definir la función
--    transforma :: (t -> Bool) -> Arbol t -> Arbol (Maybe t)
-- tal que (transforma p a) es el árbol con la misma estructura que a,
-- en el que cada elemento x que verifica el predicado p se sustituye por
-- (Just x) y los que no lo verifican se sustituyen por Nothing. Por
-- ejemplo,
--    ghci> transforma even ejArbol
--    Nodo (Nodo (Hoja Nothing) Nothing (Hoja (Just 4)))
--         Nothing
--         (Nodo (Hoja (Just 6)) Nothing (Hoja Nothing))
-- ---------------------------------------------------------------------

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

ejArbol :: Arbol Int
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. 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 :: (Double -> Double) ->
--                      Double -> Double -> Double -> Double
-- tal que (ceroBiseccionE f a b e) calcule 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. Por ejemplo,
-- si f1 y f2 son las funciones definidas por
--    f1 x = 2 - x
--    f2 x = x^2 - 3
-- entonces
--    ceroBiseccionE f1 0 3 0.0001     == 2.00006103515625
--    ceroBiseccionE f2 0 2 0.0001     == 1.7320556640625
--    ceroBiseccionE f2 (-2) 2 0.00001 == -1.732048
--    ceroBiseccionE cos 0 2 0.0001    == -1.7320480346679688
-- ---------------------------------------------------------------------

f1 x = 2 - x
f2 x = x^2 - 3

ceroBiseccionE :: (Double -> Double) ->
                  Double -> Double -> Double -> Double
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 4.1. Los polinomios de Fibonacci se definen como sigue
```

```
--    P_0 = 0
--    P_1 = 1
--    P_n = x*P_(n-1) + P_(n-2)
--
-- Definir la función
--    polFibonnaci :: Integer -> Polinomio Rational
-- tal que (polFibonnaci n) es el n-ésimo polinomio de Fibonacci. Por
-- ejemplo,
--    polFibonnaci 2 == 1 % 1*x
--    polFibonnaci 3 == x^2 + 1 % 1
--    polFibonnaci 4 == x^3 + 2 % 1*x
--    polFibonnaci 5 == x^4 + 3 % 1*x^2 + 1 % 1
-- ---------------------------------------------------------------------

-- 1ª solución (por recursión)
polFibonnaci :: Integer -> Polinomio Rational
polFibonnaci 0 = polCero
polFibonnaci 1 = polUnidad
polFibonnaci n =
    sumaPol (multPol (creaPolDispersa [1,0]) (polFibonnaci (n-1)))
            (polFibonnaci (n-2))

-- 2ª solución (evaluación perezosa)
polFibonnaciP :: Integer -> Polinomio Rational
polFibonnaciP n = sucPolinomiosFibonacci `genericIndex` n

sucPolinomiosFibonacci :: [Polinomio Rational]
sucPolinomiosFibonacci =
    polCero:polUnidad:zipWith f (tail sucPolinomiosFibonacci)
                                sucPolinomiosFibonacci
        where f p q = sumaPol (multPol (creaPolDispersa [1,0]) p) q

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar que P_2 divide a los polinomios de Fibonacci
-- de índice par hasta n = 20.
-- ---------------------------------------------------------------------

divide :: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Bool
divide p q = esPolCero (resto q p)
```

```
-- La comprobación es
-- ghci> and [divide (polFibonnaciP 2) (polFibonnaciP (2*k)) | k <- [2..20]]
-- True
```

5.1.5. Examen 5 (15 de Mayo de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (15 de mayo de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--     separados :: Eq a => a -> a -> [a] -> Bool
-- tal que (separados x y xs) se verifica si a y b no son elementos
-- consecutivos en xs. Por ejemplo,
--    separados 4 6 [1..20]      == True
--    separados 4 6 [2,4..20]    == False
--    separados 'd' 'a' "damas"  == False
--    separados 'd' 'a' "ademas" == False
--    separados 'd' 'm' "ademas" == True
-- ---------------------------------------------------------------------

-- 1ª solución
separados :: Eq a => a -> a -> [a] -> Bool
separados a b zs = and [(x,y) /= (a,b) && (x,y) /= (b,a) |
                        (x,y) <- zip zs (tail zs)]

-- 2ª solución
separados2 :: Eq a => a -> a -> [a] -> Bool
separados2 a b zs =
    (a,b) `notElem` consecutivos && (b,a) `notElem` consecutivos
    where consecutivos = zip zs (tail zs)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    subcadenasNoVacias :: [a] -> [[a]]
-- tal que (subcadenasNoVacias xs) es la lista de las subcadenas no
-- nulas de xs. Por ejemplo,
--    ghci> subcadenasNoVacias "Hola"
--    ["H","Ho","Hol","Hola","o","ol","ola","l","la","a"]
-- ---------------------------------------------------------------------

-- 1ª solución
subcadenasNoVacias :: [a] -> [[a]]
subcadenasNoVacias []     = []
subcadenasNoVacias (x:xs) = tail (inits (x:xs)) ++ subcadenasNoVacias xs

-- 2ª solución
subcadenasNoVacias2 :: [a] -> [[a]]
subcadenasNoVacias2 xs =
    [take i (drop j xs) | j <- [0..n], i <- [1..n-j]]
    where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Partiendo de un número d, se construye la sucesión que
-- empieza en d y cada término se obtiene sumándole al anterior el
-- producto de sus dígitos no nulos. Por ejemplo:
-- * Si empieza en 1, la sucesión es 1,2,4,8,16,22,26,38,62,74,...
-- * Si empieza en 30, la sucesión es 30,33,42,50,55,80,88,152,162,174,...
--
-- Definir la función
--    sucesion :: Integer -> [Integer]
-- tal que (sucesion d) es la sucesión que empieza en d. Por ejemplo,
--    ghci> take 10 (sucesion 1)
--    [1,2,4,8,16,22,26,38,62,74]
--    ghci> take 10 (sucesion 3)
--    [3,6,12,14,18,26,38,62,74,102]
--    ghci> take 10 (sucesion 30)
--    [30,33,42,50,55,80,88,152,162,174]
--    ghci> take 10 (sucesion 10)
--    [10,11,12,14,18,26,38,62,74,102]
-- ---------------------------------------------------------------------
```

```
-- 1ª definición
sucesion :: Integer -> [Integer]
sucesion d = iterate f d
    where f x = x + productoDigitosNN x

-- (productoDigitosNN x) es el producto de los dígitos no nulos de
-- x. Por ejemplo,
--    productoDigitosNN 306  ==  18
productoDigitosNN :: Integer -> Integer
productoDigitosNN = product . digitosNoNulos

-- (digitosNoNulos x) es la lista de los dígitos no nulos de x. Por
-- ejemplo,
--    digitosNoNulos 306  ==  [3,6]
digitosNoNulos:: Integer -> [Integer]
digitosNoNulos n = [read [x] | x <- show n, x /= '0']

-- 2ª definición
sucesion2 :: Integer -> [Integer]
sucesion2 d = [aux n | n <- [0..]]
    where aux 0 = d
          aux n = x + productoDigitosNN x
              where x = aux (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Las sucesiones así construidas tienen un elemento
-- común, a partir del cual los términos coinciden. Por ejemplo,
--    take 7 (sucesion 3)  ==  [3,6,    12,14,18,26,38]
--    take 7 (sucesion 5)  ==  [5,10,11,12,14,18,26]
-- se observa que las sucesiones que empiezan en 3 y 5, respectivamente,
-- coinciden a partir del término 12.
--
-- Definir la función
--     comun :: Integer -> Integer -> Integer
-- tal que (comun x y) es el primer elemento común de las sucesiones que
-- empiezan en x e y, respectivamente. Por ejemplo,
--     comun 3 5     == 12
--     comun 3 4     == 26
--     comun 3 8     == 26
```

```
--     comun 3 20    == 26
--     comun 3 34    == 126
--     comun 234 567 == 1474
-- ---------------------------------------------------------------------

comun :: Integer -> Integer -> Integer
comun x y =
    head [n | n <- sucesion x, n `elem` takeWhile (<=n) (sucesion y)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--     indicesComun :: Integer -> Integer -> (Integer, Integer)
-- tal que (indicesComun x y) calcula los índices a partir de los cuales
-- las sucesiones con valores iniciales x e y coinciden. Por ejemplo,
--     indicesComun 3 4     == (6,5)
--     indicesComun 3 5     == (3,4)
--     indicesComun 3 8     == (6,4)
--     indicesComun 3 20    == (6,3)
--     indicesComun 3 34    == (15,5)
--     indicesComun 234 567 == (16,19)
-- ---------------------------------------------------------------------

indicesComun :: Integer -> Integer -> (Integer, Integer)
indicesComun x y = (i,j)
    where z = comun x y
          i = head [k | (a,k) <- zip (sucesion x) [1..], a == z]
          j = head [k | (a,k) <- zip (sucesion y) [1..], a == z]


-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los árboles binarios definidos por
--    data Arbol a = Hoja | Nodo a (Arbol a) (Arbol a)
--                   deriving Show
--
-- Por ejemplo, el árbol
--            5
--           / \
--          /   \
--         4     7
--        / \   / \
```

```
--       1         8
--     /  \      /  \
-- se representa por
--    arbol1 = Nodo 5 (Nodo 4 (Nodo 1 Hoja Hoja) Hoja )
--                    (Nodo 7 Hoja (Nodo 8 Hoja Hoja))
--
-- Definir la función
--    takeArbolWhile :: (a -> Bool) -> Arbol a -> Arbol a
-- tal que (takeArbolWhile p ar) es el subárbol de ar empezando desde la
-- raiz mientras se verifique p. Por ejemplo,
--   takeArbolWhile odd arbol1   == Nodo 5 Hoja (Nodo 7 Hoja Hoja)
--   takeArbolWhile even arbol1  == Hoja
--   takeArbolWhile (< 6) arbol1 == Nodo 5 (Nodo 4 (Nodo 1 Hoja Hoja) Hoja) Hoja
-- ---------------------------------------------------------------------

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

arbol1 = Nodo 5 (Nodo 4 (Nodo 1 Hoja Hoja) Hoja )
                (Nodo 7 Hoja (Nodo 8 Hoja Hoja))

takeArbolWhile :: (a -> Bool) -> Arbol a -> Arbol a
takeArbolWhile p Hoja = Hoja
takeArbolWhile p (Nodo a x y)
    | p a       = Nodo a (takeArbolWhile p x) (takeArbolWhile p y)
    | otherwise = Hoja

-- ---------------------------------------------------------------------
-- Ejercicio 5. 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
-- Por ejemplo,
--    c1, c2:: Matriz Double
--    c1 = listArray ((1,1),(4,4)) [1,3,0,0,
--                                 -1, 1,-1, 1,
--                                 1,-1, 1,-1,
--                                 1, 1,-1, 1]
--
```

```
--    c2 = listArray ((1,1),(2,2)) [1,1,1,-1]
--
-- Definir la función
--     determinante:: Matriz Double -> Double
-- tal que (determinante p) es el determinante de la matriz p,
-- desarrollándolo por los elementos de una fila. Por ejemplo,
--     determinante c1 == 0.0
--     determinante c2 == -2.0
-- ---------------------------------------------------------------------

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

c1, c2:: Matriz Double
c1 = listArray ((1,1),(4,4)) [1,3,0,0,
                              -1, 1,-1, 1,
                              1,-1, 1,-1,
                              1, 1,-1, 1]

c2 = listArray ((1,1),(2,2)) [1,1,1,-1]

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,
--    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)]
--    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)
```

5.1.6. Examen 6 (18 de Junio de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (18 de junio de 2014)
-- ---------------------------------------------------------------------

-- Librerías auxiliares
-- ====================
import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1 [2 puntos]. Definir la función
--     siembra :: [a] -> [[a]] -> [[a]]
-- tal que (siembra xs yss) es la lista obtenida introduciendo cada uno
-- de los elementos de xs en la lista correspondiente de yss; es decir,
-- el primer elemento de xs en la primera lista de yss, el segundo
-- elemento de xs en la segunda lista de yss, etc. Por ejemplo,
--    siembra [1,2,3] [[4,7],[6],[9,5,8]]  ==  [[1,4,7],[2,6],[3,9,5,8]]
--    siembra [1,2] [[4,7],[6],[9,5,8]]    ==  [[1,4,7],[2,6],[9,5,8]]
--    siembra [1,2,3] [[4,7],[6]]          ==  [[1,4,7],[2,6]]
-- ---------------------------------------------------------------------

siembra :: [a] -> [[a]] -> [[a]]
siembra [] yss          = yss
siembra xs []           = []
siembra (x:xs) (ys:yss) = (x:ys) : siembra xs yss

-- ---------------------------------------------------------------------
-- Ejercicio 2 [2 puntos]. Definir la función
--    primosEquidistantes :: Integer -> [(Integer,Integer)]
-- tal que (primosEquidistantes k) es la lista de los pares de primos
-- consecutivos cuya diferencia es k. Por ejemplo,
--    take 3 (primosEquidistantes 2)  ==  [(3,5),(5,7),(11,13)]
--    take 3 (primosEquidistantes 4)  ==  [(7,11),(13,17),(19,23)]
--    take 3 (primosEquidistantes 6)  ==  [(23,29),(31,37),(47,53)]
```

```
--    take 3 (primosEquidistantes 8)  ==  [(89,97),(359,367),(389,397)]
-- ---------------------------------------------------------------------

primosEquidistantes :: Integer -> [(Integer,Integer)]
primosEquidistantes k = aux primos
    where aux (x:y:ps) | y - x == k = (x,y) : aux (y:ps)
                       | otherwise  = aux (y:ps)

-- (primo x) se verifica si x es primo. Por ejemplo,
--    primo 7  ==  True
--    primo 8  ==  False
primo :: Integer -> Bool
primo x = [y | y <- [1..x], x `rem` y == 0] == [1,x]

-- 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 : [x | x <- [3,5..], primo x]

-- ---------------------------------------------------------------------
-- Ejercicio 3 [2 puntos]. Se consideran los árboles con operaciones
-- booleanas definidos por
--    data ArbolB = H Bool
--                | Conj ArbolB ArbolB
--                | Disy ArbolB ArbolB
--                | Neg ArbolB
--
-- Por ejemplo, los árboles
--                Conj                            Conj
--               /   \                           /   \
--              /     \                         /     \
--           Disy      Conj                  Disy      Conj
--          /   \       /  \                /   \      /   \
--       Conj    Neg   Neg True          Conj    Neg   Neg  True
--       /  \    |     |                 /  \    |     |
--    True False False False          True False True  False
--
-- se definen por
--    ej1, ej2:: ArbolB
--    ej1 = Conj (Disy (Conj (H True) (H False))
```

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

data ArbolB = H Bool
            | Conj ArbolB ArbolB
            | Disy ArbolB ArbolB
            | Neg ArbolB

ej1, ej2:: ArbolB
ej1 = Conj (Disy (Conj (H True) (H False))
                 (Neg (H False)))
           (Conj (Neg (H False))
                 (H True))

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

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

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

```
-- Ejercicio 4 [2 puntos]. La matriz de Vandermonde generada por
-- [a(1),a(2),a(3),...,a(n)] es la siguiente
--     |1  a(1)  a(1)^2 ... a(1)^{n-1}|
--     |1  a(2)  a(2)^2 ... a(2)^{n-1}|
--     |1  a(3)  a(3)^2 ... a(3)^{n-1}|
--     |.  .     .          .         |
--     |.  .     .          .         |
--     |.  .     .          .         |
--     |1  a(n)  a(n)^2 ... a(n)^{n-1}|
--
-- Las matrices se representan con tablas cuyos índices son pares de
-- números naturales.
--    type Matriz a = Array (Int,Int) a
--
-- Definir la función
--    vandermonde:: [Integer] -> Matriz Integer
-- tal que (vandermonde xs) es la matriz de Vandermonde cuyos
-- generadores son los elementos de xs. Por ejemplo,
--    ghci> vandermonde [5,2,3,4]
--    array ((1,1),(4,4)) [((1,1),1),((1,2),5),((1,3),25),((1,4),125),
--                         ((2,1),1),((2,2),2),((2,3), 4),((2,4),  8),
--                         ((3,1),1),((3,2),3),((3,3), 9),((3,4), 27),
--                         ((4,1),1),((4,2),4),((4,3),16),((4,4), 64)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

-- 1ª solución
-- ===========

vandermonde1 :: [Integer] -> Matriz Integer
vandermonde1 xs = array ((1,1), (n,n))
                  [((i,j), f i j) | i <- [1..n], j <- [1..n]]
      where n     = length xs
            f i j = (xs!!(i-1))^(j-1)

-- 2ª solución
-- ===========

vandermonde2 :: [Integer] -> Matriz Integer
```

```
vandermonde2 xs = listArray ((1,1),(n,n)) (concat (listaVandermonde xs))
    where n = length xs

-- (listaVandermonde xs) es la lista correspondiente a la matriz de
-- Vandermonde generada por xs. Por ejemplo,
--    ghci> listaVandermonde [5,2,3,4]
--    [[1,5,25,125],[1,2,4,8],[1,3,9,27],[1,4,16,64]]
listaVandermonde :: [Integer] -> [[Integer]]
listaVandermonde xs = [[x^i | i <- [0..n-1]] | x <- xs]
    where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 5 [2 puntos]. El número 595 es palíndromo y, además, es
-- suma de cuadrados consecutivos, pues
--    595 = 6^2 + 7^2 + 8^2 + 9^2 + 10^2 + 11^2 + 12^2.
--
-- Definir la función
--    sucesion:: [Integer]
-- tal que sucesion es la lista de los números que son palíndromos y
-- suma de cuadrados consecutivos. Por ejemplo,
--    take 10 sucesion == [1,4,5,9,55,77,121,181,313,434]
--    take 15 sucesion == [1,4,5,9,55,77,121,181,313,434,484,505,545,595,636]
-- ---------------------------------------------------------------------

sucesion:: [Integer]
sucesion = [x | x <-[1..], palindromo x, esSumaCuadradosConsecutivos x]

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

sucSumaCuadradosDesde :: Integer -> [Integer]
sucSumaCuadradosDesde k = scanl (\s n -> s + n^2) 0 [k..]


esSumaCuadradosConsecutivos n =
    or [pertenece n (sucSumaCuadradosDesde k) | k <- [1..m]]
    where pertenece x xs = elem x (takeWhile (<=x) xs)
          m              = floor (sqrt (fromIntegral n))

-- 2ª solución para esSumaCuadradosConsecutivos:
```

```
esSumaCuadradosConsecutivos2 n = any (==n) (map sum yss)
    where m = floor (sqrt (fromIntegral n))
          xss = segmentos [1..m]
          yss = map (map (^2)) xss

segmentos :: [a] -> [[a]]
segmentos xs = concat [tail (inits ys) | ys <- init (tails xs)]

sucesion2:: [Integer]
sucesion2 = [x | x <-[1..], palindromo x, esSumaCuadradosConsecutivos2 x]
```

5.1.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 401).

5.1.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 408).

5.1.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 413).

5.2. Exámenes del grupo 2 (Antonia M. Chávez)

5.2.1. Examen 1 (6 de Noviembre de 2013)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Se dice que dos números son hermanos si tienen el
-- mismo número de divisores propios. Por ejemplo, 6 y 22 son hermanos
-- porque ambos tienen tres divisores propios.
--
-- Definir la función hermanos tal que (hermanos x y) se verifica si x e
-- y son hermanos. Por ejemplo,
```

```
--    hermanos 6 10 == True
--    hermanos 3 4  == False
-- ---------------------------------------------------------------------

hermanos x y = length (divisoresProp x) == length (divisoresProp y)

divisoresProp x = [y | y <- [1 .. x-1], mod x y == 0]

--- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función hermanosHasta tal que
-- (hermanosHasta n) es la lista de los pares de números hermanos
-- menores o iguales a n.  Por ejemplo,
--    hermanosHasta 4  ==  [(1,1),(2,2),(2,3),(3,2),(3,3),(4,4)]
-- ---------------------------------------------------------------------

hermanosHasta n = [(x,y) | x <- [1 .. n], y <- [1 .. n], hermanos x y]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la propiedad prop_hermanos1 tal que
-- (prop_hermanos1 x y) se verifica si se cumple que x es hermano de y
-- si, y sólo si, y es hermano de x.
-- ---------------------------------------------------------------------

prop_hermanos1 x y = hermanos x y == hermanos y x

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la propiedad prop_hermanos2 tal
-- (prop_hermanos2 x) se verifica si x es hermano de sí mismo.
-- ---------------------------------------------------------------------

prop_hermanos2 x = hermanos x x

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Definir la función primerosHermanos tal que
-- (primerosHermanos k) es la lista de los primeros k pares de números
-- hermanos tales que el primero es menor que el segundo. Por ejemplo,
--    ghci> primerosHermanos 10
--    [(2,3),(2,5),(3,5),(2,7),(3,7),(5,7),(6,8),(4,9),(6,10),(8,10)]
-- ---------------------------------------------------------------------
```

```
primerosHermanos k =
    take k [(x,y) | y <- [1..], x <- [1..y-1], hermanos x y]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función superDiv tal que (superDiv n) es la
-- lista de listas que contienen los divisores de cada uno de los
-- divisores de n. Por ejemplo,
--    superDiv 10 == [[1],[1,2],[1,5],[1,2,5,10]]
--    superDiv 12 == [[1],[1,2],[1,3],[1,2,4],[1,2,3,6],[1,2,3,4,6,12]]
-- ---------------------------------------------------------------------

divisores n = [x | x <- [1..n], mod n x == 0]

superDiv n = [divisores x | x <- divisores n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir una funcion noPrimos tal que (noPrimos n)
-- es la lista que resulta de sustituir cada elemento de (superDiv n)
-- por el número de números no primos que contiene. Por ejemplo.
--    noPrimos 10 == [1,1,1,2]
--    noPrimos 12 == [1,1,1,2,2,4]
-- ---------------------------------------------------------------------

noPrimos n =
    [length [x | x <- xs, not (primo x)] | xs <- superDiv n]

primo n = divisores n == [1,n]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una lista es genial si la diferencia en valor absoluto
-- entre cualesquiera dos términos consecutivos es siempre mayor o igual
-- que la posición del primero de los dos. Por ejemplo, [1,3,-4,1] es
-- genial ya que
--    |1-3|    = 2 >= 0 = posición del 1,
--    |3-(-4)| = 7 >= 1 = posición del 3,
--    |(-4)-1| = 5 >= 2 = posición del -4.
-- en cambio, [1,3,0,1,2] no es genial ya que
--    |1-0| = 1 < 2 = posición del 1.
--
-- Definir por comprensión la función genial tal que (genial xs) se
```

```
-- verifica si xs es una lista genial. Por ejemplo,
--    genial [1,3,-4,1]  ==  True
--    genial [1,3,0,1,2]  ==  False
-- ---------------------------------------------------------------------

genial :: [Int] -> Bool
genial xs =
    and [abs (x-y) >= n | ((x,y),n) <- zip (zip xs (tail xs)) [0..]]

-- 2ª definición:
genial2 :: [Int] -> Bool
genial2 xs =
    and [abs (x-y) >= n | (x,y,n) <- zip3 xs (tail xs) [0..]]
```

5.2.2. Examen 2 (4 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (4 de diciembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, sin usar recursión, la función intro tal que
-- (intro x n xs) es la lista que resulta de introducir x en el lugar n
-- de la lista xs. Si n es  negativo, x se introducirá como primer
-- elemento; si n supera la longitud de xs, x aparecerá al final de
-- xs. Por ejemplo,
--    intro 5 1 [1,2,3]    == [1,5,2,3]
--    intro 'd' 2 "deo"    == "dedo"
--    intro 5 (-3) [1,2,3] == [5,1,2,3]
--    intro 5 10 [1,2,3]   == [1,2,3,5]
-- ---------------------------------------------------------------------

intro x n xs = take n xs ++ [x] ++ drop n xs
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión, la función introR tal que sea
-- equivalenta a la función intro del ejercicio anterior.
-- ---------------------------------------------------------------------

introR x n [] = [x]
introR x n (y:xs) | n <= 0            = x:y:xs
                  | n > length (y:xs) = (y:xs) ++ [x]
                  | otherwise         = y : introR x (n-1) xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función primerosYultimos tal que
-- (primerosYultimos xss) es el par formado  por la lista de los
-- primeros elementos de las listas no vacías de xss y la lista de los
-- últimos elementos de las listas no vacías de xs. Por ejemplo,
--    ghci> primerosYultimos [[1,2],[5,3,4],[],[0,8,7,6],[],[9]]
--    ([1,5,0,9],[2,4,6,9])
-- ---------------------------------------------------------------------

primerosYultimos xss =
    ([head xs | xs <- xss, not (null xs)],
     [last xs | xs <- xss, not (null xs)])

-- ---------------------------------------------------------------------
-- Ejercicio 4. El número personal se calcula sumando las cifras del
-- día/mes/año de nacimiento sucesivamente hasta que quede un solo
-- dígito. Por ejemplo, el número personal de los que han nacido el
-- 29/10/1994 se calcula por
--    29/10/1994 --> 2+9+1+0+1+9+4
--               =   26
--               --> 2+6
--               =   8
--
-- Definir la función personal tal que (personal x y z) es el número
-- personal de los que han nacido el día x del mes y del año z. Por
-- ejemplo,
--    personal 29 10 1994 == 8
-- ---------------------------------------------------------------------

personal x y z =
```

```
    reduce (sum (concat [digitos x, digitos y, digitos z]))

digitos n | n < 10    = [n]
          | otherwise = n `rem` 10 : digitos (n `div` 10)

reduce x | x < 10    = x
         | otherwise = reduce (sum (digitos x))

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por recursión, la función parMitad tal que
-- (parMitad xs) es la lista obtenida sustituyendo cada numero par de la
-- lista xs por su mitad. Por ejemplo,
--    parMitad [1,2,3,4,5,6] = [1,1,3,2,5,3]
-- ---------------------------------------------------------------------

parMitad [] = []
parMitad (x:xs) | even x    = x `div` 2 : parMitad xs
                | otherwise = x : parMitad xs

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la funcion parMitad1 que sea equivalente a
-- parMitad, pero no recursiva.
-- ---------------------------------------------------------------------

parMitad1 = map f
    where f x | even x = div x 2
              | otherwise = x

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck que las funciones parMitad y
-- parMitad1 son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_parMitad xs = parMitad xs == parMitad1 xs

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

5.2.3. Examen 3 (23 de Enero de 2014)

El examen es común con el del grupo 3 (ver página 380).

5.2.4. Examen 4 (24 de Marzo de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (24 de marzo de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List (nub, sort)

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Los polinomios se pueden representar mediante el
-- siguiente tipo algebraico
--    data Polinomio = Indep Int | Monomio Int Int Polinomio
--                     deriving (Eq, Show)
-- Por ejemplo, el polinomio 3x^2+2x-5 se representa por
--    ej1 = Monomio 3 2 (Monomio 2 1 (Indep (-5)))
-- y el polinomio 4x^3-2x por
--    ej2 = Monomio 4 3 (Monomio (-2) 1 (Indep 0))
-- Observa que si un monomio no aparece en el polinomio, en su
-- representación tampoco aparece; es decir, el coeficiente de un
-- monomio en la representación no debe ser cero.
--
-- Definir la función
--    sPol :: Polinomio -> Polinomio -> Polinomio
-- tal que (sPol p q) es la suma de p y q. Por ejemplo,
--    sPol ej1 ej2 == Monomio 4 3 (Monomio 3 2 (Indep (-5)))
--    sPol ej1 ej1 == Monomio 6 2 (Monomio 4 1 (Indep (-10)))
-- ---------------------------------------------------------------------

data Polinomio = Indep Int | Monomio Int Int Polinomio
                 deriving (Eq, Show)

ej1 = Monomio 3 2 (Monomio 2 1 (Indep (-5)))
```

```
ej2 = Monomio 4 3 (Monomio (-2) 1 (Indep 0))

sPol :: Polinomio -> Polinomio -> Polinomio
sPol (Indep 0) q = q
sPol p (Indep 0) = p
sPol (Indep n) (Indep m) = Indep (m+n)
sPol (Indep n) (Monomio c g p)= Monomio c g (sPol p (Indep n))
sPol (Monomio c g p) (Indep n) = Monomio c g (sPol p (Indep n))
sPol p1@(Monomio c1 g1 r1) p2@(Monomio c2 g2 r2)
    | g1 >  g2   = Monomio c1 g1 (sPol r1 p2)
    | g1 <  g2   = Monomio c2 g2 (sPol p1 r2)
    | c1+c2 /= 0 = Monomio (c1+c2) g1 (sPol r1 r2)
    | otherwise  = sPol r1 r2

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Los polinomios también se pueden representar mediante
-- la lista de sus coeficientes. Por ejemplo, el polinomio ej1 se
-- representa por [3,2,-5] y el polinomio ej2 vendrá por [4,0,-2,0].
--
-- Definir la función
--    cambia :: Polinomio -> [Int]
-- tal que (cambia p) es la lista de los coeficientes de p. Por ejemplo,
--    cambia ej1 == [3,2,-5]
--    cambia ej2 == [4,0,-2,0]
-- ---------------------------------------------------------------------

cambia :: Polinomio -> [Int]
cambia (Indep n) = [n]
cambia (Monomio c g (Indep n)) = (c:(replicate 0 (g-1)))++[n]
cambia (Monomio c1 g1 (Monomio c2 g2 p)) =
    (c1:(replicate  (g1-g2-1) 0)) ++ cambia (Monomio c2 g2 p)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios se pueden representar mediante el
-- siguiente tipo de datos
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
--
-- Definir la función
```

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

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

profundidades :: (Num t, Eq t) => Arbol t -> t -> [t]
profundidades (H y) x | x == y    = [1]
                      | otherwise = []
profundidades (N y i d) x
    | x == y    = 1:[n+1 | n <- profundidades i x ++  profundidades d x]
    | otherwise =   [n+1 | n <- profundidades i x ++  profundidades d x]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. La sucesión de Phill esta definida por
--    x_0 = 2
--    x_1 = 7
--    x_3 = 2*x_(n-1) - x_(n-2), si n > 1.
--
-- Definir, por recursión, la función
--    phill :: Integer -> Integer
-- tal que (phill n) es el n-ésimo término de la sucesión de Phill. Por
-- ejemplo,
--    phill 8  ==  42
-- ---------------------------------------------------------------------

phill :: Integer -> Integer
phill 0 = 2
phill 1 = 7
phill n = 2*(phill (n-1)) - phill (n-2)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por comprensión, la función
```

```
--    phills :: [Integer]
-- tal que phills es la sucesión de Phill. Por ejemplo,
--    take 8 phills  ==  [2,7,12,17,22,27,32,37]
-- ---------------------------------------------------------------------

phills :: [Integer]
phills = [phill n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, por recursión (evaluación perezosa) la
-- función
--    phills1 :: [Integer]
-- tal que phills1 es la sucesion de Phill. Por ejemplo,
--    take 8 phills1  ==  [2,7,12,17,22,27,32,37]
-- Nota: Dar dos definiciones, una sin usar zipWith o otra usándola.
-- ---------------------------------------------------------------------

-- Sin zipWith:
phills1 :: [Integer]
phills1  = aux 2 7
    where aux x y = x : aux y (2*y-x)

-- Con zipWith:
phills2 :: [Integer]
phills2 = 2:7:zipWith f phills2 (tail phills2)
    where f x y = 2*y-x

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir la función
--    unidades :: [Integer]
-- tal que unidades es la lista de los últimos dígitos de cada término de
-- la sucesión de Phills. Por ejemplo,
--    take 15 unidades == [2,7,2,7,2,7,2,7,2,7,2,7,2,7,2]
-- ---------------------------------------------------------------------

unidades :: [Integer]
unidades = map (`mod` 10) phills2

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir, usando unidades, la propiedad
```

```
--    propPhill :: Int -> Bool
-- tal que (propPhill n) se verifica si el término n-ésimo de la
-- sucesión de Phill termina en 2 o en 7 según n sea par o impar.
--
-- Comprobar la propiedad para los 500 primeros términos.
-- ---------------------------------------------------------------------

propPhill :: Int -> Bool
propPhill n | even n    = unidades !! n == 2
            | otherwise = unidades !! n == 7

-- La comprobación es
--    ghci> and [propPhill n |n <- [0 .. 499]]
--    True
```

5.2.5. Examen 5 (19 de Mayo de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (19 de mayo de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. De una lista se pueden extraer los elementos
-- consecutivos repetidos indicando el número de veces que se repite
-- cada uno. Por ejemplo, la lista [1,1,7,7,7,5,5,7,7,7,7] comienza con
-- dos 1, seguido de tres 7, dos 5 y cuatro 7; por tanto, la extracción
-- de consecutivos repetidos devolverá [(2,1),(3,7),(2,5),(4,7)]. En
-- [1,1,7,5,7,7,7,7], la extracción será [(2,1),(4,7)] ya que el primer
-- 7 y el 5 no se repiten.
--
-- Definir la funcion
--    extraer :: Eq a => [a] -> [(Int,a)]
-- tal que (extraer xs) es la lista que resulta de la extracción de
-- consecutivos repetidos en la lista xs. Por ejemplo,
```

```
--    extraer [1,1,7,7,7,5,5,7,7,7,7]  ==  [(2,1),(3,7),(2,5),(4,7)]
--    extraer "HHoollllla"              ==  [(2,'H'),(2,'o'),(4,'l')]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
-- ===============================

extraer :: Eq a => [a] -> [(Int,a)]
extraer xs = [(length (y:ys),y) | (y:ys) <- group xs, not (null ys)]

-- 2ª definición (por recursión)
-- =============================

extraer2 :: Eq a => [a] -> [(Int,a)]
extraer2 [] = []
extraer2 (x:xs) | n == 0    = extraer2 (drop n xs)
                | otherwise = (1+n,x) : extraer2 (drop n xs)
    where n = length (takeWhile (==x) xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Partiendo de un número a, se construye la sucesión
-- de listas [xs(1), xs(2) ,xs(3), ...] tal que
--    * xs(1) = [x(1,1), x(1,2), x(1,3), ....], donde
--      x(1,1) = a + el primer  primo mayor que a,
--      x(1,2) = a + el segundo primo mayor que a, ...
--    * xs(2) = [x(2,1), x(2,2), x(2,3), ....], donde
--      x(2,i) = x(1,i) + el primer primo mayor que x(1,1),
--    * xs(3) = [x(2,1), x(2,2), x(3,3), ....], donde
--      x(3,i) = x(2,i) + el primer primo mayor que x(2,1),
-- Por ejemplo, si empieza con a = 15, la sucesión es
--    [[15+17, 15+19, 15+23, ...],
--     [(15+17)+37, (15+19)+37,(15+23)+41, ...],
--     [((15+17)+37)+71, ...]]
--    = [[32,34,38,...],[69,71,79,...],[140,144,162,...],...]
--
-- Definir la función
--    sucesionN :: Integer -> Int -> [Integer]
-- tal que (sucesionN x n) es elemento n-ésimo de la sucesión que
-- empieza por a. Por ejemplo,
--    take 10 (sucesionN 15 2) ==  [69,71,79,91,93,105,115,117,129,139]
```

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

sucesionN :: Integer -> Int -> [Integer]
sucesionN x 1 = [x+y | y <- primos, y > x]
sucesionN x n = zipWith (+) (map menorPrimoMayor (sucesionN x (n-1)))
                            (sucesionN x (n-1))

menorPrimoMayor :: Integer -> Integer
menorPrimoMayor x = head [y | y <- primos, y > x]

primos :: [Integer]
primos = [x | x <- [1 .. ], factores x == [1,x]]

factores :: Integer -> [Integer]
factores x = [y | y <- [1 ..x], mod x y ==0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    sucesion :: Integer -> [[Integer]]
-- tal que (sucesion a) es la sucesión construida a partir de a. Por
-- ejemplo,
--    ghci> take 5 (map (take 4)(sucesion 15))
--    [[32,34,38,44],[69,71,79,91],[140,144,162,188],[289,293,325,379],
--     [582,600,656,762]]
-- ---------------------------------------------------------------------

sucesion :: Integer -> [[Integer]]
sucesion a = [sucesionN a n | n <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función
--    cuenta :: Integer -> Integer -> Int -> [Int]
-- tal que (cuenta a b n) es la lista del número de elementos de las
-- primeras n listas de la (sucesion a) que son menores que b. Por
-- ejemplo,
--    ghci> cuenta 15 80 5
--    [12,3,0,0,0]
-- ---------------------------------------------------------------------

cuenta :: Integer -> Integer -> Int -> [Int]
```

```
cuenta a b n =
    map (length . takeWhile (< b)) [sucesionN a m | m <- [1 .. n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    simetricos:: Eq a => [a] -> [a]
-- tal que (simetricos xs) es la lista de los elementos de xs que
-- coinciden con su simétricos. Por ejemplo,
--    simetricos [1,2,3,4,3,2,1]     == [1,2,3]
--    simetricos [1,2,5,4,3,4,3,2,1] == [1,2,4]
--    simetricos "amiima"            == "ami"
--    simetricos "ala"               == "a"
--    simetricos [1..20]             == []
-- ---------------------------------------------------------------------

simetricos:: Eq a => [a] -> [a]
simetricos xs =
    [x | (x,y) <- zip (take m xs) (take m (reverse xs)), x==y]
    where m = div (length xs) 2

-- ---------------------------------------------------------------------
-- Ejercicio 4. La matrices piramidales son las formadas por unos y ceros
-- de forma que los unos forman una pirámide. Por ejemplo,
--   |1|   |0 1 0|   |0 0 1 0 0|   |0 0 0 1 0 0 0|
--         |1 1 1|   |0 1 1 1 0|   |0 0 1 1 1 0 0|
--                   |1 1 1 1 1|   |0 1 1 1 1 1 0|
--                                 |1 1 1 1 1 1 1|
--
-- El tipo de las matrices se define por
--    type Matriz a = Array (Int,Int) a
-- Por ejemplo, las matrices anteriores se definen por
--    p1, p2, p3 :: Matriz Int
--    p1 = listArray ((1,1),(1,1)) [1]
--    p2 = listArray ((1,1),(2,3)) [0,1,0,
--                                  1,1,1]
--    p3 = listArray ((1,1),(3,5)) [0,0,1,0,0,
--                                  0,1,1,1,0,
--                                  1,1,1,1,1]
--
-- Definir la función
```

```
--    esPiramidal :: (Eq a, Num a) => Matriz a -> Bool
-- tal que (esPiramidal p) se verifica si la matriz p es piramidal. Por
-- ejemplo,
--    esPiramidal p3                                         ==  True
--    esPiramidal (listArray ((1,1),(2,3)) [0,1,0, 1,5,1])  ==  False
--    esPiramidal (listArray ((1,1),(2,3)) [0,1,1, 1,1,1])  ==  False
--    esPiramidal (listArray ((1,1),(2,3)) [0,1,0, 1,0,1])  ==  False
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

p1, p2, p3 :: Matriz Int
p1 = listArray ((1,1),(1,1)) [1]
p2 = listArray ((1,1),(2,3)) [0,1,0,
                              1,1,1]
p3 = listArray ((1,1),(3,5)) [0,0,1,0,0,
                              0,1,1,1,0,
                              1,1,1,1,1]

esPiramidal :: (Eq a, Num a) => Matriz a -> Bool
esPiramidal p =
    p == listArray ((1,1),(n,m)) (concat (filasPiramidal n))
    where (_,(n,m)) = bounds p

-- (filasPiramidal n) es la lista dela filas de la matriz piramidal de n
-- filas. Por ejemplo,
--    filasPiramidal 1  ==  [[1]]
--    filasPiramidal 2  ==  [[0,1,0],[1,1,1]]
--    filasPiramidal 3  ==  [[0,0,1,0,0],[0,1,1,1,0],[1,1,1,1,1]]
filasPiramidal 1 = [[1]]
filasPiramidal n = [0:xs++[0] | xs <- filasPiramidal (n-1)] ++
                   [replicate (2*n-1) 1]

-- 2ª definición
-- =============

esPiramidal2 :: (Eq a, Num a) => Matriz a -> Bool
esPiramidal2 p =
    p == piramidal n
    where (_,(n,_)) = bounds p
```

```
-- (piramidal n) es la matriz piramidal con n filas. Por ejemplo,
--    ghci> piramidal 3
--    array ((1,1),(3,5)) [((1,1),0),((1,2),0),((1,3),1),((1,4),0),((1,5),0),
--                          ((2,1),0),((2,2),1),((2,3),1),((2,4),1),((2,5),0),
--                          ((3,1),1),((3,2),1),((3,3),1),((3,4),1),((3,5),1)]
piramidal :: (Eq a, Num a) => Int -> Matriz a
piramidal n =
    array ((1,1),(n,2*n-1)) [((i,j),f i j) | i <- [1..n], j <- [1..2*n-1]]
    where f i j | j <= n-i  = 0
                | j <  n+i  = 1
                | otherwise = 0

-- ---------------------------------------------------------------------
-- Ejercicio 5. Los árboles se pueden representar mediante el siguiente
-- tipo de dato
--    data Arbol a = N a [Arbol a]
--                        deriving Show
-- Por ejemplo, los árboles
--      1               3                3
--     / \             /|\              / | \
--    2   3           / | \            /  |  \
--        |          5  4  7          5   4   7
--        4          |     /\         |   |  / \
--                   6    2  1        6   1 2   1
--                                       / \
--                                      2   3
--                                          |
--                                          4

-- se representan por
--    ej1, ej2,ej3 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 3 [N 5 [N 6 []],
--               N 4 [],
--               N 7 [N 2 [], N 1 []]]
--    ej3 = N 3 [N 5 [N 6 []],
--               N 4 [N 1 [N 2 [],N 3 [N 4 []]]],
--               N 7 [N 2 [], N 1 []]]
--
```

```
-- Definir la función
--     ramifica :: Arbol a -> Arbol a -> (a -> Bool) -> Arbol a
-- tal que (ramifica a1 a2 p) el árbol que resulta de añadir una copia
-- del árbol a2 a los nodos de a1 que cumplen un predicado p. Por
-- ejemplo,
--    ghci> ramifica (N 3 [N 5 [N 6 []],N 4 [],N 7 [N 2 [],N 1 []]]) (N 8 []) (>5)
--    N 3 [N 5 [N 6 [N 8 []]],N 4 [],N 7 [N 2 [],N 1 [],N 8 []]]
-- ---------------------------------------------------------------------

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

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

ramifica (N x xs) a2 p
         | p x       = N x ([ramifica a a2 p | a <- xs] ++ [a2])
         | otherwise = N x  [ramifica a a2 p | a <- xs]
```

5.2.6. Examen 6 (18 de Junio de 2014)

El examen es común con el del grupo 3 (ver página 393).

5.2.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 401).

5.2.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 408).

5.2.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 413).

5.3. Exámenes del grupo 3 (José A. Alonso y Luis Valencia)

5.3.1. 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]
```

5.3.2. 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]
```

5.3.3. Examen 3 (23 de Enero de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 3º examen de evaluación continua (23 de enero de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. El factorial de 7 es
--    7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040
-- por tanto, el último dígito no nulo del factorial de 7 es 4.
--
-- Definir la función
--    ultimoNoNuloFactorial :: Integer -> Integer
-- tal que (ultimoNoNuloFactorial n) es el último dígito no nulo del
-- factorial de n. Por ejemplo,
--    ultimoNoNuloFactorial 7  ==  4
-- ---------------------------------------------------------------------

ultimoNoNuloFactorial :: Integer -> Integer
ultimoNoNuloFactorial n = ultimoNoNulo (factorial n)

-- (ultimoNoNulo n) es el último dígito no nulo de n. Por ejemplo,
```

```
--    ultimoNoNulo 5040  ==  4
ultimoNoNulo :: Integer -> Integer
ultimoNoNulo n | m /= 0    = m
               | otherwise = ultimoNoNulo (n `div` 10)
               where m = n `rem` 10

-- 2ª definición (por comprensión)
ultimoNoNulo2 :: Integer -> Integer
ultimoNoNulo2 n = read [head (dropWhile (=='0') (reverse (show n)))]

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 7  ==  5040
factorial :: Integer -> Integer
factorial n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Una lista se puede comprimir indicando el número de
-- veces consecutivas que aparece cada elemento. Por ejemplo, la lista
-- comprimida de [1,1,7,7,7,5,5,7,7,7,7] es [(2,1),(3,7),(2,5),(4,7)],
-- indicando que comienza con dos 1, seguido de tres 7, dos 5 y cuatro
-- 7.
--
-- Definir, por comprensión, la función
--    expandidaC :: [(Int,a)] -> [a]
-- tal que (expandidaC ps) es la lista expandida correspondiente a ps
-- (es decir, es la lista xs tal que la comprimida de xs es ps). Por
-- ejemplo,
--    expandidaC [(2,1),(3,7),(2,5),(4,7)]  ==  [1,1,7,7,7,5,5,7,7,7,7]
-- ---------------------------------------------------------------------

expandidaC :: [(Int,a)] -> [a]
expandidaC ps = concat [replicate k x | (k,x) <- ps]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    expandidaR :: [(Int,a)] -> [a]
-- tal que (expandidaR ps) es la lista expandida correspondiente a ps
-- (es decir, es la lista xs tal que la comprimida de xs es ps). Por
-- ejemplo,
--    expandidaR [(2,1),(3,7),(2,5),(4,7)]  ==  [1,1,7,7,7,5,5,7,7,7,7]
```

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

expandidaR :: [(Int,a)] -> [a]
expandidaR []         = []
expandidaR ((n,x):ps) = replicate n x ++ expandidaR ps

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Un número n es de Angelini si n y 2n tienen algún
-- dígito común. Por ejemplo, 2014 es un número de Angelini ya que 2014
-- y su doble (4028) comparten los dígitos 4 y 0.
--
-- Definir la función
--    angelini :: Integer -> Bool
-- tal que (angelini n) se verifica si n es un número de Angelini. Por
-- ejemplo,
--    angelini 2014  ==  True
--    angelini 2067  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (con any)
angelini :: Integer -> Bool
angelini n = any (`elem` (show (2*n))) (show n)

-- 2ª definición (por comprensión)
angelini2 :: Integer -> Bool
angelini2 n = not (null [x | x <- show n, x `elem` show (2*n)])

-- 3ª definición (por recursión)
angelini3 :: Integer -> Bool
angelini3 n = aux (show n) (show (2*n))
    where aux [] _      = False
          aux (x:xs) ys = x `elem` ys || aux xs ys

-- 4ª definición (por plegado)
angelini4 :: Integer -> Bool
angelini4 n = aux (show n)
    where aux   = foldr f False
          f x y = x `elem` show (2*n) || y

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

```
-- Ejercicio 3.2. ¿Cuál es el primer año que no será de Angelini?
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [2014..], not (angelini n)]
--    2057

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El número 37 es primo y se puede escribir como suma de
-- primos menores distintos (en efecto, los números 3, 11 y 23 son
-- primos y su suma es 37.
--
-- Definir la función
--    primoSumaDePrimos :: Integer -> Bool
-- tal que (primoSumaDePrimos n) se verifica si n es primo y se puede
-- escribir como suma de primos menores que n. Por ejemplo,
--    primoSumaDePrimos 37  ==  True
--    primoSumaDePrimos 39  ==  False
--    primoSumaDePrimos 11  ==  False
-- ---------------------------------------------------------------------

primoSumaDePrimos :: Integer -> Bool
primoSumaDePrimos n = primo n && esSumaDePrimos n

-- (esSumaDePrimos n) se verifica si n es una suma de primos menores que
-- n. Por ejemplo,
--    esSumaDePrimos 37  ==  True
--    esSumaDePrimos 11  ==  False
esSumaDePrimos :: Integer -> Bool
esSumaDePrimos n = esSuma n [x | x <- 2:[3,5..n-1], primo x]

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 37  ==  True
--    primo 38  ==  False
primo :: Integer -> Bool
primo n = [x | x <- [1..n], n `rem` x == 0] == [1,n]

-- (esSuma n xs) s verifica si n es suma de elementos de xs. Por ejemplo,
--    esSuma 20 [4,2,7,9]  ==  True
--    esSuma 21 [4,2,7,9]  ==  False
```

```
esSuma :: Integer -> [Integer] -> Bool
esSuma 0 _                  = True
esSuma n []                 = False
esSuma n (x:xs) | n == x    = True
                | n > x     = esSuma n xs || esSuma (n-x) xs
                | otherwise = esSuma n xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. ¿Cuál será el próximo año primo suma de primos? ¿y el
-- anterior?
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [p | p <- [2014..], primoSumaDePrimos p]
--    2017
--    ghci> head [p | p <- [2014,2013..], primoSumaDePrimos p]
--    2011
```

5.3.4. Examen 4 (21 de Marzo de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (21 de marzo de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Definir la función
--    interpretaciones :: [a] -> [[(a,Int)]]
-- tal que (interpretaciones xs) es la lista de las interpretaciones
-- sobre la lista de las variables proposicionales xs sobre los valores
-- de verdad 0 y 1. Por ejemplo,
--    ghci> interpretaciones "A"
--    [[('A',0)],[('A',1)]]
--    ghci> interpretaciones "AB"
--    [[('A',0),('B',0)],[('A',0),('B',1)],
--     [('A',1),('B',0)],[('A',1),('B',1)]]
--    ghci> interpretaciones "ABC"
--    [[('A',0),('B',0),('C',0)],[('A',0),('B',0),('C',1)],
--     [('A',0),('B',1),('C',0)],[('A',0),('B',1),('C',1)],
--     [('A',1),('B',0),('C',0)],[('A',1),('B',0),('C',1)],
--     [('A',1),('B',1),('C',0)],[('A',1),('B',1),('C',1)]]
```

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

interpretaciones :: [a] -> [[(a,Int)]]
interpretaciones [] = [[]]
interpretaciones (x:xs) =
    [(x,0):i | i <-is] ++ [(x,1):i | i <- is]
    where is = interpretaciones xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Los números de Hamming forman una sucesión
-- estrictamente creciente de números que cumplen las siguientes
-- condiciones:
--    * El número 1 está en la sucesión.
--    * Si x está en la sucesión, entonces 2x, 3x y 5x también están.
--    * Ningún otro número está en la sucesión.
-- Los primeros términos de la sucesión de Hamming son
--    1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, ...
--
-- Definir la función
--    siguienteHamming :: Int -> Int
-- tal que (siguienteHamming x) es el menor término de la sucesión de
-- Hamming mayor que x. Por ejemplo,
--    siguienteHamming 10  ==  12
--    siguienteHamming 12  ==  15
--    siguienteHamming 15  ==  16
-- ---------------------------------------------------------------------

siguienteHamming :: Int -> Int
siguienteHamming x = head (dropWhile (<=x) hamming)

-- hamming es la sucesión de Hamming. Por ejemplo,
--    take 12 hamming == [1,2,3,4,5,6,8,9,10,12,15,16]
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]

-- (mezcla3 xs ys zs) es la lista obtenida mezclando las listas
-- ordenadas xs, ys y zs y eliminando los elementos duplicados. Por
-- ejemplo,
```

```
--    ghci> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
--    [2,3,4,5,6,8,9,10,12]
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)

-- (mezcla2 xs ys zs) es la lista obtenida mezclando las listas
-- ordenadas xs e ys y eliminando los elementos duplicados. Por ejemplo,
--    ghci> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
--    [2,3,4,6,8,9,10,12]
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []       ys                   = ys
mezcla2 xs       []                   = xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Las operaciones de suma, resta y
-- multiplicación se pueden representar mediante el siguiente tipo de
-- datos
--    data Op = S | R | M
-- La expresiones aritméticas con dichas operaciones se pueden
-- representar mediante el siguiente tipo de dato algebraico
--    data Expr = N Int | A Op Expr Expr
-- Por ejemplo, la expresión
--    (7-3)+(2*5)
-- se representa por
--    A S (A R (N 7) (N 3)) (A M (N 2) (N 5))
--
-- Definir la función
--    valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--    valor (A S (A R (N 7) (N 3)) (A M (N 2) (N 5)))  ==  14
--    valor (A M (A R (N 7) (N 3)) (A S (N 2) (N 5)))  ==  28
-- ---------------------------------------------------------------------

data Op = S | R | M

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

```
valor :: Expr -> Int
valor (N x) = x
valor (A o e1 e2) = aplica o (valor e1) (valor e2)

aplica :: Op -> Int -> Int -> Int
aplica S x y = x+y
aplica R x y = x-y
aplica M x y = x*y

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] Los polinomios con coeficientes naturales
-- se pueden representar mediante el siguiente tipo algebraico
--    data Polinomio = O | C Int Int Polinomio
-- Por ejemplo, el polinomio 2x^5 + 4x^3 + 2x se representa por
--    C 2 5 (C 4 3 (C 2 1 O))
-- También se pueden representar mediante listas no crecientes de
-- naturales. Por ejemplo, el polinomio 2x^5 + 4x^3 + 2x se representa
-- por
--    [5,5,3,3,3,3,1,1]
-- en la lista anterior, el número de veces que aparece cada número n es
-- igual al coeficiente de x^n en el polinomio.
--
-- Definir la función
--    transformaPol :: Polinomio -> [Int]
-- tal que (transformaPol p) es el polinomio obtenido transformado el
-- polinomio p de la primera representación a la segunda. Por ejemplo,
--    transformaPol (C 2 5 (C 4 3 (C 2 1 O)))  ==  [5,5,3,3,3,3,1,1]
--    transformaPol (C 2 100 (C 3 1 O))        ==  [100,100,1,1,1]
-- ---------------------------------------------------------------------

data Polinomio = O | C Int Int Polinomio

transformaPol :: Polinomio -> [Int]
transformaPol O         = []
transformaPol (C a n p) = replicate a n ++ transformaPol p
```

5.3.5. Examen 5 (16 de Mayo de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (16 de mayo de 2014)
```

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

import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Un mínimo local de una lista es un elemento
-- de la lista que es menor que su predecesor y que su sucesor en la
-- lista. Por ejemplo, 1 es un mínimo local de [3,2,1,3,7,7,1,0,2] ya
-- que es menor  que 2 (su predecesor) y que 3 (su sucesor).
--
-- Definir la función
--    minimosLocales :: Ord a => [a] -> [a]
-- tal que (minimosLocales xs) es la lista de los mínimos locales de la
-- lista xs. Por ejemplo,
--    minimosLocales [3,2,1,3,7,7,9,6,8]  ==  [1,6]
--    minimosLocales [1..100]             ==  []
--    minimosLocales "mqexvzat"           ==  "eva"
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
minimosLocales1 :: Ord a => [a] -> [a]
minimosLocales1 (x:y:z:xs) | y < x && y < z = y : minimosLocales1 (z:xs)
                           | otherwise      = minimosLocales1 (y:z:xs)
minimosLocales1 _                           = []

-- 2ª definición (por comprensión):
minimosLocales2 :: Ord a => [a] -> [a]
minimosLocales2 xs =
    [y | (x,y,z) <- zip3 xs (tail xs) (drop 2 xs), y < x, y < z]

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    sumaExtremos :: Num a => [a] -> [a]
-- tal que (sumaExtremos xs) es la lista sumando el primer elemento de
-- xs con el último, el segundo con el penúltimo y así
-- sucesivamente. Por ejemplo,
--    sumaExtremos [6,5,3,1]              ==  [7,8]
--    sumaExtremos [6,5,3]                ==  [9,10]
--    sumaExtremos [3,2,3,2]              ==  [5,5]
--    sumaExtremos [6,5,3,1,2,0,4,7,8,9]  ==  [15,13,10,5,2]
```

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

-- 1ª definición (por recursión):
sumaExtremos1 :: Num a => [a] -> [a]
sumaExtremos1 []     = []
sumaExtremos1 [x]    = [x+x]
sumaExtremos1 (x:xs) = (x + last xs) : sumaExtremos1 (init xs)


-- 2ª definición (por recursión):
sumaExtremos2 :: Num a => [a] -> [a]
sumaExtremos2 xs = aux (take n xs) (take n (reverse xs))
    where aux [] []         = []
          aux (x:xs) (y:ys) = x+y : aux xs ys
          m = length xs
          n | even m    = m `div` 2
            | otherwise = 1 + (m `div` 2)

-- 3ª definición (con zip):
sumaExtremos3 :: Num a => [a] -> [a]
sumaExtremos3 xs = take n [x+y | (x,y) <- zip xs (reverse xs)]
    where m = length xs
          n | even m    = m `div` 2
            | otherwise = 1 + (m `div` 2)

-- 4ª definición (con zipWith):
sumaExtremos4 :: Num a => [a] -> [a]
sumaExtremos4 xs = take n (zipWith (+) xs (reverse xs))
    where m = length xs
          n | even m    = m `div` 2
            | otherwise = 1 + (m `div` 2)

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    listaRectangular :: Int -> Int -> a -> [a] -> [[a]]
-- tal que (listaRectangular m n x xs) es una lista de m listas de
-- longitud n formadas con los elementos de xs completada con x, si no
-- xs no tiene suficientes elementos. Por ejemplo,
--    listaRectangular 2 4 7 [0,3,5,2,4]  ==  [[0,3,5,2],[4,7,7,7]]
--    listaRectangular 4 2 7 [0,3,5,2,4]  ==  [[0,3],[5,2],[4,7],[7,7]]
```

```
--    listaRectangular 2 3 7 [0..]        ==  [[0,1,2],[3,4,5]]
--    listaRectangular 3 2 7 [0..]        ==  [[0,1],[2,3],[4,5]]
--    listaRectangular 3 2 'p' "eva"      ==  ["ev","ap","pp"]
--    listaRectangular 3 2 'p' ['e'..]    ==  ["ef","gh","ij"]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
listaRectangular1 :: Int -> Int -> a -> [a] -> [[a]]
listaRectangular1 m n x xs =
    take m (grupos n (xs ++ repeat x))

-- (grupos n xs) es la lista obtenida agrupando los elementos de xs en
-- grupos de n elementos, salvo el último que puede tener menos. Por
-- ejemplo,
--    grupos 2 [4,2,5,7,6]     ==  [[4,2],[5,7],[6]]
--    take 3 (grupos 3 [1..])  ==  [[1,2,3],[4,5,6],[7,8,9]]
grupos :: Int -> [a] -> [[a]]
grupos _ [] = []
grupos n xs = take n xs : grupos n (drop n xs)

-- 2ª definición (por comprensión)
listaRectangular2 :: Int -> Int -> a -> [a] -> [[a]]
listaRectangular2 m n x xs =
    take m [take n ys | m <- [0,n..n^2],
                        ys <- [drop m xs ++ (replicate m x)]]

-- 3ª definición (por iteración):
listaRectangular3 :: Int -> Int -> a -> [a] -> [[a]]
listaRectangular3 m n x xs =
    take n [take n ys | ys <- iterate (drop n) (xs ++ repeat x)]

-- 4ª definición (sin el 4º argumento):
listaRectangular4 :: Int -> Int -> a -> [a] -> [[a]]
listaRectangular4 m n x =
    take m . map (take n) . iterate (drop n) . (++ repeat x)

-- ---------------------------------------------------------------------
-- Ejercicio 4 [2 puntos] Las expresiones aritméticas se pueden definir
-- usando el siguiente tipo de datos
--    data Expr = N Int
```

```
--              | S Expr Expr
--              | P Expr Expr
--              deriving (Eq, Show)
-- Por ejemplo, la expresión
--    3*5 + 6*7
-- se puede definir por
--    S (P (N 3) (N 5)) (P (N 6) (N 7))
--
-- Definir la función
--    aplica :: (Int -> Int) -> Expr -> Expr
-- tal que (aplica f e) es la expresión obtenida aplicando la función f
-- a cada uno de los números de la expresión e. Por ejemplo,
--    ghci> aplica (+2) (S (P (N 3) (N 5)) (P (N 6) (N 7)))
--    S (P (N 5) (N 7)) (P (N 8) (N 9))
--    ghci> aplica (*2) (S (P (N 3) (N 5)) (P (N 6) (N 7)))
--    S (P (N 6) (N 10)) (P (N 12) (N 14))
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] Las matrices enteras se pueden representar
-- mediante tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
-- Por ejemplo, la matriz
--    |4 1 3|
--    |1 2 8|
--    |6 5 7|
-- se puede definir por
--    listArray ((1,1),(3,3)) [4,1,3, 1,2,8, 6,5,7]
--
-- Definir la función
```

```
--    sumaColumnas :: Matriz -> Matriz
-- tal que (sumaColumnas p) es la matriz obtenida sumando a cada columna
-- la anterior salvo a la primera que le suma la última columna. Por
-- ejemplo,
--    ghci> sumaColumnas (listArray ((1,1),(3,3)) [4,1,3, 1,2,8, 6,5,7])
--    array ((1,1),(3,3)) [((1,1),7), ((1,2),5), ((1,3),4),
--                         ((2,1),9), ((2,2),3), ((2,3),10),
--                         ((3,1),13),((3,2),11),((3,3),12)]
-- es decir, el resultado es la matriz
--    | 7  5  4|
--    | 9  3 10|
--    |13 11 12|
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

sumaColumnas :: Matriz -> Matriz
sumaColumnas p =
    array ((1,1),(m,n))
          [((i,j), f i j) | i <- [1..m], j <- [1..n]]
    where (_,(m,n)) = bounds p
          f i 1 = p!(i,1) + p!(i,m)
          f i j = p!(i,j) + p!(i,j-1)
```

5.3.6. Examen 6 (18 de Junio de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (18 de junio de 2014)
-- ---------------------------------------------------------------------

-- Librería auxiliar
-- =================
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1 [2 puntos]. Definir la función
--    divisiblesPorPrimero :: [Int] -> Bool
-- tal que (divisibles xs) se verifica si todos los elementos positivos
-- de xs son divisibles por el primero. Por ejemplo,
--    divisiblesPorPrimero [2,6,-3,0,18,-17,10]  ==  True
--    divisiblesPorPrimero [-13]                 ==  True
```

```
--    divisiblesPorPrimero [-3,6,1,-3,9,18]      ==  False
--    divisiblesPorPrimero [5,-2,-6,3]           ==  False
--    divisiblesPorPrimero []                    ==  False
--    divisiblesPorPrimero [0,2,4]               ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
divisiblesPorPrimero1 :: [Int] -> Bool
divisiblesPorPrimero1 []     = False
divisiblesPorPrimero1 (0:_)  = False
divisiblesPorPrimero1 (x:xs) = and [y `rem` x == 0 | y <- xs, y > 0]

-- 2ª definición (por recursión)
divisiblesPorPrimero2 :: [Int] -> Bool
divisiblesPorPrimero2 []     = False
divisiblesPorPrimero2 (0:_)  = False
divisiblesPorPrimero2 (x:xs) = aux xs
    where aux [] = True
          aux (y:ys) | y > 0     = y `rem` x == 0 && aux ys
                     | otherwise = aux ys

-- ---------------------------------------------------------------------
-- Ejercicio 2 [2 puntos]. Definir la constante
--    primosConsecutivosConMediaCapicua :: [(Int,Int,Int)]
-- tal que primosConsecutivosConMediaCapicua es la lista de las ternas
-- (x,y,z) tales que x e y son primos consecutivos tales que su media,
-- z, es capicúa. Por ejemplo,
--    ghci> take 5 primosConsecutivosConMediaCapicua
--    [(3,5,4),(5,7,6),(7,11,9),(97,101,99),(109,113,111)]
-- Calcular cuántos hay anteriores a 2014.
-- ---------------------------------------------------------------------

primosConsecutivosConMediaCapicua :: [(Int,Int,Int)]
primosConsecutivosConMediaCapicua =
    [(x,y,z) | (x,y) <- zip primos (tail primos),
               let z = (x + y) `div` 2,
               capicua z]

-- (primo x) se verifica si x es primo. Por ejemplo,
--    primo 7  ==  True
```

```haskell
--    primo 8  ==  False
primo :: Int -> Bool
primo x = [y | y <- [1..x], x `rem` y == 0] == [1,x]

-- primos es la lista de los números primos mayores que 2. Por ejemplo,
--    take 10 primos  ==  [3,5,7,11,13,17,19,23,29]
primos :: [Int]
primos = [x | x <- [3,5..], primo x]

-- (capicua x) se verifica si x es capicúa. Por ejemplo,
capicua :: Int -> Bool
capicua x = ys == reverse ys
    where ys = show x

-- El cálculo es
--    ghci> length (takeWhile (\(x,y,z) -> y < 2014) primosConsecutivosConMediaCapicua
--    20

-- ---------------------------------------------------------------------
-- Ejercicio 3 [2 puntos]. Un elemento x de un conjunto xs es minimal
-- respecto de una relación r si no existe ningún elemento y en xs tal
-- que (r y x). Por ejemplo,
--
-- Definir la función
--    minimales :: Eq a => (a -> a -> Bool) -> [a] -> [a]
-- tal que (minimales xss) es la lista de los elementos minimales de
-- xs. Por ejemplo,
--    ghci> minimales (\x y -> y `rem` x == 0) [2,3,6,12,18]
--    [2,3]
--    ghci> minimales (\x y -> x `rem` y == 0) [2,3,6,12,18]
--    [12,18]
--    ghci> minimales (\x y -> maximum x < maximum y) ["ac","cd","aacb"]
--    ["ac","aacb"]
--    ghci> minimales (\xs ys -> all (`elem` ys) xs) ["ab","c","abc","d","dc"]
--    ["ab","c","d"]
-- ---------------------------------------------------------------------

minimales :: Eq a => (a -> a -> Bool) -> [a] -> [a]
minimales r xs = [x | x <- xs, esMinimal r xs x]
```

```
-- (esMinimal r xs x) s verifica si xs no tiene ningún elemento menor
-- que x respecto de la relación r.
esMinimal :: Eq a => (a -> a -> Bool) -> [a] -> a -> Bool
esMinimal r xs x = null [y | y <- xs, y /= x, r y x]

-- ---------------------------------------------------------------------
-- Ejercicio 4 [2 puntos]. Una matriz es monomial si en cada una de sus
-- filas y columnas todos los elementos son nulos excepto 1. Por
-- ejemplo, de las matrices
--    |0  0 3 0|     |0  0 3 0|
--    |0 -2 0 0|     |0 -2 0 0|
--    |1  0 0 0|     |1  0 0 0|
--    |0  0 0 1|     |0  1 0 1|
-- la primera es monomial y la segunda no lo es.
--
-- Las matrices puede representarse mediante tablas cuyos
-- índices son pares de números naturales:
--    type Matriz = Array (Int,Int) Int
-- Por ejemplo, las matrices anteriores se pueden definir por
--    ej1, ej2 :: Matriz
--    ej1 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
--                                   0, -2, 0, 0,
--                                   1,  0, 0, 0,
--                                   0,  0, 0, 1]
--    ej2 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
--                                   0, -2, 0, 0,
--                                   1,  0, 0, 0,
--                                   0,  1, 0, 1]
-- Definir la función
--    esMonomial :: Matriz -> Bool
-- tal que (esMonomial p) se verifica si la matriz p es monomial. Por
-- ejemplo,
--    esMonomial ej1  ==  True
--    esMonomial ej2  ==  False
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

ej1, ej2 :: Matriz
ej1 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
```

```
                               0, -2, 0, 0,
                               1,  0, 0, 0,
                               0,  0, 0, 1]
ej2 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
                               0, -2, 0, 0,
                               1,  0, 0, 0,
                               0,  1, 0, 1]

esMonomial :: Matriz -> Bool
esMonomial p = all esListaMonomial (filas p ++ columnas p)

-- (filas p) es la lista de las filas de la matriz p. Por ejemplo,
--    filas ej1  ==  [[0,0,3,0],[0,-2,0,0],[1,0,0,0],[0,0,0,1]]
filas :: Matriz -> [[Int]]
filas p = [[p!(i,j) | j <- [1..n]] | i <- [1..m]]
    where (_,(m,n)) = bounds p

-- (columnas p) es la lista de las columnas de la matriz p. Por ejemplo,
--    columnas ej1  ==  [[0,0,1,0],[0,-2,0,0],[3,0,0,0],[0,0,0,1]]
columnas :: Matriz -> [[Int]]
columnas p = [[p!(i,j) | i <- [1..m]] | j <- [1..n]]
    where (_,(m,n)) = bounds p

-- (esListaMonomial xs) se verifica si todos los elementos de xs excepto
-- uno son nulos. Por ejemplo,
--    esListaMonomial [0,3,0,0]  ==  True
--    esListaMonomial [0,3,0,2]  ==  False
--    esListaMonomial [0,0,0,0]  ==  False
esListaMonomial :: [Int] -> Bool
esListaMonomial xs = length (filter (/=0) xs) == 1

-- ---------------------------------------------------------------------
-- Ejercicio 5 [2 puntos]. Se consideran las expresiones vectoriales
-- formadas por un vector, la suma de dos expresiones vectoriales o el
-- producto de un entero por una expresión vectorial. El siguiente tipo
-- de dato define las expresiones vectoriales
--    data ExpV = Vec Int Int
--              | Sum ExpV ExpV
--              | Mul Int ExpV
--              deriving Show
```

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

data ExpV = Vec Int Int
          | Sum ExpV ExpV
          | Mul Int ExpV
          deriving Show

-- 1ª solución
-- ===========
valor :: ExpV -> (Int,Int)
valor (Vec x y)   = (x,y)
valor (Sum e1 e2) = (x1+x2,y1+y2) where (x1,y1) = valor e1
                                        (x2,y2) = valor e2
valor (Mul n e)   = (n*x,n*y) where (x,y) = valor e

-- 2ª solución
-- ===========
valor2 :: ExpV -> (Int,Int)
valor2 (Vec a b)   = (a, b)
valor2 (Sum e1 e2) = suma (valor2 e1) (valor2 e2)
valor2 (Mul n e1)  = multiplica n (valor2 e1)

suma :: (Int,Int) -> (Int,Int) -> (Int,Int)
suma (a,b) (c,d) = (a+c,b+d)

multiplica :: Int -> (Int, Int) -> (Int, Int)
multiplica n (a,b) = (n*a,n*b)
```

5.3.7. Examen 7 (4 de Julio de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 1ª convocatoria (4 de julio de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] Una lista de longitud n > 0 es completa si el
-- valor absoluto de las diferencias de sus elementos consecutivos toma
-- todos los valores entre 1 y n-1 (sólo una vez). Por ejemplo,
-- [4,1,2,4] es completa porque los valores absolutos de las diferencias
-- de sus elementos consecutivos es [3,1,2].
--
-- Definir la función
--    esCompleta :: [Int] -> Bool
-- tal que (esCompleta xs) se verifica si xs es completa. Por ejemplo,
--    esCompleta [4,1,2,4]  ==  True
--    esCompleta [6]        ==  True
--    esCompleta [6,7]      ==  True
--    esCompleta [6,8]      ==  False
--    esCompleta [6,7,9]    ==  True
--    esCompleta [8,7,5]    ==  True
-- ---------------------------------------------------------------------

esCompleta :: [Int] -> Bool
esCompleta xs =
    sort [abs (x-y) | (x,y) <- zip xs (tail xs)] == [1..length xs - 1]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    unionG :: Ord a => [[a]] -> [a]
-- tal que (unionG xss) es la unión de xss cuyos elementos son listas
-- estrictamente crecientes (posiblemente infinitas). Por ejemplo,
--    ghci> take 10 (unionG [[2,4..],[3,6..],[5,10..]])
```

```
--    [2,3,4,5,6,8,9,10,12,14]
--    ghci> take 10 (unionG [[2,5..],[3,8..],[4,10..],[16..]])
--    [2,3,4,5,8,10,11,13,14,16]
--    ghci> unionG [[3,8],[4,10],[2,5],[16]]
--    [2,3,4,5,8,10,16]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión)
-- =============================
unionG1 :: Ord a => [[a]] -> [a]
unionG1 []          = []
unionG1 [xs]        = xs
unionG1 (xs:ys:zss) = xs `unionB` unionG1 (ys:zss)

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

-- 2ª definición (por plegado)
-- ===========================
unionG2 :: Ord a => [[a]] -> [a]
unionG2 = foldr unionB []

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    noEsSuma :: [Integer] -> Integer
-- tal que (noEsSuma xs) es el menor entero positivo que no se puede
-- expresar como suma de elementos de la lista creciente de números
-- positivos xs (ningún elemento se puede usar más de una vez). Por
-- ejemplo,
--    noEsSuma [1,2,3,8]      ==  7
--    noEsSuma (1:[2,4..10])  ==  32
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========
noEsSuma1 :: [Integer] -> Integer
```

```
noEsSuma1 xs = head [n | n <- [1..], n `notElem` sumas1 xs]

-- (sumas1 xs) es la lista de las sumas con los elementos de xs, donde
-- cada elemento se puede sumar como máximo una vez. Por ejemplo,
--    sumas1 [3,8]      ==  [11,3,8,0]
--    sumas1 [3,8,17]   ==  [28,11,20,3,25,8,17,0]
--    sumas1 [1,2,3,8]  ==  [14,6,11,3,12,4,9,1,13,5,10,2,11,3,8,0]
sumas1 :: [Integer] -> [Integer]
sumas1 [] = [0]
sumas1 (x:xs) = [x+y | y <- ys] ++ ys
    where ys = sumas1 xs

-- 2ª solución
-- ===========
noEsSuma2 :: [Integer] -> Integer
noEsSuma2 xs = head [n | n <- [1..], not (esSuma n xs)]

esSuma :: Integer -> [Integer] -> Bool
esSuma n [] = n == 0
esSuma n (x:xs) | n < x     = False
                | n == x    = True
                | otherwise = esSuma (n-x) xs || esSuma n xs

-- 3ª solución
-- ===========
noEsSuma3 :: [Integer] -> Integer
noEsSuma3 xs = aux xs 0
    where aux [] n     = n+1
          aux (x:xs) n | x <= n+1  = aux xs (n+x)
                       | otherwise = n+1

-- Comparaciones de eficiencia
-- ===========================

-- Las comparaciones son
--    ghci> noEsSuma1 ([1..10]++[12..20])
--    200
--    (8.28 secs, 946961604 bytes)
--    ghci> noEsSuma2 ([1..10]++[12..20])
--    200
```

```
--    (2.52 secs, 204156056 bytes)
--    ghci> noEsSuma3 ([1..10]++[12..20])
--    200
--    (0.01 secs, 520348 bytes)
--
--    ghci> noEsSuma2 (1:[2,4..30])
--    242
--    (4.97 secs, 399205788 bytes)
--    ghci> noEsSuma3 (1:[2,4..30])
--    242
--    (0.01 secs, 514340 bytes)
--
--    ghci> noEsSuma3 (1:[2,4..2014])
--    1015058
--    (0.01 secs, 1063600 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Los divisores medios de un número son los que
-- ocupan la posición media entre los divisores de n, ordenados de menor
-- a mayor. Por ejemplo, los divisores de 60 son
-- [1,2,3,4,5,6,10,12,15,20,30,60] y sus divisores medios son 6 y 10.
--
-- El árbol de factorización de un número compuesto n se construye de la
-- siguiente manera:
--    * la raíz es el número n,
--    * la rama izquierda es el árbol de factorización de su divisor
--      medio menor y
--    * la rama derecha es el árbol de factorización de su divisor
--      medio mayor
-- Si el número es primo, su árbol de factorización sólo tiene una hoja
-- con dicho número. Por ejemplo, el árbol de factorización de 60 es
--        60
--       /  \
--      6    10
--     / \   / \
--    2   3 2   5
--
-- Los árboles se representarán por
--    data Arbol = H Int
--               | N Int Arbol Arbol
```

```
--               deriving Show
--
-- Definir la función
--    arbolFactorizacion :: Int -> Arbol
-- tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
-- ejemplo,
--    ghci> arbolFactorizacion 60
--    N 60 (N 6 (H 2) (H 3)) (N 10 (H 2) (H 5))
--    ghci> arbolFactorizacion 45
--    N 45 (H 5) (N 9 (H 3) (H 3))
--    ghci> arbolFactorizacion 7
--    H 7
--    ghci> arbolFactorizacion 14
--    N 14 (H 2) (H 7)
--    ghci> arbolFactorizacion 28
--    N 28 (N 4 (H 2) (H 2)) (H 7)
--    ghci> arbolFactorizacion 84
--    N 84 (H 7) (N 12 (H 3) (N 4 (H 2) (H 2)))
-- ---------------------------------------------------------------------

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

-- 1ª definición
-- =============
arbolFactorizacion :: Int -> Arbol
arbolFactorizacion n
    | esPrimo n = H n
    | otherwise = N n (arbolFactorizacion x) (arbolFactorizacion y)
    where (x,y) = divisoresMedio n

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

-- (divisoresMedio n) es el par formado por los divisores medios de
-- n. Por ejemplo,
```

```
--    divisoresMedio 30  ==  (5,6)
--    divisoresMedio  7  ==  (1,7)
divisoresMedio :: Int -> (Int,Int)
divisoresMedio n = (n `div` x,x)
    where xs = divisores n
          x  = xs !! (length xs `div` 2)

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

-- 2ª definición
-- =============
arbolFactorizacion2 :: Int -> Arbol
arbolFactorizacion2 n
    | x == 1    = H n
    | otherwise = N n (arbolFactorizacion x) (arbolFactorizacion y)
    where (x,y) = divisoresMedio n

-- (divisoresMedio2 n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio2 30  ==  (5,6)
--    divisoresMedio2  7  ==  (1,7)
divisoresMedio2 :: Int -> (Int,Int)
divisoresMedio2 n = (n `div` x,x)
    where m  = ceiling (sqrt (fromIntegral n))
          x = head [y | y <- [m..n], n `rem` y == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] El triángulo de Pascal es un triángulo de
-- números
--          1
--         1 1
--        1 2 1
--      1  3 3  1
--     1 4  6  4 1
--    1 5 10 10 5 1
--   ...............
-- construido de la siguiente forma
```

```
-- * la primera fila está formada por el número 1;
-- * las filas siguientes se construyen sumando los números adyacentes
--   de la fila superior y añadiendo un 1 al principio y al final de la
--   fila.
--
-- La matriz de Pascal es la matriz cuyas filas son los elementos de la
-- correspondiente fila del triángulo de Pascal completadas con
-- ceros. Por ejemplo, la matriz de Pascal de orden 6 es
--    |1 0  0  0 0 0|
--    |1 1  0  0 0 0|
--    |1 2  1  0 0 0|
--    |1 3  3  1 0 0|
--    |1 4  6  4 1 0|
--    |1 5 10 10 5 1|
--
-- Las matrices se definen mediante el tipo
--    type Matriz = Array (Int,Int) Int
--
-- Definir la función
--    matrizPascal :: Int -> Matriz
-- tal que (matrizPascal n) es la matriz de Pascal de orden n. Por
-- ejemplo,
--    ghci> matrizPascal 5
--    array ((1,1),(5,5))
--          [((1,1),1),((1,2),0),((1,3),0),((1,4),0),((1,5),0),
--           ((2,1),1),((2,2),1),((2,3),0),((2,4),0),((2,5),0),
--           ((3,1),1),((3,2),2),((3,3),1),((3,4),0),((3,5),0),
--           ((4,1),1),((4,2),3),((4,3),3),((4,4),1),((4,5),0),
--           ((5,1),1),((5,2),4),((5,3),6),((5,4),4),((5,5),1)]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

-- 1ª solución
-- ===========

matrizPascal1 :: Int -> Matriz
matrizPascal1 1 = array ((1,1),(1,1)) [((1,1),1)]
matrizPascal1 n =
    array ((1,1),(n,n)) [((i,j), f i j) | i <- [1..n], j <- [1..n]]
```

```
    where f i j | i < n && j <  n  = p!(i,j)
                | i < n && j == n  = 0
                | j == 1 || j == n = 1
                | otherwise        = p!(i-1,j-1) + p!(i-1,j)
          p = matrizPascal2 (n-1)

-- 2ª solución
-- ===========

matrizPascal2 :: Int -> Matriz
matrizPascal2 n = listArray ((1,1),(n,n)) (concat xss)
    where yss = take n pascal
          xss = map (take n) (map (++ (repeat 0)) yss)

pascal :: [[Int]]
pascal = [1] : map f pascal
    where f xs = zipWith (+) (0:xs) (xs++[0])


-- 2ª solución
-- ===========

matrizPascal3 :: Int -> Matriz
matrizPascal3 n =
    array ((1,1),(n,n)) [((i,j), f i j) | i <- [1..n], j <- [1..n]]
    where f i j | i >=  j   = comb (i-1) (j-1)
                | otherwise = 0

-- (comb n k) es el número de combinaciones (o coeficiente binomial) de
-- n sobre k. Por ejemplo,
comb :: Int -> Int -> Int
comb n k = product [n,n-1..n-k+1] `div` product [1..k]
```

5.3.8. Examen 8 (10 de Septiembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 2ª convocatoria (10 de septiembre de 2014)
-- ---------------------------------------------------------------------

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

```
-- § Librerías auxiliares                                            --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] En una Olimpiada matemática de este año se
-- planteó el siguiente problema
--    Determinar el menor entero positivo M que tiene las siguientes
--    propiedades a la vez:
--    * El producto de los dígitos de M es 112.
--    * El producto de los dígitos de M+6 también es 112.
--
-- Definir la función
--    especiales :: Int -> Int -> [Int]
-- tal que (especiales k a) es la lista de los números naturales n tales
-- que
--    * El producto de los dígitos de n es a.
--    * El producto de los dígitos de n+k también es a.
-- Por ejemplo,
--    take 3 (especiales 8 24) == [38,138,226]
-- En efecto,   3*8 = 24,  38+8 = 46  y   4*6 = 24
--            2*2*6 = 24, 226+8 = 234 y 2*3*4 = 24
--
-- Usando la función especiales, calcular la solución del problema.
-- ---------------------------------------------------------------------

especiales :: Int -> Int -> [Int]
especiales k a =
    [n | n <- [1..], product (digitos n) == a,
                     product (digitos (n+k)) == a]

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

-- La solución del problema es
--    ghci> head (especiales 6 112)
--    2718
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] 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))
-- La dual de una expresión es la expresión obtenida intercambiando las
-- sumas y los productos. Por ejemplo, la dual de 2*(3+7) es 2+(3*7).
--
-- Definir la función
--    dual :: Expr -> Expr
-- tal que (dual e) es la dual de la expresión e. Por ejemplo,
--    dual (P (N 2) (S (N 3) (N 7)))  ==  S (N 2) (P (N 3) (N 7))
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] La sucesión con saltos se obtiene a partir de
-- los números naturales saltando 1, cogiendo 2, saltando 3, cogiendo 4,
-- saltando 5, etc. Por ejemplo,
--    (1), 2,3, (4,5,6), 7,8,9,10, (11,12,13,14,15), 16,17,18,19,20,21,
-- en la que se ha puesto entre paréntesis los números que se salta; los
-- que quedan son
--    2,3, 7,8,9,10, 16,17,18,19,20,21, ...
--
-- Definir la función
--    saltos :: [Integer]
-- tal que saltos es la lista de los términos de la sucesión con saltos.
-- Por ejemplo,
--    ghci> take 22 saltos
--    [2,3, 7,8,9,10, 16,17,18,19,20,21, 29,30,31,32,33,34,35,36, 46,47]
-- ---------------------------------------------------------------------
```

```
-- 1ª solución:
saltos :: [Integer]
saltos = aux (tail (scanl (+) 0 [1..]))
    where aux (a:b:c:ds) = [a+1..b] ++ aux (c:ds)

-- 2ª solución:
saltos2 :: [Integer]
saltos2 = aux [1..] [1..]
    where aux (m:n:ns) xs = take n (drop m xs) ++ aux ns (drop (m+n) xs)

-- 3ª solución:
saltos3 :: [Integer]
saltos3 = aux pares [1..]
    where pares             = [(x,x+1) | x <- [1,3..]]
          aux ((m,n):ps) xs = take n (drop m xs) ++ aux ps (drop (m+n) xs)

-- 4ª solución:
saltos4 :: [Integer]
saltos4 = concat (map sig pares)
    where pares     = [(x,x+1) | x <-[1,3..]]
          sig (m,n) = take n (drop (m*(m+1) `div` 2) [1..])

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] (Basado en el problema 362 del proyecto
-- Euler). El número 54 se puede factorizar de 7 maneras distintas con
-- factores mayores que 1
--    54, 2×27, 3×18, 6×9, 3×3×6, 2×3×9 y 2×3×3×3.
-- Si exigimos que los factores sean libres de cuadrados (es decir, que
-- no se puedan dividir por ningún cuadrado), entonces sólo quedan dos
-- factorizaciones
--    3×3×6 y 2×3×3×3.
--
-- Definir la función
--    factorizacionesLibresDeCuadrados :: Int -> [[Int]]
-- tal que (factorizacionesLibresDeCuadrados n) es la lista de las
-- factorizaciones de n libres de cuadrados. Por ejemplo,
--    factorizacionesLibresDeCuadrados 54  ==  [[2,3,3,3],[3,3,6]]
-- ---------------------------------------------------------------------
```

```
factorizacionesLibresDeCuadrados :: Int -> [[Int]]
factorizacionesLibresDeCuadrados n =
    [xs | xs <- factorizaciones n, listaLibreDeCuadrados xs]

-- (factorizaciones n) es la lista creciente de números mayores que 1
-- cuyo producto es n. Por ejemplo,
--    factorizaciones 12  ==  [[2,2,3],[2,6],[3,4],[12]]
--    factorizaciones 54  ==  [[2,3,3,3],[2,3,9],[2,27],[3,3,6],[3,18],[6,9],[54]]
factorizaciones ::  Int -> [[Int]]
factorizaciones n = aux n 2
    where aux 1 _ = [[]]
          aux n a = [m:xs | m <- [a..n],
                            n `rem` m == 0,
                            xs <- aux (n `div` m) m]


-- (listaLibreDeCuadrados xs) se verifica si todos los elementos de xs
-- son libres de cuadrados. Por ejemplo,
--    listaLibreDeCuadrados [3,6,15,10]  ==  True
--    listaLibreDeCuadrados [3,6,15,20]  ==  False
listaLibreDeCuadrados :: [Int] -> Bool
listaLibreDeCuadrados = all libreDeCuadrado

-- (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 5. [2 puntos] (Basado en el problema 196 del proyecto
-- Euler). Para cada número n la matriz completa de orden n es la matriz
-- cuadrada de orden n formada por los números enteros consecutivos. Por
-- ejemplo, la matriz completa de orden 3 es
--    |1 2 3|
--    |4 5 6|
--    |7 8 9|
-- las ternas primas de orden n son los listas formadas por un
```

```haskell
-- elemento de la matriz junto con dos de sus vecinos de manera que los
-- tres son primos. Por ejemplo, en la matriz anterior una terna prima
-- es [2,3,5] (formada por el elemento 2, su vecino derecho 3 y su
-- vecino inferior 5), otra es [5,2,7] (formada por el elemento 5, su
-- vecino superior 2 y su vecino inferior-izquierda 7) y otra es [5,3,7]
-- (formada por el elemento 5, su vecino superior-derecha 3 y y su
-- vecino inferior-izquierda 7).
--
-- Definir la función
--    ternasPrimasOrden :: Int -> [[Int]]
-- tal que (ternasPrimasOrden n) es el conjunto de las ternas primas de
-- la matriz completa de orden n. Por ejemplo,
--    ghci> ternasPrimasOrden 3
--    [[2,3,5],[3,2,5],[5,2,3],[5,2,7],[5,3,7]]
--    ghci> ternasPrimasOrden 4
--    [[2,3,5],[2,3,7],[2,5,7],[3,2,7],[7,2,3],[7,2,11],[7,3,11]]
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

type Matriz = Array (Int,Int) Int

ternasPrimasOrden :: Int -> [[Int]]
ternasPrimasOrden = ternasPrimas . matrizCompleta

-- (ternasPrimas p) es la lista de las ternas primas de p. Por ejemplo,
--    ghci> ternasPrimas (listArray ((1,1),(3,3)) [2,3,7,5,4,1,6,8,9])
--    [[2,3,5],[3,2,7],[3,2,5],[3,7,5],[5,2,3]]
ternasPrimas :: Matriz -> [[Int]]
ternasPrimas p =
    [xs | xs <- ternas p, all esPrimo xs]

-- (ternas p) es la lista de las ternas de p formadas por un elemento de
-- p junto con dos vecinos. Por ejemplo,
--    ghci>  ternas (listArray ((1,1),(3,3)) [2,3,7,5,4,0,6,8,9])
--     [[2,3,5],[2,3,4],[2,5,4],[3,2,7],[3,2,5],[3,2,4],[3,2,0],[3,7,5],
--      [3,7,4],[3,7,0],[3,5,4],[3,5,0],[3,4,0],[7,3,4],[7,3,0],[7,4,0],
--      [5,2,3],[5,2,4],[5,2,6],[5,2,8],[5,3,4],[5,3,6],[5,3,8],[5,4,6],
--      [5,4,8],[5,6,8],[4,2,3],[4,2,7],[4,2,5],[4,2,0],[4,2,6],[4,2,8],
```

```
--       [4,2,9],[4,3,7],[4,3,5],[4,3,0],[4,3,6],[4,3,8],[4,3,9],[4,7,5],
--       [4,7,0],[4,7,6],[4,7,8],[4,7,9],[4,5,0],[4,5,6],[4,5,8],[4,5,9],
--       [4,0,6],[4,0,8],[4,0,9],[4,6,8],[4,6,9],[4,8,9],[0,3,7],[0,3,4],
--       [0,3,8],[0,3,9],[0,7,4],[0,7,8],[0,7,9],[0,4,8],[0,4,9],[0,8,9],
--       [6,5,4],[6,5,8],[6,4,8],[8,5,4],[8,5,0],[8,5,6],[8,5,9],[8,4,0],
--       [8,4,6],[8,4,9],[8,0,6],[8,0,9],[8,6,9],[9,4,0],[9,4,8],[9,0,8]]
ternas :: Matriz -> [[Int]]
ternas p =
    [[p!(i1,j1),p!(i2,j2),p!(i3,j3)] |
     (i1,j1) <- indices p,
     ((i2,j2):ps) <- tails (vecinos (i1,j1) n),
     (i3,j3) <- ps]
    where (_,(n,_)) = bounds p

-- (vecinos (i,j) n) es la lista de las posiciones vecinas de la (i,j)
-- en una matriz cuadrada de orden n. Por ejemplo,
--    vecinos (2,3) 4  ==  [(1,2),(1,3),(1,4),(2,2),(2,4),(3,2),(3,3),(3,4)]
--    vecinos (2,4) 4  ==  [(1,3),(1,4),(2,3),(3,3),(3,4)]
--    vecinos (1,4) 4  ==  [(1,3),(2,3),(2,4)]
vecinos :: (Int,Int) -> Int -> [(Int,Int)]
vecinos (i,j) n = [(a,b) | a <- [max 1 (i-1)..min n (i+1)],
                           b <- [max 1 (j-1)..min n (j+1)],
                           (a,b) /= (i,j)]

-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo  7  ==  True
--    esPrimo 15  ==  False
esPrimo :: Int -> Bool
esPrimo n = [x | x <- [1..n], n `rem` x == 0] == [1,n]

-- (matrizCompleta n) es la matriz completa de orden n. Por ejemplo,
--    ghci> matrizCompleta 3
--    array ((1,1),(3,3)) [((1,1),1),((1,2),2),((1,3),3),
--                         ((2,1),4),((2,2),5),((2,3),6),
--                         ((3,1),7),((3,2),8),((3,3),9)]
matrizCompleta :: Int -> Matriz
matrizCompleta n =
    listArray ((1,1),(n,n)) [1..n*n]

-- 2ª definición
```

```
-- =============

ternasPrimasOrden2 :: Int -> [[Int]]
ternasPrimasOrden2 = ternasPrimas2 . matrizCompleta

ternasPrimas2 :: Matriz -> [[Int]]
ternasPrimas2 p =
    [[p!(i1,j1),p!(i2,j2),p!(i3,j3)] |
     (i1,j1) <- indices p,
     esPrimo (p!(i1,j1)),
     ((i2,j2):ps) <- tails (vecinos (i1,j1) n),
     esPrimo (p!(i2,j2)),
     (i3,j3) <- ps,
     esPrimo (p!(i3,j3))]
    where (_,(n,_)) = bounds p

-- Comparación:
--    ghci> length (ternasPrimasOrden 30)
--    51
--    (5.52 secs, 211095116 bytes)
--    ghci> length (ternasPrimasOrden2 30)
--    51
--    (0.46 secs, 18091148 bytes)
```

5.3.9. Examen 9 (20 de Noviembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 3º convocatoria (20 de noviembre de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. 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. 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. 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. 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
```

5.4. Exámenes del grupo 4 (Francisco J. Martín)

5.4.1. 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]
```

```
-- ============================================================================
```

5.4.2. 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)
-- ---------------------------------------------------------------------

import Test.QuickCheck

prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    numeroConsecutivosC xs == numeroConsecutivosC2 xs

-- ---------------------------------------------------------------------
-- 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 [1,3,5,7,9]      ==  1
--    numeroConsecutivosC [1,2,3,4,5,7,9]  ==  5
--    numeroConsecutivosC []               ==  0
--    numeroConsecutivosC [4,5]            ==  2
--    numeroConsecutivosC [4,7]            ==  1
--    numeroConsecutivosC [4,5,0,4,5,6]    ==  2
-- ---------------------------------------------------------------------

-- 1ª solución (con índices)
numeroConsecutivosC :: (Num a, Eq a) => [a] -> Int
numeroConsecutivosC xs
    | null ys   = length xs
    | otherwise = head ys
    where ys = [n | n <- [1..length xs -1], xs !! n /= 1 + xs !! (n-1)]

-- 2ª solución (con zip3)
numeroConsecutivosC2 :: (Num a, Eq a) => [a] -> Int
numeroConsecutivosC2 [] = 0
numeroConsecutivosC2 [x] = 1
numeroConsecutivosC2 [x,y] | x+1 == y  = 2
                           | otherwise = 1
```

```
numeroConsecutivosC2 xs =
    head [k | (x,y,k) <- zip3 xs (tail xs) [1..], y /= x+1]

-- 3ª solución (con takeWhile)
numeroConsecutivosC3 [] = 0
numeroConsecutivosC3 xs =
    1 + length (takeWhile (==1) [y-x | (x,y) <- zip xs (tail xs)])

-- ---------------------------------------------------------------------
-- 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,
--    numeroConsecutivosC [1,3,5,7,9]      ==  1
--    numeroConsecutivosC [1,2,3,4,5,7,9]  ==  5
--    numeroConsecutivosC []               ==  0
--    numeroConsecutivosC [4,5]            ==  2
--    numeroConsecutivosC [4,7]            ==  1
--    numeroConsecutivosC [4,5,0,4,5,6]    ==  2
-- ---------------------------------------------------------------------

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
```

5.4.3. Examen 3 (23 de Enero de 2014)

El examen es común con el del grupo 1 (ver página 337).

5.4.4. Examen 4 (20 de Marzo de 2014)

```
-- Informática (1º del Grado en Matemáticas y en Matemáticas y Estadística)
-- 4º examen de evaluación continua (20 de marzo de 2014)
-- ============================================================================

-- ----------------------------------------------------------------------------
-- Ejercicio 1. Se consideran los árboles binarios representados
-- mediante el tipo Arbol definido por
--    data Arbol = Hoja Int
--               | Nodo Int Arbol Arbol
--               deriving Show
-- Por ejemplo, el árbol
--         1
--        / \
--       /   \
--      3     2
--     / \   / \
--    5   4 6   7
-- se puede representar por
--    Nodo 1 (Nodo 3 (Hoja 5) (Hoja 4)) (Nodo 2 (Hoja 6) (Hoja 7))
-- En los ejemplos se usarán los árboles definidos por
--    ej1 = Nodo 1 (Nodo 3 (Hoja 5) (Hoja 4)) (Nodo 2 (Hoja 6) (Hoja 7))
--    ej2 = Nodo 3 (Hoja 1) (Hoja 4)
--    ej3 = Nodo 2 (Hoja 3) (Hoja 5)
--    ej4 = Nodo 1 (Hoja 2) (Nodo 2 (Hoja 3) (Hoja 3))
--    ej5 = Nodo 1 (Nodo 2 (Hoja 3) (Hoja 5)) (Hoja 2)
--
-- Las capas de un árbol binario son las listas de elementos que están a
-- la misma profundidad. Por ejemplo, las capas del árbol
--         1
--        / \
--       /   \
--      3     2
--     / \   / \
--    5   4 6   7
-- son: [1], [3,2] y [5,4,6,7]
--
-- Definir la función
--    capas :: Arbol -> [[Int]]
-- tal que (capas a) es la lista de las capas de dicho árbol ordenadas
```

```
-- según la profunidad. Por ejemplo,
--    capas ej1  ==  [[1],[3,2],[5,4,6,7]]
--    capas ej2  ==  [[3],[1,4]]
--    capas ej3  ==  [[2],[3,5]]
--    capas ej4  ==  [[1],[2,2],[3,3]]
--    capas ej5  ==  [[1],[2,2],[3,5]]
-- ---------------------------------------------------------------------

data Arbol = Hoja Int
           | Nodo Int Arbol Arbol
           deriving Show

ej1 = Nodo 1 (Nodo 3 (Hoja 5) (Hoja 4)) (Nodo 2 (Hoja 6) (Hoja 7))
ej2 = Nodo 3 (Hoja 1) (Hoja 4)
ej3 = Nodo 2 (Hoja 3) (Hoja 5)
ej4 = Nodo 1 (Hoja 2) (Nodo 2 (Hoja 3) (Hoja 3))
ej5 = Nodo 1 (Nodo 2 (Hoja 3) (Hoja 5)) (Hoja 2)

capas :: Arbol -> [[Int]]
capas (Hoja n) = [[n]]
capas (Nodo n i d) = [n] : union (capas i) (capas d)

-- (union xss yss) es la lista obtenida concatenando los
-- correspondientes elementos de xss e yss. Por ejemplo,
--    union [[3,4],[2]] [[5],[7,6,8]]  ==  [[3,4,5],[2,7,6,8]]
--    union [[3,4]]     [[5],[7,6,8]]  ==  [[3,4,5],[7,6,8]]
--    union [[3,4],[2]] [[5]]          ==  [[3,4,5],[2]]
union :: [[a]] -> [[a]] -> [[a]]
union [] yss = yss
union xss [] = xss
union (xs:xss) (ys:yss) = (xs ++ ys) : union xss yss

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un árbol es subárbol de otro si se puede establecer una
-- correspondencia de los nodos del primero con otros mayores o iguales
-- en el segundo, de forma que se respeten las relaciones de
-- descendencia. Este concepto se resume en varias situaciones posibles:
-- * El primer árbol es subárbol del hijo izquierdo del segundo
--   árbol. De esta forma ej2 es subárbol de ej1.
-- * El primer árbol es subárbol del hijo derecho del segundo árbol. De
```

```
--   esta forma ej3 es subárbol de ej1.
-- * La raíz del primer árbol es menor o igual que la del segundo, el
--   hijo izquierdo del primer árbol es subárbol del hijo izquierdo del
--   segundo y el hijo derecho del primer árbol es subárbol del hijo
--   derecho del segundo. De esta forma ej4 es subárbol de ej1.
--
-- Definir la función
--    subarbol :: Arbol -> Arbol -> Bool
-- tal que (subarbol a1 a2) se verifica si a1 es subárbol de a2. Por
-- ejemplo,
--    subarbol ej2 ej1  ==  True
--    subarbol ej3 ej1  ==  True
--    subarbol ej4 ej1  ==  True
--    subarbol ej5 ej1  ==  False
-- ---------------------------------------------------------------------

subarbol :: Arbol -> Arbol -> Bool
subarbol (Hoja n) (Hoja m) =
    n <= m
subarbol (Hoja n) (Nodo m i d) =
    n <= m || subarbol (Hoja n) i || subarbol (Hoja n) d
subarbol (Nodo _ _ _) (Hoja _) =
    False
subarbol (Nodo n i1 d1) (Nodo m i2 d2) =
    subarbol (Nodo n i1 d1) i2 ||
    subarbol (Nodo n i1 d1) d2 ||
    n <= m && (subarbol i1 i2) && (subarbol d1 d2)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1 (1.2 puntos): Definir la función
--    intercalaRep :: Eq a => a -> [a] -> [[a]]
-- tal que (intercalaRep x ys), es la lista de las listas obtenidas
-- intercalando x entre los elementos de ys, hasta la primera ocurrencia
-- del elemento x en ys. Por ejemplo,
--    intercalaRep 1 []      ==  [[1]]
--    intercalaRep 1 [1]     ==  [[1,1]]
--    intercalaRep 1 [2]     ==  [[1,2],[2,1]]
--    intercalaRep 1 [1,1]   ==  [[1,1,1]]
--    intercalaRep 1 [1,2]   ==  [[1,1,2]]
--    intercalaRep 1 [2,1]   ==  [[1,2,1],[2,1,1]]
```

```haskell
--    intercalaRep 1 [1,2,1]  ==  [[1,1,2,1]]
--    intercalaRep 1 [2,1,1]  ==  [[1,2,1,1],[2,1,1,1]]
--    intercalaRep 1 [1,1,2]  ==  [[1,1,1,2]]
--    intercalaRep 1 [1,2,2]  ==  [[1,1,2,2]]
--    intercalaRep 1 [2,1,2]  ==  [[1,2,1,2],[2,1,1,2]]
--    intercalaRep 1 [2,2,1]  ==  [[1,2,2,1],[2,1,2,1],[2,2,1,1]]
-- ---------------------------------------------------------------------

-- 1ª definición (con map):
intercalaRep :: Eq a => a -> [a] -> [[a]]
intercalaRep x [] = [[x]]
intercalaRep x (y:ys)
    | x == y    = [x:y:ys]
    | otherwise = (x:y:ys) : (map (y:) (intercalaRep x ys))

-- 2ª definición (sin map):
intercalaRep2 :: Eq a => a -> [a] -> [[a]]
intercalaRep2 x [] = [[x]]
intercalaRep2 x (y:ys)
    | x == y    = [x:y:ys]
    | otherwise = (x:y:ys) : [y:zs | zs <- intercalaRep2 x ys]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2.Definir la función
--    permutacionesRep :: Eq a => [a] -> [[a]]
-- tal que (permutacionesRep xs) es la lista (sin elementos repetidos)
-- de todas las permutaciones con repetición de la lista xs. Por
-- ejemplo,
--    permutacionesRep []         ==  [[]]
--    permutacionesRep [1]        ==  [[1]]
--    permutacionesRep [1,1]      ==  [[1,1]]
--    permutacionesRep [1,2]      ==  [[1,2],[2,1]]
--    permutacionesRep [1,2,1]    ==  [[1,2,1],[2,1,1],[1,1,2]]
--    permutacionesRep [1,1,2]    ==  [[1,1,2],[1,2,1],[2,1,1]]
--    permutacionesRep [2,1,1]    ==  [[2,1,1],[1,2,1],[1,1,2]]
--    permutacionesRep [1,1,1]    ==  [[1,1,1]]
--    permutacionesRep [1,1,2,2]  ==  [[1,1,2,2],[1,2,1,2],[2,1,1,2],
--                                     [1,2,2,1],[2,1,2,1],[2,2,1,1]]
-- ---------------------------------------------------------------------
```

```
permutacionesRep :: Eq a => [a] -> [[a]]
permutacionesRep []     = [[]]
permutacionesRep [x]    = [[x]]
permutacionesRep (x:xs) =
    concat (map (intercalaRep x) (permutacionesRep xs))

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un montón de barriles se construye apilando unos encima
-- de otros por capas, de forma que en cada capa todos los barriles
-- están apoyados sobre dos de la capa inferior y todos los barriles de
-- una misma capa están pegados unos a otros. Por ejemplo, los
-- siguientes montones son válidos:
--             _          _   _                   _
--            / \        / \ / \                 / \
--           _\_/_      _\_/_\_/_   _       _   _\_/_   _
--          / \ / \    / \ / \ / \ / \     / \ / \ / \ / \
--          \_/ \_/    \_/ \_/ \_/ \_/     \_/ \_/ \_/ \_/
--
-- y los siguientes no son válidos:
--         _   _          _       _               _   _
--        / \ / \        / \     / \             / \ / \
--        \_/_\_/_      _\_/_   _\_/_       _   _\_/_\_/
--          / \ / \    / \ / \ / \ / \     / \ / \ / \
--          \_/ \_/    \_/ \_/ \_/ \_/     \_/ \_/ \_/
--
-- Se puede comprobar que el número de formas distintas de construir
-- montones con n barriles en la base M_n viene dado por la siguiente
-- fórmula:
--
--              (n-1)
--             -------
--              \
--               \
--   M_n = 1 +    )    (n-j) * M_j
--               /
--              /
--             -------
--              j = 1
--
-- Definir la función
```

```
--    montones :: Integer -> Integer
-- tal que (montones n) es el número de formas distintas de construir
-- montones con n barriles en la base. Por ejemplo,
--    montones 1   ==  1
--    montones 10  ==  4181
--    montones 20  ==  63245986
--    montones 30  ==  956722026041
--
-- Calcular el número de formas distintas de construir montones con 50
-- barriles en la base.
-- ---------------------------------------------------------------------

montones :: Integer -> Integer
montones 1 = 1
montones n = 1 + sum [(n-j)*(montones j) | j <- [1..n-1]]

-- 2ª definición, a partir de la siguiente observación
--    M(1) = 1                          = 1
--    M(2) = 1 + M(1)                   = M(1) + M(1)
--    M(3) = 1 + 2*M(1) + M(2)          = M(2) + (M(1) + M(2))
--    M(4) = 1 + 3*M(1) + 2*M(2) + M(3) = M(3) + (M(1) + M(2) + M(3))
montones2 :: Int -> Integer
montones2 n = montonesSuc !! (n-1)

montonesSuc :: [Integer]
montonesSuc = 1 : zipWith (+) montonesSuc (scanl1 (+) montonesSuc)

-- 3ª definición
montones3 :: Integer -> Integer
montones3 0 = 0
montones3 n = head (montonesAcc [] n)

montonesAcc :: [Integer] -> Integer -> [Integer]
montonesAcc ms 0 = ms
montonesAcc ms n =
    montonesAcc ((1 + sum (zipWith (*) ms [1..])):ms) (n-1)

-- El cálculo es
--    ghci> montones2 50
--    218922995834555169026
```

5.4.5. Examen 5 (22 de Mayo de 2014)

```
-- Informática (1º del Grado en Matemáticas y en Matemáticas y Estadística)
-- 5º examen de evaluación continua (22 de mayo de 2014)
-- =========================================================================

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.Array
import GrafoConListas
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una sucesión creciente es aquella en la que todo
-- elemento es estrictamente mayor que el anterior. Una sucesión
-- super-creciente es una sucesión creciente en la que la diferencia
-- entre un elemento y el siguiente es estrictamente mayor que la
-- diferencia entre dicho elemento y el anterior. Por ejemplo, [1,2,4,7]
-- es una sucesión super-creciente, pero [1,4,8,12] no. Otra
-- caracterización de las sucesiones super-crecientes es que la sucesión
-- de las diferencias entre elementos consecutivos es creciente.
--
-- Definir, utilizando exclusivamente recursión en todas las
-- definiciones (incluyendo auxiliares), la función
--    superCrecienteR :: (Num a, Ord a) => [a] -> Bool
-- tal que (superCrecienteR xs) se verifica si la secuencia xs es
-- super-creciente. Por ejemplo,
--    superCrecienteR [1,2,4,7]   ==  True
--    superCrecienteR [1,4,8,12]  ==  False
-- ---------------------------------------------------------------------------

-- 1ª solución de superCrecienteR:
superCrecienteR :: (Num a, Ord a) => [a] -> Bool
superCrecienteR xs = crecienteR (diferenciasR xs)

crecienteR :: Ord a => [a] -> Bool
crecienteR (x1:x2:xs) = x1 < x2 && crecienteR (x2:xs)
crecienteR _          = True
```

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

-- 2ª solución de superCrecienteR:
superCrecienteR2 :: (Num a, Ord a) => [a] -> Bool
superCrecienteR2 [x1,x2]       = x1 < x2
superCrecienteR2 (x1:x2:x3:xs) = x1 < x2 && x2-x1 < x3-x2 &&
                                 superCrecienteR2 (x2:x3:xs)
superCrecienteR2 _             = True

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir sin utilizar recursión en ninguna de las definiciones
-- (incluyendo auxiliares), la función
--    superCrecienteC :: (Num a, Ord a) => [a] -> Bool
-- tal que  (superCrecienteC xs) se verifica si la secuencia xs es
-- super-creciente. Por ejemplo,
--    superCrecienteC [1,2,4,7]   ==  True
--    superCrecienteC [1,4,8,12]  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición de superCrecienteC:
superCrecienteC :: (Num a, Ord a) => [a] -> Bool
superCrecienteC xs = crecienteC (diferenciasC xs)

crecienteC :: Ord a => [a] -> Bool
crecienteC xs = and [x1 < x2 | (x1,x2) <- zip xs (tail xs)]

diferenciasC :: Num t => [t] -> [t]
diferenciasC xs = zipWith (-) (tail xs) xs

-- 2ª definición de superCrecienteC:
superCrecienteC2 :: (Num a, Ord a) => [a] -> Bool
superCrecienteC2 xs =
    and [x1 < x2 && x2-x1 < x3-x2 |
         (x1,x2,x3) <- zip3 xs (tail xs) (drop 2 xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se considera la secuencia infinita de todos los números
-- naturales.
```

```
--    [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...]
-- Si todos los números de esta secuencia se descomponen en sus dígitos,
-- se obtiene la secuencia infinita:
--    [1,2,3,4,5,6,7,8,9,1,0,1,1,1,2,1,3,1,4,1,5,1,6,1,7,1,8,1,9,2,,...]
--
-- Definir la función
--    secuenciaDigitosNaturales :: [Int]
-- tal que su valor es la secuencia infinita de los dígitos de todos los
-- elementos de la secuencia de números naturales. Por ejemplo,
--    take 11 secuenciaDigitosNaturales  ==  [1,2,3,4,5,6,7,8,9,1,0]
-- ---------------------------------------------------------------------

secuenciaDigitosNaturales :: [Int]
secuenciaDigitosNaturales = [read [c] | n <- [1..], c <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Consideremos las matrices representadas como tablas
-- cuyos índices son pares de números naturales.
--    type Matriz a = Array (Int,Int) a
-- La dimensión de una matriz es el par formado por el número de filas y
-- el número de columnas:
--    dimension :: Num a => Matriz a -> (Int,Int)
--    dimension = snd . bounds
--
-- Una matriz tridiagonal es aquella en la que sólo hay elementos distintos de
-- 0 en la diagonal principal o en las diagonales por encima y por debajo de la
-- diagonal principal. Por ejemplo,
--    ( 1 2 0 0 0 0 )
--    ( 3 4 5 0 0 0 )
--    ( 0 6 7 8 0 0 )
--    ( 0 0 9 1 2 0 )
--    ( 0 0 0 3 4 5 )
--    ( 0 0 0 0 6 7 )
--
-- Definir la función
--    creaTridiagonal :: Int -> Matriz Int
-- tal que (creaTridiagonal n) es la siguiente matriz tridiagonal
-- cuadrada con n filas y n columnas:
--    ( 1 1 0 0 0 0 ... 0  0  )
--    ( 1 2 2 0 0 0 ... 0  0  )
```

```
--    ( 0 2 3 3 0 0 ... 0  0  )
--    ( 0 0 3 4 4 0 ... 0  0  )
--    ( 0 0 0 4 5 5 ... 0  0  )
--    ( 0 0 0 0 5 6 ... 0  0  )
--    ( .................... )
--    ( 0 0 0 0 0 0 ... n  n  )
--    ( 0 0 0 0 0 0 ... n n+1 )
-- Por ejemplo,
--    ghci> creaTridiagonal 4
--    array ((1,1),(4,4)) [((1,1),1),((1,2),1),((1,3),0),((1,4),0),
--                         ((2,1),1),((2,2),2),((2,3),2),((2,4),0),
--                         ((3,1),0),((3,2),2),((3,3),3),((3,4),3),
--                         ((4,1),0),((4,2),0),((4,3),3),((4,4),4)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

dimension :: Num a => Matriz a -> (Int,Int)
dimension = snd . bounds

creaTridiagonal :: Int -> Matriz Int
creaTridiagonal n =
    array ((1,1),(n,n))
          [((i,j),valores i j) | i <- [1..n], j <- [1..n]]
    where valores i j | i == j     = i
                      | i == j+1   = j
                      | i+1 == j   = i
                      | otherwise  = 0

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    esTridiagonal :: Matriz Int -> Bool
-- tal que (esTridiagonal m) se verifica si la matriz m es tridiagonal. Por
-- ejemplo,
--    ghci> esTridiagonal (listArray ((1,1),(3,3)) [1..9])
--    False
--    ghci> esTridiagonal (creaTridiagonal 5)
--    True
-- ---------------------------------------------------------------------
```

```
esTridiagonal :: Matriz Int -> Bool
esTridiagonal m =
    and [m!(i,j) == 0 | i <- [1..p], j <- [1..q], (j < i-1 || j > i+1)]
    where (p,q) = dimension m

-- ---------------------------------------------------------------------
-- Ejercicio 6. Consideremos una implementación del TAD de los grafos,
-- por ejemplo en la que los grafos se representan mediante listas. Un
-- ejemplo de grafo es el siguiente:
--    g0 :: Grafo Int Int
--    g0 = creaGrafo D (1,6) [(1,3,2),(1,5,4),(3,5,6),(5,1,8),(5,5,10),
--                            (2,4,1),(2,6,3),(4,6,5),(4,4,7),(6,4,9)]
--
-- Definir la función
--    conectados :: Grafo Int Int -> Int -> Int -> Bool
-- tal que (conectados g v1 v2) se verifica si los vértices v1 y v2
-- están conectados en el grafo g. Por ejemplo,
--    conectados g0 1 3  ==  True
--    conectados g0 1 4  ==  False
--    conectados g0 6 2  ==  False
--    conectados g0 2 6  ==  True
-- ----------------------------------------------------------------------------

g0 :: Grafo Int Int
g0 = creaGrafo D (1,6) [(1,3,2),(1,5,4),(3,5,6),(5,1,8),(5,5,10),
                        (2,4,1),(2,6,3),(4,6,5),(4,4,7),(6,4,9)]

conectados :: Grafo Int Int -> Int -> Int -> Bool
conectados g v1 v2 = elem v2 (conectadosAux g [] [v1])

conectadosAux :: Grafo Int Int -> [Int] -> [Int] -> [Int]
conectadosAux g vs [] = vs
conectadosAux g vs (w:ws)
    | elem w vs = conectadosAux g vs ws
    | otherwise = conectadosAux g (union [w] vs) (union ws (adyacentes g w))
```

5.4.6. Examen 6 (18 de Junio de 2014)

El examen es común con el del grupo 1 (ver página 354).

5.4.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 401).

5.4.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 408).

5.4.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 413).

5.5. Exámenes del grupo 5 (Andrés Cordón y Miguel A. Martínez)

5.5.1. 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]
```

5.5.2. 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
```

5.5.3. Examen 3 (23 de Enero de 2014)

El examen es común con el del grupo 1 (ver página 337).

5.5.4. Examen 4 (19 de Marzo de 2014)

```
-- Informática (1º Grado Matemáticas y doble Grado Matemáticas y Física)
-- 4º examen de evaluación continua (19 de marzo de 2014)
-- =====================================================================
```

```
-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número se dirá ordenado si sus cifras están en orden
-- creciente. Por ejemplo, 11257 es ordenado pero 2423 no lo es.
--
-- Definir la lista
--    ordenados :: [Integer]
-- formada  por todos los enteros ordenados. Por ejemplo,
--    ghci> take 20 (dropWhile (<30) ordenados)
--    [33,34,35,36,37,38,39,44,45,46,47,48,49,55,56,57,58,59,66,67]
-- ---------------------------------------------------------------------

ordenados :: [Integer]
ordenados = [n | n <- [1..], esOrdenado n]

-- (esOrdenado x) se verifica si el número x es ordenado. Por ejemplo,
--    esOrdenado 359  ==  True
--    esOrdenado 395  ==  False
esOrdenado :: Integer -> Bool
esOrdenado = esOrdenada . show

-- (esOrdenada xs) se verifica si la lista xs está ordenada. Por
-- ejemplo,
--    esOrdenada [3,5,9]  ==  True
--    esOrdenada [3,9,5]  ==  False
--    esOrdenada "359"  ==  True
--    esOrdenada "395"  ==  False
esOrdenada :: Ord a => [a] -> Bool
esOrdenada (x:y:xs) = x <= y && esOrdenada (y:xs)
esOrdenada _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcular en qué posición de la lista aparece el número
-- 13333.
-- ---------------------------------------------------------------------
```

```
-- El cálculo es
--    ghci> length (takeWhile (<=13333) ordenados)
--    1000

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Una lista se dirá comprimida si sus elementos
-- consecutivos han sido agrupados. Por ejemplo, la comprimida de
-- "aaabcccc" es [(3,'a'),(1,'b'),(4,'c')].
--
-- Definir la función
--    comprimida :: Eq a => [a] -> [(a,Int)]
-- tal que (comprimida xs) es la comprimida de la lista xs. Por ejemplo,
--    comprimida "aaabcccc"  ==  [(3,'a'),(1,'b'),(4,'c')]
-- ---------------------------------------------------------------------

-- 2ª definición (por recursión usando takeWhile):
comprimida :: Eq a => [a] -> [(Int,a)]
comprimida [] = []
comprimida (x:xs) =
    (1 + length (takeWhile (==x) xs),x) : comprimida (dropWhile (==x) xs)

-- 2ª definición (por recursión sin takeWhile)
comprimida2 :: Eq a => [a] -> [(Int,a)]
comprimida2 xs = aux xs 1
    where aux (x:y:zs) n | x == y    = aux (y:zs) (n+1)
                         | otherwise = (n,x) : aux (y:zs) 1
          aux [x]      n             = [(n,x)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    expandida :: [(Int,a)] -> [a]
-- tal que (expandida ps) es la lista expandida correspondiente a ps (es
-- decir, es la lista xs tal que la comprimida de xs es ps). Por
-- ejemplo,
--    expandida [(2,1),(3,7),(2,5),(4,7)]  ==  [1,1,7,7,7,5,5,7,7,7,7]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
expandida :: [(Int,a)] -> [a]
```

```
expandida ps = concat [replicate k x | (k,x) <- ps]

-- 2ª definición (por recursión)
expandida2 :: [(Int,a)] -> [a]
expandida2 [] = []
expandida2 ((n,x):ps) = replicate n x ++ expandida2 ps

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Los árboles binarios pueden representarse mediante el
-- tipo de dato
--    data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
-- Un ejemplo de árbol es
--    ejArbol = Nodo 7 (Nodo 2 (Hoja 5) (Hoja 4)) (Hoja 9)
--
-- Un elemento de un árbol se dirá de nivel k si aparece en el árbol a
-- distancia k  de la raíz.
--
-- Definir la función
--    nivel :: Int -> Arbol a -> [a]
-- tal que (nivel k a) es la lista de los elementos de nivel k del árbol
-- a. Por ejemplo,
--    nivel 0 ejArbol  ==  [7]
--    nivel 1 ejArbol  ==  [2,9]
--    nivel 2 ejArbol  ==  [5,4]
--    nivel 3 ejArbol  ==  []
-- ---------------------------------------------------------------------

data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

ejArbol = Nodo 7 (Nodo 2 (Hoja 5) (Hoja 4)) (Hoja 9)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    todosDistintos :: Eq a => Arbol a -> Bool
```

```
-- tal que (todosDistintos a) se verifica si todos los elementos del
-- árbol a son distintos entre sí. Por ejemplo,
--    todosDistintos ejArbol                                       == True
--    todosDistintos (Nodo 7 (Hoja 3) (Nodo 4 (Hoja 7) (Hoja 2)))  ==  False
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. En un colisionador de partículas se disponen m placas,
-- con n celdillas cada una. Cada celdilla detecta si alguna partícula
-- ha pasado por ella (1 si ha detectado una partícula, 0 en caso
-- contrario). El siguiente ejemplo muestra 5 placas con 9 celdillas
-- cada una:
--    experimento:: [[Int]]
--    experimento = [[0, 0, 1, 1, 0, 1, 0, 0, 1],
--                   [0, 1, 0, 1, 0, 1, 0, 1, 0],
--                   [1, 0, 1, 0, 0, 0, 1, 0, 0],
--                   [0, 1, 0, 0, 0, 1, 0, 1, 0],
--                   [1, 0, 0, 0, 1, 0, 0, 0, 1]]
-- Se quiere reconstruir las trayectorias que han realizado las
-- partículas que  atraviesan dichas placas.
--
-- Una trayectoria de una partícula vendrá dada por un par, donde la
-- primera componente indica la celdilla por la que pasó en la primera
-- placa y la segunda componente será una lista que indica el camino
-- seguido en las sucesivas placas. Es decir, cada elemento de la lista
-- indicará si de una placa a la siguiente, la partícula se desvió una
-- celdilla hacia la derecha (+1), hacia la izquierda (-1) o pasó por la
-- misma celdilla (0). Por ejemplo, una trayectoria en el ejemplo
-- anterior sería:
--        [(2,[-1,1,-1,-1])]
-- Se puede observar que es posible crear más de una trayectoria para la
-- misma  partícula.
```

```
--
-- Definir la función
--    calculaTrayectorias :: [[Int]] -> [(Int,[Int])]
-- que devuelva una lista con todas las trayectorias posibles para un
-- experimento. Por ejemplo,
--    ghci> calculaTrayectorias experimento
--    [(2,[-1,-1,1,-1]),  (2,[-1,1,-1,-1]), (2,[1,-1,-1,-1]),
--     (3,[0,-1,-1,-1]),
--     (5,[0,1,-1,-1]),   (5,[0,1,1,1]),
--     (8,[-1,-1,-1,-1]), (8,[-1,-1,1,1])]
-- ---------------------------------------------------------------------

experimento:: [[Int]]
experimento = [[0, 0, 1, 1, 0, 1, 0, 0, 1],
               [0, 1, 0, 1, 0, 1, 0, 1, 0],
               [1, 0, 1, 0, 0, 0, 1, 0, 0],
               [0, 1, 0, 0, 0, 1, 0, 1, 0],
               [1, 0, 0, 0, 1, 0, 0, 0, 1]]

calculaTrayectorias :: [[Int]] -> [(Int,[Int])]
calculaTrayectorias [] = []
calculaTrayectorias (xs:xss) =
    [(i,ys) | (i,e) <- zip [0..] xs, e==1, ys <- posiblesTrayectorias i xss]

-- Solución 1, con recursión
posiblesTrayectorias :: Int -> [[Int]] -> [[Int]]
posiblesTrayectorias i [] = [[]]
posiblesTrayectorias i (xs:xss) =
    [desp:ys | desp <- [-1,0,1],
               i+desp >= 0 && i+desp < length xs,
               xs!!(i+desp) == 1,
               ys <- posiblesTrayectorias (i+desp) xss]

-- Solución 2, con recursión con acumulador
posiblesTrayectorias' i xss = posiblesTrayectoriasRecAcum i [[]] xss

posiblesTrayectoriasRecAcum i yss [] = yss
posiblesTrayectoriasRecAcum i yss (xs:xss) =
  concat[posiblesTrayectoriasRecAcum idx [ys++[idx-i] | ys <- yss] xss
      | idx <- [i-1..i+1], idx >= 0 && idx < length xs, xs!!idx == 1]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Consideraremos una trayectoria válida si no cambia de
-- dirección. Esto es, si una partícula tiene una trayectoria hacia la
-- izquierda, no podrá desviarse a la derecha posteriormenete y vice versa.
--
-- Definir la función
--    trayectoriasValidas :: [(Int,[Int])] -> [(Int,[Int])]
-- tal que (trayectoriasValidas xs) es la lista de las trayectorias de
-- xs que son válidas. Por ejemplo,
--    ghci> trayectoriasValidas (calculaTrayectorias experimento)
--    [(3,[0,-1,-1,-1]),(5,[0,1,1,1]),(8,[-1,-1,-1,-1])]
-- ---------------------------------------------------------------------

trayectoriasValidas :: [(Int,[Int])] -> [(Int,[Int])]
trayectoriasValidas xss = [(i,xs) | (i,xs) <- xss, trayectoriaValida3 xs]

-- 1ª definición (con recursión)
trayectoriaValida :: [Int] -> Bool
trayectoriaValida [] = True
trayectoriaValida (x:xs) | x == 0  = trayectoriaValida xs
                         | x == -1 = notElem 1 xs
                         | x == 1  = notElem (-1) xs

-- 2ª definición (con operaciones lógicas)
trayectoriaValida2 :: [Int] -> Bool
trayectoriaValida2 xs = (dirDer `xor` dirIzq) || dirRec
    where xor x y = x/=y
          dirDer  = elem 1 xs
          dirIzq  = elem (-1) xs
          dirRec  = elem 0 xs && not dirDer && not dirIzq

-- 3ª definición
trayectoriaValida3 :: [Int] -> Bool
trayectoriaValida3 xs = not (1 `elem` xs && (-1) `elem` xs)
```

5.5.5. Examen 5 (21 de Mayo de 2014)

```
-- Informática (1º Grado Matemáticas y doble Grado Matemáticas y Física)
-- 5º examen de evaluación continua (21 de mayo de 2014)
-- =====================================================================
```

```
-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    subcadena :: String -> String -> Int
-- tal que (subcadena xs ys) es el número de veces que aparece la cadena
-- xs dentro de la cadena ys. Por ejemplo,
--    subcadena "abc" "abc1ab1c1abc1" == 2
--    subcadena "abc" "a1b1bc"        == 0
-- ---------------------------------------------------------------------

subcadena :: String -> String -> Int
subcadena _ [] = 0
subcadena xs cs@(y:ys)
    | xs == take n cs = 1 + subcadena xs (drop n cs)
    | otherwise       = subcadena xs ys
    where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    ocurrencias :: Int -> (Integer -> Integer) -> Integer -> Int
-- tal que (ocurrencias n f x) es el número de veces que aparecen las
-- cifras de n como subcadena de las cifras del número f(x). Por
-- ejemplo,
--    ocurrencias 0 (1+) 399          == 2
--    ocurrencias 837 (2^) 1000       == 3
-- ---------------------------------------------------------------------

ocurrencias :: Int -> (Integer -> Integer) -> Integer -> Int
ocurrencias n f x = subcadena (show n) (show (f x))

-- ----------------------------------------------------------------------
-- Ejercicio 2.1. Representamos árboles binarios mediante el tipo de
-- dato
--    data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

```
-- Por ejemplo, los árboles
--        5           5
--       / \         / \
--      5   3       5   3
--     / \         / \
--    2   5       2   5
-- se representan por
--    arbol1, arbol2 :: Arbol Int
--    arbol1 = (Nodo 5 (Nodo 5 (Hoja 2) (Hoja 5)) (Hoja 3))
--    arbol2 = (Nodo 5 (Nodo 5 (Hoja 2) (Hoja 1)) (Hoja 3))
--
-- Definir el predicado
--    ramaIgual :: Eq a => Arbol a -> Bool
-- tal que (ramaIgual t) se verifica si el árbol t contiene, al menos,
-- una rama con todos sus elementos iguales. Por ejemplo,
--    ramaIgual arbol1 == True
--    ramaIgual arbol2 == False
-- ---------------------------------------------------------------------

data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

arbol1, arbol2 :: Arbol Int
arbol1 = (Nodo 5 (Nodo 5 (Hoja 2) (Hoja 5)) (Hoja 3))
arbol2 = (Nodo 5 (Nodo 5 (Hoja 2) (Hoja 1)) (Hoja 3))

ramaIgual :: Eq a => Arbol a -> Bool
ramaIgual t = or [todosI xs | xs <- ramas t]

-- (ramas t) es la lista de las ramas del árbol t. Por ejemplo,
--    ramas arbol1  ==  [[5,5,2],[5,5,5],[5,3]]
--    ramas arbol2  ==  [[5,5,2],[5,5,1],[5,3]]
ramas (Hoja x)     = [[x]]
ramas (Nodo x i d) = map (x:) (ramas i ++ ramas d)

-- (todosI xs) se verifica si todos los elementos de xs son iguales. Por
-- ejemplo,
--    todosI [6,6,6]  ==  True
--    todosI [6,9,6]  ==  False
todosI :: Eq a => [a] -> Bool
todosI (x:y:xs) = x == y && todosI (y:xs)
```

```
todosI _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 2.2 Definir el predicado
--    ramasDis :: Eq a => Arbol a -> Bool
-- tal que (ramasDis t) se verifica si todas las ramas del árbol t
-- contienen, al menos, dos elementos distintos. Por ejemplo:
--    ramasDis arbol2 == True
--    ramasDis arbol1  ==  False
-- ---------------------------------------------------------------------

ramasDis :: Eq a => Arbol a -> Bool
ramasDis = not . ramaIgual

-- ----------------------------------------------------------------------
-- Ejercicio 3. Representamos polinomios en una variable mediante el
-- tipo de dato
--    data Pol a = PolCero | Cons Int a (Pol a) deriving Show
-- Por ejemplo, el polinomio 3x^5 - x^3 + 7x^2 se representa por
--    pol :: Pol Int
--    pol = Cons 5 3 (Cons 3 (-1) (Cons 2 7 PolCero))
--
-- Definir la función
--    derivadaN :: Num a => Int -> Pol a -> Pol a
-- tal que (derivadaN n p) es la derivada n-ésima del polinomio p. Por
-- ejemplo,
--    derivadaN 1 pol  ==  Cons 4 15 (Cons 2 (-3) (Cons 1 14 PolCero))
--    derivadaN 2 pol  ==  Cons 3 60 (Cons 1 (-6) (Cons 0 14 PolCero))
--    derivadaN 3 pol  ==  Cons 2 180 (Cons 0 (-6) PolCero)
--    derivadaN 1 (Cons 5 3.2 PolCero)  ==  Cons 4 16.0 PolCero
-- ---------------------------------------------------------------------

data Pol a = PolCero | Cons Int a (Pol a) deriving Show

pol :: Pol Int
pol = Cons 5 3 (Cons 3 (-1) (Cons 2 7 PolCero))

derivadaN :: Num a => Int -> Pol a -> Pol a
derivadaN n p = (iterate derivada p)!!n
```

```
derivada :: Num a => Pol a -> Pol a
derivada PolCero      = PolCero
derivada (Cons 0 x p) = PolCero
derivada (Cons n x p) = Cons (n-1) (x * fromIntegral n) (derivada p)

-- ---------------------------------------------------------------------
-- Ejercicio 4. El algoritmo de Jacobi se utiliza para calcular el
-- gradiente de temperatura en una malla de cuerpos dispuestos en dos
-- dimensiones. Se emplea para ello una matriz con el siguiente contenido:
--    a) Se define una frontera, que son los elementos de la primera fila,
--       primera columna, última fila y última columna. Estos elementos
--       indican la temperatura exterior, y su valor es siempre constante.
--    b) Los elementos del interior indican la temperatura de cada
--       cuerpo.
-- En cada iteración del algoritmo la matriz p se transforma en otra q,
-- de la misma dimensión, cuyos elementos son:
--    a) Elementos de la frontera:
--          q(i,j)=p(i,j).
--    b) Elementos del interior:
--          q(i,j)=0.2*(p(i,j)+p(i+1,j)+p(i-1,j)+p(i,j+1)+p(i,j-1))
-- Por ejemplo, la transformada de la matriz de la izquierda es la de la
-- derecha
--    |2, 2, 2, 2, 2|          |2.0, 2.0, 2.0, 2.0, 2.0|
--    |2, 0, 0, 0, 2|          |2.0, 0.8, 0.4, 0.8, 2.0|
--    |2, 0, 0, 0, 2|          |2.0, 0.4, 0.0, 0.4, 2.0|
--    |2, 0, 0, 0, 2|          |2.0, 0.8, 0.4, 0.8, 2.0|
--    |2, 2, 2, 2, 2|          |2.0, 2.0, 2.0, 2.0, 2.0|
--
-- Representaremos la matriz con el tipo de dato:
--    type Matriz = Array (Int,Int) Float
-- La dos matrices anteriores se representan por
--    matriz1, matriz2 :: Matriz
--    matriz1 = listArray ((1,1),(5,5)) ([2, 2, 2, 2, 2,
--                                        2, 0, 0, 0, 2,
--                                        2, 0, 0, 0, 2,
--                                        2, 0, 0, 0, 2,
--                                        2, 2, 2, 2, 2])
--    matriz2 = listArray ((1,1),(5,5)) ([2.0, 2.0, 2.0, 2.0, 2.0,
--                                        2.0, 0.8, 0.4, 0.8, 2.0,
--                                        2.0, 0.4, 0.0, 0.4, 2.0,
```

```
--                                        2.0, 0.8, 0.4, 0.8, 2.0,
--                                        2.0, 2.0, 2.0, 2.0, 2.0])
--
-- Definir la función
--    iteracion_jacobi:: Matriz -> Matriz
-- tal que (iteracion_jacobi p) es la matriz obtenida aplicándole una
-- transformación de Jacobi a la matriz p. Por ejemplo,
--    iteracion_jacobi matriz1  ==  matriz2
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Float

matriz1 :: Matriz
matriz1 = listArray ((1,1),(5,5)) ([2, 2, 2, 2, 2,
                                    2, 0, 0, 0, 2,
                                    2, 0, 0, 0, 2,
                                    2, 0, 0, 0, 2,
                                    2, 2, 2, 2, 2])

matriz2 :: Matriz
matriz2 = listArray ((1,1),(5,5)) ([2.0, 2.0, 2.0, 2.0, 2.0,
                                    2.0, 0.8, 0.4, 0.8, 2.0,
                                    2.0, 0.4, 0.0, 0.4, 2.0,
                                    2.0, 0.8, 0.4, 0.8, 2.0,
                                    2.0, 2.0, 2.0, 2.0, 2.0])

-- 1ª definición:
iteracion_jacobi :: Matriz -> Matriz
iteracion_jacobi p = array ((1,1),(n,m)) [((i,j), f i j) | i <- [1..n], j<-[1..m]]
    where (_,(n,m)) = bounds p
          f i j | frontera (i,j) = p!(i,j)
                | otherwise      = 0.2*(p!(i,j)+p!(i+1,j)+p!(i-1,j)+p!(i,j+1)+p!(i,j-
          frontera (i,j) = i == 1 || i == n || j == 1 || j == m

-- 2ª definición:
iteracion_jacobi2 :: Matriz -> Matriz
iteracion_jacobi2 p =
    array ((1,1),(n,m))
          ([((i,j), 0.2*(p!(i,j)+p!(i+1,j)+p!(i-1,j)+p!(i,j+1)+p!(i,j-1))) |
            i <- [2..n-1], j <- [2..m-1]] ++
```

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

5.5.6. Examen 6 (18 de Junio de 2014)

El examen es común con el del grupo 1 (ver página 354).

5.5.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 401).

5.5.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 408).

5.5.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 413).

6

Exámenes del curso 2014–15

6.1. Exámenes del grupo 1 (Francisco J. Martín)

6.1.1. Examen 1 (3 de Noviembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (3 de noviembre de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. El módulo de un vector de n dimensiones,
-- xs = (x1,x2,...,xn), es la raíz cuadrada de la suma de los cuadrados
-- de sus componentes. Por ejemplo,
--    el módulo de (1,2)   es 2.236068
--    el módulo de (1,2,3) es 3.7416575
--
-- El normalizado de un vector de n dimensiones, xs = (x1,x2,...,xn), es
-- el vector que se obtiene dividiendo cada componente por su módulo. De
-- esta forma, el módulo del normalizado de un vector siempre es 1. Por
-- ejemplo,
--    el normalizado de (1,2)   es (0.4472136,0.8944272)
--    el normalizado de (1,2,3) es (0.26726124,0.5345225,0.8017837)
--
-- Definir, por comprensión, la función
--    modulo :: [Float] -> Float
-- tal que (modulo xs) es el módulo del vector xs. Por ejemplo,
--    modulo [1,2]      ==  2.236068
--    modulo [1,2,3]    ==  3.7416575
--    modulo [1,2,3,4]  ==  5.477226
```

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

modulo :: [Float] -> Float
modulo xs = sqrt (sum [x^2 | x <- xs])

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por comprensión, la función
--    normalizado :: [Float] -> [Float]
-- tal que (normalizado xs) es el vector resultado de normalizar el
-- vector xs. Por ejemplo,
--    normalizado [1,2]      ==  [0.4472136,0.8944272]
--    normalizado [1,2,3]    ==  [0.26726124,0.5345225,0.8017837]
--    normalizado [1,2,3,4]  ==  [0.18257418,0.36514837,0.5477225,0.73029673]
-- ---------------------------------------------------------------------

normalizado :: [Float] -> [Float]
normalizado xs = [x / modulo xs | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. En un bloque de pisos viven "Ana", "Beatriz", "Carlos"
-- y "Daniel", cada uno de ellos tiene ciertos alimentos en sus
-- respectivas despensas. Esta información está almacenada en una lista
-- de asociación de la siguiente forma: (<nombre>,<despensa>)
--    datos = [("Ana",["Leche","Huevos","Sal"]),
--             ("Beatriz",["Jamon","Lechuga"]),
--             ("Carlos",["Atun","Tomate","Jamon"]),
--             ("Daniel",["Salmon","Huevos"])]
--
-- Definir, por comprensión, la función
--    tienenProducto :: String -> [(String,[String])] -> [String]
-- tal que (tienenProducto x ys) es la lista de las personas que tienen el
-- producto x en sus despensas. Por ejemplo,
--    tienenProducto "Lechuga" datos  ==  ["Beatriz"]
--    tienenProducto "Huevos"  datos  ==  ["Ana","Daniel"]
--    tienenProducto "Pan"     datos  ==  []
-- ---------------------------------------------------------------------

datos = [("Ana",["Leche","Huevos","Sal"]),
         ("Beatriz",["Jamon","Lechuga"]),
         ("Carlos",["Atun","Tomate","Jamon"]),
```

```
         ("Daniel",["Salmon","Huevos"])]

tienenProducto :: String -> [(String,[String])] -> [String]
tienenProducto x ys = [z | (z,ds) <- ys, x `elem` ds]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por comprensión, la función
--    proveedores :: String -> [(String,[String])] -> [String]
-- tal que (proveedores xs ys) es la lista de las personas que pueden
-- proporcionar algún producto de los de la lista xs. Por ejemplo,
--    proveedores ["Leche","Jamon"] datos == ["Ana","Beatriz","Carlos"]
--    proveedores ["Sal","Atun"]    datos == ["Ana","Carlos"]
--    proveedores ["Leche","Sal"]   datos == ["Ana"]
-- ---------------------------------------------------------------------

proveedores :: [String] -> [(String,[String])] -> [String]
proveedores xs ys =
    [z | (z,ds) <- ys, not (null [d | d <- ds, d `elem` xs])]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión, la función
--    intercalaNumeros :: Integer -> Integer -> Integer
-- tal que (intercalaNumeros n m) es el número resultante de
-- "intercalar" las cifras de los números 'n' y 'm'. Por ejemplo, el
-- resultado de intercalar las cifras de los números 123 y 768 sería:
--      1 2 3
--       7 6 8
--    ----------
--      172638
-- Si uno de los dos números tiene más cifras que el otro, simplemente
-- se ponen todas al principio en el mismo orden. Por ejemplo, el
-- resultado de intercalar las cifras de los números 1234 y 56 sería:
--      1 2 3 4
--           5 6
--    ------------
--      1 2 3546
-- De esta forma:
--    intercalaNumeros 123 768  ==  172638
--    intercalaNumeros 1234 56  ==  123546
--    intercalaNumeros 56 1234  ==  125364
```

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

-- 1ª definición:
intercalaNumeros :: Integer -> Integer -> Integer
intercalaNumeros 0 y = y
intercalaNumeros x 0 = x
intercalaNumeros x y =
    let rx = mod x 10
        dx = div x 10
        ry = mod y 10
        dy = div y 10
    in (intercalaNumeros dx dy)*100 + rx*10 + ry

-- 2ª definición:
intercalaNumeros2 :: Integer -> Integer -> Integer
intercalaNumeros2 0 y = y
intercalaNumeros2 x 0 = x
intercalaNumeros2 x y = 100 * intercalaNumeros2 dx dy + 10*rx + ry
    where (dx,rx) = divMod x 10
          (dy,ry) = divMod y 10

-- ---------------------------------------------------------------------
-- Ejercicio 4. La carga de una lista es el número de elementos
-- estrictamente positivos menos el número de elementos estrictamente
-- negativos.
--
-- Definir, por recursión, la función
--    carga :: [Integer] -> Integer
-- tal que (carga xs) es la carga de la lista xs. Por ejemplo,
--    carga [1,2,0,-1]     ==  1
--    carga [1,0,2,0,3]    ==  3
--    carga [1,0,-2,0,3]   ==  1
--    carga [1,0,-2,0,-3]  ==  -1
--    carga [1,0,-2,2,-3]  ==  0
-- ---------------------------------------------------------------------

carga :: [Integer] -> Integer
carga [] = 0
carga (x:xs)
    | x > 0     = 1 + carga xs
```

```
    | x == 0    = carga xs
    | otherwise = carga xs - 1
```

6.1.2. Examen 2 (1 de Diciembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (1 de diciembre de 2014)
-- =========================================================================

-- -------------------------------------------------------------------------
-- § Librería auxiliar                                                    --
-- -------------------------------------------------------------------------

import Test.QuickCheck

-- -------------------------------------------------------------------------
-- Ejercicio 1.1. Decimos que una lista está equilibrada con respecto a
-- una propiedad si el número de elementos de la lista que cumplen la
-- propiedad es igual al número de elementos de la lista que no la
-- cumplen. Por ejemplo, la lista [1,2,3,4,5,6,7,8] está equilibrada con
-- respecto a la propiedad 'ser par' y con respecto a la propiedad 'ser
-- primo', pero no con respecto a la propiedad 'ser múltiplo de 3'.
--
-- Definir, por comprensión, la función
--    listaEquilibradaC :: [a] -> (a -> Bool) -> Bool
-- tal que (listaEquilibradaC xs p) se verifica si la lista xs está
-- equilibrada con respecto a la propiedad p. Por ejemplo,
--    listaEquilibradaC [1..8] even                  ==  True
--    listaEquilibradaC [1..8] (\x -> mod x 3 == 0)  ==  False
-- -------------------------------------------------------------------------

listaEquilibradaC :: [a] -> (a -> Bool) -> Bool
listaEquilibradaC xs p =
    length [x | x <- xs, p x] == length [x | x <- xs, not (p x)]

-- -------------------------------------------------------------------------
-- Ejercicio 1.2. Definir, usando funciones de orden superior, la
-- función
--    listaEquilibradaS :: [a] -> (a -> Bool) -> Bool
```

```
-- tal que (listaEquilibradaS xs p) se verifica si la lista xs está
-- equilibrada con respecto a la propiedad 'p'. Por ejemplo,
--    listaEquilibradaS [1..8] even                     ==  True
--    listaEquilibradaS [1..8] (\ x -> mod x 3 == 0)  ==  False
-- ---------------------------------------------------------------------

listaEquilibradaS :: [a] -> (a -> Bool) -> Bool
listaEquilibradaS xs p =
    length (filter p xs) == length (filter (not . p) xs)

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que la longitud de las listas
-- que están equilibradas respecto a la propiedad 'ser impar' es pae
-- ---------------------------------------------------------------------

-- La propiedad es
prop_listaEquilibradaImpar :: [Int] -> Property
prop_listaEquilibradaImpar xs =
    listaEquilibradaC xs odd ==> even (length xs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por recursión, la función
--    diferenciasParidadR :: [Int] -> [Int]
-- tal que (diferenciasParidadR xs) es la lista de las diferencias entre
-- elementos consecutivos de xs que tengan la misma paridad. Por ejemplo,
--    diferenciasParidadR [1,2,3,4,5,6,7,8]   ==  []
--    diferenciasParidadR [1,2,4,5,9,6,12,9]  ==  [2,4,6]
--    diferenciasParidadR [1,7,3]             ==  [6,-4]
-- ---------------------------------------------------------------------

-- 1ª definición:
diferenciasParidadR :: [Int] -> [Int]
diferenciasParidadR (x1:x2:xs)
    | even x1 == even x2 = x2-x1 : diferenciasParidadR (x2:xs)
    | otherwise          = diferenciasParidadR (x2:xs)
diferenciasParidadR _    = []
```

```
-- 2ª definición:
diferenciasParidadR2 :: [Int] -> [Int]
diferenciasParidadR2 xs = aux (zip xs (tail xs))
    where aux [] = []
          aux ((x,y):ps) | even x == even y = y-x : aux ps
                         | otherwise        = aux ps

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, usando plegado con flodr, la función
--    diferenciasParidadP :: [Int] -> [Int]
-- tal que (diferenciasParidadP xs) es la lista de las diferencias entre
-- elementos consecutivos de xs que tengan la misma paridad. Por ejemplo,
--    diferenciasParidadP [1,2,3,4,5,6,7,8]   ==  []
--    diferenciasParidadP [1,2,4,5,9,6,12,9]  ==  [2,4,6]
--    diferenciasParidadP [1,7,3]             ==  [6,-4]
-- ---------------------------------------------------------------------

-- 1ª definición:
diferenciasParidadP :: [Int] -> [Int]
diferenciasParidadP xs =
    foldr (\ (x,y) r -> if even x == even y
                        then y-x : r
                        else r) [] (zip xs (tail xs))

-- 2ª definición:
diferenciasParidadP2 :: [Int] -> [Int]
diferenciasParidadP2 xs =
    foldr f [] (zip xs (tail xs))
    where f (x,y) r | even x == even y = y-x : r
                    | otherwise        = r

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que todos los elementos de la
-- lista de las diferencias entre elementos consecutivos de xs que
-- tengan igual paridad son pares.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_diferenciasParidad :: [Int] -> Bool
```

```
prop_diferenciasParidad xs =
    all even (diferenciasParidadP xs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. La sucesión generalizada de Fibonacci de grado N
-- (N >= 1) se construye comenzando con el número 1 y calculando el
-- resto de términos como la suma de los N términos anteriores (si
-- existen). Por ejemplo,
-- * la sucesión generalizada de Fibonacci de grado 2 es:
--      1, 1, 2, 3, 5, 8, 13, 21, 34, 55
-- * la sucesión generalizada de Fibonacci de grado 4 es:
--      1, 1, 2, 4, 8, 15, 29, 56, 108, 208
-- * la sucesión generalizada de Fibonacci de grado 6 es:
--      1, 1, 2, 4, 8, 16, 32, 63, 125, 248
--
-- Ejercicio 3.1. Definir, por recursión con acumulador, la función
--    fibsGenAR :: Int -> Int -> Int
-- tal que (fibsGenAR n m) es el término m de la sucesión generalizada
-- de Fibonacci de grado n. Por ejemplo,
--    fibsGenAR 4 3  ==  4
--    fibsGenAR 4 4  ==  8
--    fibsGenAR 4 5  ==  15
--    fibsGenAR 4 6  ==  29
--    fibsGenAR 4 7  ==  56
-- ---------------------------------------------------------------------

fibsGenAR :: Int -> Int -> Int
fibsGenAR = fibsGenARAux [1]

fibsGenARAux :: [Int] -> Int -> Int -> Int
fibsGenARAux ac _ 0 = head ac
fibsGenARAux ac n m =
    fibsGenARAux (sum (take n ac) : ac) n (m-1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, usando plegado con foldl, la función
```

```
--    fibsGenAP :: Int -> Int -> Int
-- tal que (fibsGenAP n m) es el término m de la sucesión generalizada
-- de Fibonacci de grado n. Por ejemplo,
--    fibsGenAP 4 3  ==  4
--    fibsGenAP 4 4  ==  8
--    fibsGenAP 4 5  ==  15
--    fibsGenAP 4 6  ==  29
--    fibsGenAP 4 7  ==  56
-- ---------------------------------------------------------------------

fibsGenAP :: Int -> Int -> Int
fibsGenAP n m =
    head (foldl (\ac x -> (sum (take n ac): ac)) [1] [1..m])

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por recursión, la función
--    fibsGen :: Int -> [Int]
-- tal que (fibsGen n) es la sucesión (infinita) generalizada de
-- Fibonacci de grado n. Por ejemplo,
--    take 10 (fibsGen 2)  ==  [1,1,2,3,5,8,13,21,34,55]
--    take 10 (fibsGen 4)  ==  [1,1,2,4,8,15,29,56,108,208]
--    take 10 (fibsGen 6)  ==  [1,1,2,4,8,16,32,63,125,248]
-- ---------------------------------------------------------------------

fibsGen :: Int -> [Int]
fibsGen = fibsGenAux [1]

fibsGenAux :: [Int] -> Int -> [Int]
fibsGenAux ac n = head ac : fibsGenAux (sum bc : bc) n
    where bc = take n ac
```

6.1.3. Examen 3 (23 de enero de 2015)

El examen es común con el del grupo 5 (ver página 562).

6.1.4. Examen 4 (16 de marzo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (16 de marzo de 2015)
-- ---------------------------------------------------------------------
```

```
-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.Char
import Data.Array
import I1M.Pol
import Data.Numbers.Primes
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Se dice que dos números naturales son parientes si
-- tienen exactamente un factor primo en común, independientemente de su
-- multiplicidad. Por ejemplo,
-- + Los números 12 (2^2*3) y 40 (2^3*5) son parientes, pues tienen al 2
--   como único factor primo en común.
-- + Los números 49 (7^2) y 63 (3^2*7) son parientes, pues tienen al 7
--   como único factor primo en común.
-- + Los números 12 (2^2*3) y 30 (2*3*5) no son parientes, pues tienen
--   dos factores primos en común.
-- + Los números 49 (7^2) y 25 (5^2) no son parientes, pues no tienen
--   factores primos en común.
--
-- Se dice que una lista de números naturales es una secuencia de
-- parientes si cada par de números consecutivos son parientes. Por ejemplo,
-- + La lista [12,40,35,28] es una secuencia de parientes.
-- + La lista [12,30,21,49] no es una secuencia de parientes.
--
-- Definir la función
--    secuenciaParientes :: [Int] -> Bool
-- tal que (secuenciaParientes xs) se verifica si xs es una secuencia de
-- parientes. Por ejemplo,
--    secuenciaParientes [12,40,35,28]  ==  True
--    secuenciaParientes [12,30,21,49]  ==  False
-- ----------------------------------------------------------------------------

parientes :: Int -> Int -> Bool
parientes x y =
    length [p | p <- takeWhile (<= d) primes, d `mod` p == 0] == 1
    where d = gcd x y
```

```
-- Definiciones de secuenciaParientes
-- ==================================

-- 1ª definición (por recursión)
secuenciaParientes :: [Int] -> Bool
secuenciaParientes []         = True
secuenciaParientes [x]        = True
secuenciaParientes (x1:x2:xs) =
    parientes x1 x2 && secuenciaParientes (x2:xs)

-- 2ª definición (por recursión con 2 ecuaciones)
secuenciaParientes2 :: [Int] -> Bool
secuenciaParientes2 (x1:x2:xs) =
    parientes x1 x2 && secuenciaParientes2 (x2:xs)
secuenciaParientes2 _         = True

-- 3ª definición (sin recursión):
secuenciaParientes3 :: [Int] -> Bool
secuenciaParientes3 xs = all (\(x,y) -> parientes x y) (zip xs (tail xs))

-- 4ª definición
secuenciaParientes4 :: [Int] -> Bool
secuenciaParientes4 xs = all (uncurry parientes) (zip xs (tail xs))

-- Equivalencia de las 4 definiciones
prop_secuenciaParientes :: [Int] -> Bool
prop_secuenciaParientes xs =
    secuenciaParientes2 xs == ys &&
    secuenciaParientes3 xs == ys &&
    secuenciaParientes4 xs == ys
    where ys = secuenciaParientes xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. En lógica temporal la expresión AFp significa que en
-- algún momento en el futuro se cumple la propiedad p. Trasladado a
```

```
-- su interpretación en forma de árbol lo que quiere decir es que en
-- todas las ramas (desde la raíz hasta una hoja) hay un nodo que cumple
-- la propiedad p.
--
-- Consideramos el siguiente tipo algebraico de los árboles binarios:
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                   deriving (Show,Eq)
-- y el siguiente árbol
--    a1 :: Arbol Int
--    a1 = N 9 (N 3 (H 2) (N 4 (H 1) (H 5))) (H 8)
-- En este árbol se cumple (AF par); es decir, en todas las ramas hay un
-- número  par; pero no se cumple (AF primo); es decir, hay ramas en las
-- que no hay ningún número primo. Donde una rama es la secuencia de
-- nodos desde el nodo inicial o raíz hasta una hoja.
--
-- Definir la función
--    propiedadAF :: (a -> Bool) -> Arbol a -> Bool
-- tal que (propiedadAF p a) se verifica si se cumple (AF p) en el árbol
-- a; es decir, si en todas las ramas hay un nodo (interno u hoja) que
-- cumple la propiedad p. Por ejemplo
--    propiedadAF even a1     ==  True
--    propiedadAF isPrime a1  ==  False
-- ---------------------------------------------------------------------

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

a1 :: Arbol Int
a1 = N 9 (N 3 (H 2) (N 4 (H 1) (H 5))) (H 8)

propiedadAF :: (a -> Bool) -> Arbol a -> Bool
propiedadAF p (H a)     = p a
propiedadAF p (N a i d) = p a || (propiedadAF p i && propiedadAF p d)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideramos las matrices representadas como tablas
-- cuyos índices son pares de números naturales.
--    type Matriz a = Array (Int,Int) a
```

```
--
-- Una matriz cruzada es una matriz cuadrada en la que sólo hay elementos
-- distintos de 0 en las diagonales principal y secundaria. Por ejemplo
--    | 1 0 0 0 3 |     | 1 0 0 3 |
--    | 0 2 0 1 0 |     | 0 2 3 0 |
--    | 0 0 3 0 0 |     | 0 4 5 0 |
--    | 0 2 0 1 0 |     | 2 0 0 3 |
--    | 1 0 0 0 3 |
--
-- Definir la función
--    creaCruzada :: Int -> Matriz Int
-- tal que (creaCruzada n) es la siguiente matriz cruzada con n filas y n
-- columnas:
--    | 1  0   0  ...  0   0  1 |
--    | 0  2   0  ...  0   2  0 |
--    | 0  0   3  ...  3   0  0 |
--    | ....................... |
--    | 0  0  n-2 ... n-2  0  0 |
--    | 0 n-1  0  ...  0  n-1 0 |
--    | n  0   0  ...  0   0  n |
-- Es decir, los elementos de la diagonal principal son [1,...,n], en
-- orden desde la primera fila hasta la última; y los elementos de la
-- diagonal secundaria son [1,...,n], en orden desde la primera fila
-- hasta la última.
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

creaCruzada :: Int -> Matriz Int
creaCruzada n =
    array ((1,1),(n,n))
          [((i,j),valores n i j) | i <- [1..n], j <- [1..n]]
    where valores n i j | i == j     = i
                        | i+j == n+1 = i
                        | otherwise  = 0

-- ---------------------------------------------------------------------
-- Ejercicio 4. Consideramos el TAD de los polinomios y los siguientes
-- ejemplos de polinomios
--    p1  =  4*x^4 + 6*x^3 + 7*x^2 + 5*x + 2
```

```
--    p2  =  6*x^5 + 2*x^4 + 8*x^3 + 5*x^2 + 8*x + 4
-- En Haskell,
--    p1, p2 :: Polinomio Int
--    p1 = consPol 4 4
--                 (consPol 3 6
--                          (consPol 2 7
--                                   (consPol 1 5 (consPol 0 2 polCero))))
--    p2 = consPol 5 6
--                 (consPol 4 2
--                          (consPol 3 8
--                                   (consPol 2 5
--                                            (consPol 1 8
--                                                     (consPol 0 4 polCero)))))
--
-- El cociente entero de un polinomio P(x) por un monomio ax^n es el
-- polinomio que se obtiene a partir de los términos de P(x) con un
-- grado mayor o igual que n, realizando la división entera entre sus
-- coeficientes y el coeficiente del monomio divisor y restando el valor
-- de n al de sus grados. Por ejemplo,
-- + El cociente entero de 4x^4 + 6x^3 + 7x^2 + 5x + 2 por el monomio
--   3x^2 se obtiene a partir de los términos 4x^4 + 6x^3 + 7x^2
--   realizando la división entera entre sus coeficientes y el número 3
--   y restando 2 a sus grados. De esta forma se obtiene 1x^2 + 2x + 2
-- + El cociente entero de 6x^5 + 2x^4 + 8x^3 + 5x^2 + 8x + 4 por el
--   monomio 4x^3 se obtiene a partir de los términos 6x^5 + 2x^4 + 8x^3
--   realizando la división entera entre sus coeficientes y el número 4
--   y restando 3 a sus grados. De esta forma se obtiene 1x^2 + 2
--
-- Definir la función
--    cocienteEntero :: Polinomio Int -> Int -> Int -> Polinomio Int
-- tal que (cocienteEntero p a n) es el cociente entero del polinomio p
-- por el monomio de grado n y coeficiente a. Por ejemplo,
--    cocienteEntero p1 3 2  =>  x^2 + 2*x + 2
--    cocienteEntero p2 4 3  =>  x^2 + 2
-- Nota: Este ejercicio debe realizarse usando únicamente las funciones
-- de la signatura del tipo abstracto de dato Polinomio.
-- ---------------------------------------------------------------------

p1, p2 :: Polinomio Int
p1 = consPol 4 4
```

```
             (consPol 3 6
                      (consPol 2 7
                               (consPol 1 5 (consPol 0 2 polCero))))
p2 = consPol 5 6
             (consPol 4 2
                      (consPol 3 8
                               (consPol 2 5
                                        (consPol 1 8
                                                 (consPol 0 4 polCero)))))

cocienteEntero :: Polinomio Int -> Int -> Int -> Polinomio Int
cocienteEntero p a n
    | grado p < n = polCero
    | otherwise   = consPol (grado p - n) (coefLider p `div` a)
                            (cocienteEntero (restoPol p) a n)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Decimos que un número es de suma prima si la suma de
-- todos sus dígitos es un número primo. Por ejemplo el número 562 es de
-- suma prima pues la suma de sus dígitos es el número primo 13; sin
-- embargo, el número 514 no es de suma prima pues la suma de sus
-- dígitos es 10, que no es primo.
--
-- Decimos que un número es de suma prima hereditario por la derecha si
-- es de suma prima y los números que se obtienen eliminando sus últimas
-- cifras también son de suma prima. Por ejemplo 7426 es de suma prima
-- hereditario por la derecha pues 7426, 742, 74 y 7 son todos números
-- de suma prima.
--
-- Definir la constante (función sin argumentos)
--    listaSumaPrimaHD :: [Integer]
-- cuyo valor es la lista infinita de los números de suma prima
-- hereditarios por la derecha. Por ejemplo,
--    take 10 listaSumaPrimaHD  ==  [2,3,5,7,20,21,23,25,29,30]
-- ---------------------------------------------------------------------------

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 325  ==  [3,2,5]

-- 1ª definición de digitos
```

```
digitos1 :: Integer -> [Integer]
digitos1 n = map (read . (:[])) (show n)

-- 2ª definición de digitos
digitos2 :: Integer -> [Integer]
digitos2 = map (read . (:[])) . show

-- 3ª definición de digitos
digitos3 :: Integer -> [Integer]
digitos3 n = [read [d] | d <- show n]

-- 4ª definición de digitos
digitos4 :: Integer -> [Integer]
digitos4 n = reverse (aux n)
    where aux n | n < 10    = [n]
                | otherwise = rem n 10 : aux (div n 10)

-- 5ª definición de digitos
digitos5 :: Integer -> [Integer]
digitos5 = map (fromIntegral . digitToInt) . show

-- Se usará la 1ª definición
digitos :: Integer -> [Integer]
digitos = digitos1

-- (sumaPrima n) se verifica si n es un número de suma prima. Por
-- ejemplo,
--    sumaPrima 562  ==  True
--    sumaPrima 514  ==  False

-- 1ª definición de sumaPrima
sumaPrima :: Integer -> Bool
sumaPrima n = isPrime (sum (digitos n))

-- 2ª definición de sumaPrima
sumaPrima2 :: Integer -> Bool
sumaPrima2 = isPrime . sum . digitos

-- (sumaPrimaHD n) se verifica si n es de suma prima hereditario por la
-- derecha. Por ejemplo,
```

```
--    sumaPrimaHD 7426  ==  True
--    sumaPrimaHD 7427  ==  False
sumaPrimaHD n
    | n < 10    = isPrime n
    | otherwise = sumaPrima n && sumaPrimaHD (n `div` 10)

-- 1ª definición de listaSumaPrimaHD
listaSumaPrimaHD1 :: [Integer]
listaSumaPrimaHD1 = filter sumaPrimaHD [1..]

-- 2ª definición de listaSumaPrimaHD
-- =================================

listaSumaPrimaHD2 :: [Integer]
listaSumaPrimaHD2 = map fst paresSumaPrimaHDDigitos

paresSumaPrimaHDDigitos :: [(Integer, Integer)]
paresSumaPrimaHDDigitos =
    paresSumaPrimaHDDigitosAux 1 [(2,2),(3,3),(5,5),(7,7)]

paresSumaPrimaHDDigitosAux :: Integer -> [(Integer,Integer)] ->
                              [(Integer,Integer)]
paresSumaPrimaHDDigitosAux n ac =
    ac ++ paresSumaPrimaHDDigitosAux (n+1)
                                     (concatMap extiendeSumaPrimaHD ac)

extiendeSumaPrimaHD :: (Integer,Integer) -> [(Integer,Integer)]
extiendeSumaPrimaHD (n,s) = [(n*10+k,s+k) | k <- [0..9], isPrime (s+k)]

-- 3ª definición de listaSumaPrimaHD
-- =================================

listaSumaPrimaHD3 :: [Integer]
listaSumaPrimaHD3 =
    map fst (concat (iterate (concatMap extiendeSumaPrimaHD3)
                             [(2,2),(3,3),(5,5),(7,7)]))

extiendeSumaPrimaHD3 :: (Integer,Integer) -> [(Integer,Integer)]
extiendeSumaPrimaHD3 (n,s) = [(n*10+k,s+k) | k <- extensiones ! s]
```

```
extensiones :: Array Integer [Integer]
extensiones = array (1,1000)
              [(n,[k | k <- [0..9], isPrime (n+k)]) | n <- [1..1000]]

-- Comparación de eficiencia
--    ghci> listaSumaPrimaHD1 !! 600
--    34004
--    (2.47 secs, 1565301720 bytes)
--    ghci> listaSumaPrimaHD2 !! 600
--    34004
--    (0.02 secs, 7209000 bytes)
--    ghci> listaSumaPrimaHD3 !! 600
--    34004
--    (0.01 secs, 1579920 bytes)
--
--    ghci> listaSumaPrimaHD2 !! 2000000
--    3800024668046
--    (45.41 secs, 29056613824 bytes)
--    ghci> listaSumaPrimaHD3 !! 2000000
--    3800024668046
--    (4.29 secs, 973265400 bytes)
```

6.1.5. Examen 5 (5 de mayo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (4 de mayo de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.Array
import Data.Char
import Data.List
import Data.Numbers.Primes
import I1M.Grafo
import Test.QuickCheck
import qualified Data.Matrix as M
import qualified Data.Set as S
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Dado dos números n y m, decimos que m es un múltiplo
-- especial de n si m es un múltiplo de n y m no tiene ningún factor
-- primo que sea congruente con 1 módulo 3.
--
-- Definir la función
--    multiplosEspecialesCota :: Int -> Int -> [Int]
-- tal que (multiplosEspecialesCota n l) es la lista ordenada de todos los
-- múltiplos especiales de n que son menores o iguales que l. Por ejemplo,
--    multiplosEspecialesCota 5 50  ==  [5,10,15,20,25,30,40,45,50]
--    multiplosEspecialesCota 7 50  ==  []
-- ---------------------------------------------------------------------

multiplosEspecialesCota :: Int -> Int -> [Int]
multiplosEspecialesCota n l =
    [m | m <- [k*n | k <- [1..l `div` n]],
         all (\p -> p `mod` 3 /= 1) (primeFactors m)]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Dado un grafo no dirigido G, un camino en G es una
-- secuencia de nodos [v(1),v(2),v(3),...,v(n)] tal que para todo i
-- entre 1 y n-1, (v(i),v(i+1)) es una arista de G. Por ejemplo, dados
-- los grafos
--    g1 = creaGrafo ND (1,3) [(1,2,0),(1,3,0),(2,3,0)]
--    g2 = creaGrafo ND (1,4) [(1,2,0),(1,3,0),(1,4,0),(2,4,0),(3,4,0)]
-- la lista [1,2,3] es un camino en g1, pero no es un camino en g2
-- puesto que la arista (2,3) no existe en g2.
--
-- Definir la función
--    camino :: (Ix a, Num t) => (Grafo a t) -> [a] -> Bool
-- tal que (camino g vs) se verifica si la lista de nodos vs es un camino
-- en el grafo g. Por ejemplo,
--    camino g1 [1,2,3]  ==  True
--    camino g2 [1,2,3]  ==  False
-- ---------------------------------------------------------------------

g1 = creaGrafo ND (1,3) [(1,2,0),(1,3,0),(2,3,0)]
g2 = creaGrafo ND (1,4) [(1,2,0),(1,3,0),(1,4,0),(2,4,0),(3,4,0)]

camino :: (Ix a, Num t) => (Grafo a t) -> [a] -> Bool
```

```
camino g vs = all (aristaEn g) (zip vs (tail vs))

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una relación binaria homogénea R sobre un conjunto A se
-- puede representar mediante un par (xs,ps) donde xs es el conjunto de
-- los elementos de A (el universo de R) y ps es el conjunto de pares de
-- R (el grafo de R). El tipo de las relaciones binarias se define por
--    type Rel a = (S.Set a,S.Set (a,a))
-- Algunos ejemplos de relaciones binarias homogéneas son
--    r1 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(1,3),(4,3)])
--    r2 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(3,3),(4,3)])
--    r3 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(1,4),(4,3)])
--    r4 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(3,4),(4,3)])
--
-- Una relación binaria homogénea R = (U,G) es inyectiva si para todo x
-- en U hay un único y en U tal que (x,y) está en G. Por ejemplo, las
-- relaciones r2 y r4 son inyectivas, pero las relaciones r1 y r3 no.
--
-- Una relación binaria homogénea R = (U,G) es sobreyectiva si para todo
-- y en U hay algún x en U tal que (x,y) está en G. Por ejemplo, las
-- relaciones r3 y r4 son sobreyectivas, pero las relaciones r1 y r2
-- no.
--
-- Una relación binaria homogénea R = (U,G) es biyectiva si es inyectiva y
-- sobreyectiva. Por ejemplo, la relación r4 es biyectiva, pero las
-- relaciones r1, r2 y r3 no.
--
-- Definir la función
--    biyectiva :: (Ord a, Eq a) => Rel a -> Bool
-- tal que (biyectiva r) si verifica si la relación r es biyectiva. Por
-- ejemplo,
--    biyectiva r1  ==  False
--    biyectiva r2  ==  False
--    biyectiva r3  ==  False
--    biyectiva r4  ==  True
-- ---------------------------------------------------------------------

type Rel a = (S.Set a,S.Set (a,a))

r1 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(1,3),(4,3)])
```

```
r2 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(3,3),(4,3)])
r3 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(1,4),(4,3)])
r4 = (S.fromList [1..4], S.fromList [(1,2),(2,1),(3,4),(4,3)])

biyectiva :: (Ord a, Eq a) => Rel a -> Bool
biyectiva (u,g) = S.map fst g == u && S.map snd g == u

-- ---------------------------------------------------------------------
-- Ejercicio 4. La visibilidad de una lista es el número de elementos
-- que son estrictamente mayores que todos los anteriores. Por ejemplo,
-- la visibilidad de la lista [1,2,5,2,3,6] es 4.
--
-- La visibilidad de una matriz P es el par formado por las
-- visibilidades de las filas de P y las visibilidades de las
-- columnas de P. Por ejemplo, dada la matriz
--        ( 4 2 1 )
--    Q = ( 3 2 5 )
--        ( 6 1 8 )
-- la visibilidad de Q es ([1,2,2],[2,1,3]).
--
-- Definir las funciones
--    visibilidadLista  :: [Int] -> Int
--    visibilidadMatriz :: M.Matrix Int -> ([Int],[Int])
-- tales que
-- + (visibilidadLista xs) es la visibilidad de la lista xs. Por
--   ejemplo,
--      visibilidadLista [1,2,5,2,3,6]   ==  4
--      visibilidadLista [0,-2,5,1,6,6]  ==  3
-- + (visibilidadMatriz p) es la visibilidad de la matriz p. Por ejemplo,
--      ghci> visibilidadMatriz (M.fromLists [[4,2,1],[3,2,5],[6,1,8]])
--      ([1,2,2],[2,1,3])
--      ghci> visibilidadMatriz (M.fromLists [[0,2,1],[0,2,5],[6,1,8]])
--      ([2,3,2],[2,1,3])
-- ---------------------------------------------------------------------

visibilidadLista :: [Int] -> Int
visibilidadLista xs =
    length [x | (ys,x) <- zip (inits xs) xs, all (<x) ys]

visibilidadMatriz :: M.Matrix Int -> ([Int],[Int])
```

```
visibilidadMatriz p =
    ([visibilidadLista [p M.! (i,j) | j <- [1..n]] | i <- [1..m]],
     [visibilidadLista [p M.! (i,j) | i <- [1..m]] | j <- [1..n]])
     where m = M.nrows p
           n = M.ncols p

-- ---------------------------------------------------------------------
-- Ejercicio 5. Decimos que un número es alternado si no tiene dos
-- cifras consecutivas iguales ni tres cifras consecutivas en orden
-- creciente no estricto o decreciente no estricto. Por ejemplo, los
-- números 132425 y 92745 son alternados, pero los números 12325 y 29778
-- no. Las tres primeras cifras de 12325 están en orden creciente y
-- 29778 tiene dos cifras iguales consecutivas.
--
-- Definir la constante
--    alternados :: [Integer]
-- cuyo valor es la lista infinita de los números alternados. Por ejemplo,
--    take 10 alternados                      ==  [0,1,2,3,4,5,6,7,8,9]
--    length (takeWhile (< 1000) alternados)  ==  616
--    alternados !! 1234567                   ==  19390804
-- ----------------------------------------------------------------------------

-- 1ª definición
-- =============

-- (cifras n) es la lista de las cifras de n. Por ejemplo.
--    cifras 325  ==  [3,2,5]
cifras :: Integer -> [Int]
cifras n = map digitToInt (show n)

-- (cifrasAlternadas xs) se verifica si las lista de cifras xs es
-- alternada. Por ejemplo,
--    cifrasAlternadas [1,3,2,4,2,5]  ==  True
--    cifrasAlternadas [9,2,7,4,5]    ==  True
--    cifrasAlternadas [1,2,3,2,5]    ==  False
--    cifrasAlternadas [2,9,7,7,8]    ==  False
cifrasAlternadas :: [Int] -> Bool
cifrasAlternadas [x1,x2] = x1 /= x2
cifrasAlternadas (x1:x2:x3:xs) =
    not (((x1 <= x2) && (x2 <= x3)) || ((x1 >= x2) && (x2 >= x3))) &&
```

```
    cifrasAlternadas (x2:x3:xs)
cifrasAlternadas _ = True

-- (alternado n) se verifica si n es un número alternado. Por ejemplo,
--    alternado 132425  ==  True
--    alternado 92745   ==  True
--    alternado 12325   ==  False
--    alternado 29778   ==  False
alternado :: Integer -> Bool
alternado n = cifrasAlternadas (cifras n)

alternados1 :: [Integer]
alternados1 = filter alternado [0..]

-- 2ª definición
-- =============

-- (extiendeAlternado n) es la lista de números alternados que se pueden
-- obtener añadiendo una cifra al final del número alternado n. Por
-- ejemplo,
--    extiendeAlternado 7   ==  [70,71,72,73,74,75,76,78,79]
--    extiendeAlternado 24  ==  [240,241,242,243]
--    extiendeAlternado 42  ==  [423,424,425,426,427,428,429]
extiendeAlternado :: Integer -> [Integer]
extiendeAlternado n
    | n < 10    = [n*10+h | h <- [0..n-1]++[n+1..9]]
    | d < c     = [n*10+h | h <- [0..c-1]]
    | otherwise = [n*10+h | h <- [c+1..9]]
    where c = n `mod` 10
          d = (n `mod` 100) `div` 10

alternados2 :: [Integer]
alternados2 = concat (iterate (concatMap extiendeAlternado) [0])
```

6.1.6. Examen 6 (15 de junio de 2015)

El examen es común con el del grupo 4 (ver página 546).

6.1.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 5 (ver página 582).

6.1.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 5 (ver página 589).

6.1.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 5 (ver página 600).

6.2. Exámenes del grupo 2 (Antonia M. Chávez)

6.2.1. Examen 1 (6 de Noviembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (6 de noviembre de 2014)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir por comprensión, recursión y con funciones de
-- orden superior, la función
--    escalonada :: (Num a, Ord a) => [a] -> Bool
-- tal que (escalonada xs) se verifica si la diferencia entre números
-- consecutivos de xs  es siempre, en valor absoluto, mayor que 2. Por
-- ejemplo,
--    escalonada [1,5,8,23,5] == True
--    escalonada [3,6,8,1]    == False
--    escalonada [-5,-2,4 ]   == True
--    escalonada [5,2]        == True
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
escalonadaC :: (Num a, Ord a) => [a] -> Bool
escalonadaC xs = and [abs (x-y) > 2 | (x,y) <- zip xs (tail xs)]

-- 2ª definición (por recursión):
escalonadaR :: (Num a, Ord a) => [a] -> Bool
escalonadaR (x:y:zs) = abs (x-y) > 2 && escalonadaR (y:zs)
escalonadaR _        = True

-- 3ª definición (con funciones de orden superior):
escalonadaO :: (Num a, Ord a) => [a] -> Bool
escalonadaO xs = all ((>2) . abs) (zipWith (-) xs (tail xs))
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    elementos :: Ord a => [a] -> [a] -> Int
-- tal que (elementos xs ys) es el número de elementos de xs son menores
-- que los correpondientes de ys hasta el primero que no lo sea. Por
-- ejemplo,
--    elementos "prueba" "suspenso"        ==  2
--    elementos [1,2,3,4,5] [2,3,4,3,8,9]  ==  3
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
elementosR :: Ord a => [a] -> [a] -> Int
elementosR (x:xs) (y:ys) | x < y     = 1 + elementosR xs ys
                         | otherwise = 0
elementosR _ _ = 0

-- 2ª definición (con funciones de orden superior):
elementosO :: Ord a => [a] -> [a] -> Int
elementosO xs ys = length (takeWhile menor (zip xs ys))
    where menor (x,y) = x < y

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir por comprensión  y recursión la función
--    sumaPosParR :: Int -> Int
-- tal que (sumaPosParR x) es la suma de los dígitos de x que ocupan
-- posición par. Por ejemplo,
--    sumaPosPar 987651 = 8+6+1 = 15
--    sumaPosPar 98765  = 8+6   = 14
--    sumaPosPar 9876   = 8+6   = 14
--    sumaPosPar 987    = 8     =  8
--    sumaPosPar 9              =  0
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
sumaPosParR :: Int -> Int
sumaPosParR x
    | even (length (digitos x )) = aux x
    | otherwise                  = aux (div x 10)
    where aux x | x < 10    = 0
```

```
               | otherwise = mod x 10 + sumaPosParR (div x 10)

digitos :: Int -> [Int]
digitos x = [read [y] | y <- show x]

-- 2ª definición (por comprensión):
sumaPosParC :: Int -> Int
sumaPosParC x = sum [y | (y,z) <- zip (digitos x) [1 ..], even z]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Define la función
--    sinCentrales :: [a] -> [a]
-- tal que (sinCentrales xs) es la lista obtenida eliminando el elemento
-- central de xs si xs es de longitud impar y sus dos elementos
-- centrales si es de longitud par. Por ejemplo,
--    sinCentrales [1,2,3,4] ==  [1,4]
--    sinCentrales [1,2,3]   ==  [1,3]
--    sinCentrales [6,9]     ==  []
--    sinCentrales [7]       ==  []
--    sinCentrales []        ==  []
-- --------------------------------------------------------------------

sinCentrales :: [a] -> [a]
sinCentrales [] = []
sinCentrales xs | even n    = init ys ++ zs
                | otherwise = ys ++ zs
      where n         = length xs
            (ys,z:zs) = splitAt (n `div` 2) xs
```

6.2.2. Examen 2 (4 de Diciembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (4 de diciembre de 2014)
-- --------------------------------------------------------------------

-- --------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    vuelta :: [Int] -> [a] -> [a]
-- tal que (vuelta ns xs) es la lista que resulta de repetir cada
-- elemento de xs tantas veces como indican los elementos de ns
-- respectivamente. Por ejemplo,
```

```
--    vuelta [1,2,3,2,1] "ab"       == "abbaaabba"
--    vuelta [2,3,1,5] [6,5,7]      == [6,6,5,5,5,7,6,6,6,6,6]
--    take 13 (vuelta [1 ..] [6,7]) == [6,7,7,6,6,6,7,7,7,7,6,6,6]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
vuelta :: [Int] -> [a] -> [a]
vuelta (n:ns) (x:xs) = replicate n x ++ vuelta ns (xs++[x])
vuelta [] _ = []

-- 2ª definición (por comprensión):
vuelta2 :: [Int] -> [a] -> [a]
vuelta2 ns xs = concat [replicate n x | (n,x) <- zip ns (rep xs)]

-- (rep xs) es la lista obtenida repitiendo los elementos de xs. Por
-- ejemplo,
--    take 20 (rep "abbccc")  ==  "abbcccabbcccabbcccab"
rep xs = xs ++ rep xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir (por comprensión, recursión y plegado por la
-- derecha,  acumulador y plegado por la izquierda) la funcion
--    posit :: ([Int] -> Int) -> [[Int]] ->  [[Int]]
-- tal que (posit f xss) es la lista formada por las listas de xss tales
-- que, al evaluar f sobre ellas, devuelve un valor positivo. Por
-- ejemplo,
--    posit head [[1,2],[0,-4],[2,-3]]   ==  [[1,2],[2,-3]]
--    posit sum  [[1,2],[9,-4],[-8,3]]   ==  [[1,2],[9,-4]]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
positC :: ([Int] -> Int) -> [[Int]] ->  [[Int]]
positC f xss = [xs | xs <- xss, f xs > 0]

-- 2ª definición (por recursión):
positR :: ([Int] -> Int) -> [[Int]] ->  [[Int]]
positR f [] = []
positR f (xs:xss) | f xs > 0  = xs : positR f xss
                  | otherwise = positR f xss
```

```
-- 3ª definición (por plegado por la derecha):
positP :: ([Int] -> Int) -> [[Int]] ->  [[Int]]
positP f = foldr g []
    where g xs yss | f xs > 0  = xs : yss
                   | otherwise = yss

-- 4ª definición (con acumulador):
positAC :: ([Int] -> Int) -> [[Int]] ->  [[Int]]
positAC f xss = reverse (aux f xss [])
    where aux f [] yss = yss
          aux f (xs:xss) yss | f xs > 0  = aux f xss (xs:yss)
                             | otherwise = aux f xss yss

-- 5ª definición (por plegado por la izquierda):
positPL :: ([Int] -> Int) -> [[Int]] ->  [[Int]]
positPL f xss = reverse (foldl g [] xss)
    where g yss xs | f xs > 0  = xs : yss
                   | otherwise = yss

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, usando la función posit,
--    p1 :: [[Int]]
-- tal que p1 es la lista de listas de [[1,2,-3],[4,-5,-1],[4,1,2,-5,-6]]
-- que cumplen que la suma de los elementos que ocupan posiciones pares
-- es negativa o cero.
-- ---------------------------------------------------------------------

p1 :: [[Int]]
p1 = [xs | xs <- l, xs `notElem` positP f l]
    where l = [[1,2,-3],[4,-5,-1],[4,1,2,-5,-6]]
          f []       = 0
          f [x]      = x
          f (x:_:xs) = x + f xs

-- El cálculo es
--    ghci> p1
--    [[1,2,-3],[4,1,2,-5,-6]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
```

```
--    maxCumplen :: (a -> Bool) -> [[a]] -> [a]
-- tal que (maxCumplen p xss) es la lista de xss que tiene más elementos
-- que cumplen el predicado p. Por ejemplo,
--    maxCumplen even [[3,2],[6,8,7],[5,9]]  ==  [6,8,7]
--    maxCumplen odd  [[3,2],[6,8,7],[5,9]]  ==  [5,9]
--    maxCumplen (<5) [[3,2],[6,8,7],[5,9]]  ==  [3,2]
-- ---------------------------------------------------------------------

maxCumplen :: (a -> Bool) -> [[a]] -> [a]
maxCumplen p xss = head [xs | xs <- xss, f xs == m]
    where m    = maximum [f xs | xs <- xss]
          f xs = length (filter p xs)
```

6.2.3. Examen 3 (23 de enero de 2015)

El examen es común con el del grupo 4 (ver página 524).

6.2.4. Examen 4 (12 de marzo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (12 de marzo de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Librerías auxiliares
-- ---------------------------------------------------------------------

import Data.Array
import Data.List
import I1M.PolOperaciones
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Dado un polinomio p con coeficientes enteros, se
-- llama parejo si está formado exclusivamente por monomios de grado par.
--
-- Definir la funcion
--    parejo :: Polinomio Int -> Bool
-- tal que (parejo p) se verifica si el polinomio p es parejo. Por
-- ejemplo,
```

```
--    ghci> let p1 = consPol 3 2 (consPol 4 1 polCero)
--    ghci> parejo p1
--    False
--    ghci> let p2 = consPol 6 3 (consPol 4 1 (consPol 0 5  polCero))
--    ghci> parejo p2
--    True
-- ---------------------------------------------------------------------

parejo :: Polinomio Int -> Bool
parejo p = all even (grados p)

grados p | esPolCero p = [0]
         | otherwise   = grado p : grados (restoPol p)

-- ---------------------------------------------------------------------
-- Ejercicio 2 . Las matrices pueden representarse mediante tablas cuyos
-- indices son pares de numeros naturales:
--    type Matriz a = Array (Int,Int) a
--
-- Definir la funcion
--    mayorElem :: Matriz -> Matriz
-- tal que (mayorElem p) es la matriz obtenida añadiéndole al principio
-- una columna con el mayor elemento de cada fila. Por ejemplo,
-- aplicando mayorElem a las matrices
--    |1 8 3|      |1 2|
--    |4 5 6|      |7 4|
--                 |5 6|
-- se obtienen, respectivamente
--    |8 1 8 3|   |2 1 2|
--    |6 4 5 6|   |7 7 4|
--                |6 5 6|
-- En Haskell,
--    ghci> mayorElem (listArray ((1,1),(2,3)) [1,8,3, 4,5,6])
--          array ((1,1),(2,4))
--          [((1,1),8),((1,2),1),((1,3),8),((1,4),3),
--           ((2,1),6),((2,2),4),((2,3),5),((2,4),6)]
--    ghci> mayorElem (listArray ((1,1),(3,2)) [1,2, 7,4, 5,6])
--          array ((1,1),(3,3))
--          [((1,1),2),((1,2),1),((1,3),2),
--           ((2,1),7),((2,2),7),((2,3),4),
```

```
--           ((3,1),6),((3,2),5),((3,3),6)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

mayorElem :: Matriz  Int -> Matriz Int
mayorElem p = listArray ((1,1),(m,n+1))
                        [f i j | i <- [1..m], j <- [1..n+1]]
    where
      m = fst(snd(bounds p))
      n = snd(snd(bounds p))
      f i j | j > 1   = p ! (i,j-1)
            | j==1    =  maximum [p!(i,j)|j<- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la sucesion (infinita)
--    numerosConDigitosPrimos :: [Int]
-- tal que sus elementos son los números enteros positivos con todos sus
-- dígitos primos. Por ejemplo,
--    ghci> take 22 numerosConDigitosPrimos
--    [2,3,5,7,22,23,25,27,32,33,35,37,52,53,55,57,72,73,75,77,222,223]
-- ---------------------------------------------------------------------

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

-- (digitosPrimos n) se verifica si todos los digitos de n son
-- primos. Por ejemplo,
--    digitosPrimos 352  ==  True
--    digitosPrimos 362  ==  False
digitosPrimos :: Int -> Bool
digitosPrimos n = all (`elem` "2357") (show n)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    alterna ::  Int -> Int -> Matriz Int
-- tal que (alterna n x) es la matriz de dimensiones nxn que contiene el
-- valor x alternado con 0 en todas sus posiciones. Por ejemplo,
--    ghci> alterna 4 2
```

```
--    array ((1,1),(4,4)) [((1,1),2),((1,2),0),((1,3),2),((1,4),0),
--                         ((2,1),0),((2,2),2),((2,3),0),((2,4),2),
--                         ((3,1),2),((3,2),0),((3,3),2),((3,4),0),
--                         ((4,1),0),((4,2),2),((4,3),0),((4,4),2)]
--    ghci>alterna 3 1
--    array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),1),
--                         ((2,1),0),((2,2),1),((2,3),0),
--                         ((3,1),1),((3,2),0),((3,3),1)]
-- ---------------------------------------------------------------------

alterna ::  Int -> Int -> Matriz Int
alterna n x =
    array ((1,1),(n,n)) [((i,j),f i j) | i <- [1..n], j <- [1..n]]
    where f i j | even (i+j) = x
                | otherwise  = 0

-- ---------------------------------------------------------------------
-- Ejercicio 5. Los árboles binarios con datos en nodos y hojas se
-- define por
--    data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
-- Por ejemplo, el árbol
--           3
--          / \
--         /   \
--        4     7
--       / \   / \
--      5   0 0   3
--     / \
--    2   0
-- se representa por
--    ejArbol :: Arbol Integer
--    ejArbol = N 3 (N 4 (N 5 (H 2)(H 0)) (H 0)) (N 7 (H 0) (H 3))
--
-- Definir la función
--    caminos :: Eq a => a -> Arbol a -> [[a]]
-- tal que (caminos x ar) es la lista de caminos en el arbol ar hasta
-- llegar a x. Por ejemplo
--    caminos 0 ejArbol  ==  [[3,4,5,0],[3,4,0],[3,7,0]]
--    caminos 3 ejArbol  ==  [[3],[3,7,3]]
--    caminos 1 ejArbol  ==  []
```

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

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

ejArbol :: Arbol Integer
ejArbol = N 3 (N 4 (N 5 (H 2)(H 0)) (H 0)) (N 7 (H 0) (H 3))

caminos :: Eq a => a -> Arbol a -> [[a]]
caminos x (H y) | x == y    = [[y]]
                | otherwise = []
caminos x (N y i d)
    | x == y    = [y] : [y:xs | xs <- caminos x i ++ caminos x d ]
    | otherwise = [y:xs | xs <- caminos x i ++ caminos x d]
```

6.2.5. Examen 5 (7 de mayo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- Informática: 5º examen de evaluación continua (7 de mayo de 2015)
-- ---------------------------------------------------------------------

import Data.Array
import Data.List
import Data.Numbers.Primes
import I1M.Grafo
import I1M.Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número tiene una inversión cuando existe un dígito
-- x a la derecha de otro dígito de forma que x es menor que y. Por
-- ejemplo, en el número 1745 hay dos inversiones ya que 4 es menor
-- que 7 y 5 es menor que 7 y están a la derecha de 7.
--
-- Definir la función
--    nInversiones :: Integer -> Int
-- tal que (nInversiones n) es el número de inversiones de n. Por
-- ejemplo,
--    nInversiones 1745  ==  2
-- ---------------------------------------------------------------------

-- 1ª definición
```

```
-- =============

nInversiones1 :: Integer -> Int
nInversiones1 = length . inversiones . show

-- (inversiones xs) es la lista de las inversiones de xs. Por ejemplo,
--    inversiones "1745"   ==  [('7','4'),('7','5')]
--    inversiones "cbafd"  ==  [('c','b'),('c','a'),('b','a'),('f','d')]
inversiones :: Ord a => [a] -> [(a,a)]
inversiones []     = []
inversiones (x:xs) = [(x,y) | y <- xs, y < x] ++ inversiones xs

-- 2ª definición
-- =============

nInversiones2 :: Integer -> Int
nInversiones2 = aux . show
    where aux [] = 0
          aux (y:ys) | null xs   = aux ys
                     | otherwise = length xs + aux ys
                     where xs = [x | x <- ys, x < y]

-- 3ª solución
-- ===========

nInversiones3 :: Integer -> Int
nInversiones3 x = sum $ map f xss
    where xss = init $ tails (show x)
          f (x:xs) = length $ filter (<x) xs

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

-- La comparación es
--    ghci> let f1000 = product [1..1000]
--    ghci> nInversiones1 f1000
--    1751225
--    (2.81 secs, 452526504 bytes)
--    ghci> nInversiones2 f1000
--    1751225
```

```haskell
--    (2.45 secs, 312752672 bytes)
--    ghci> nInversiones3 f1000
--    1751225
--    (0.71 secs, 100315896 bytes)

-- En lo sucesivo, se usa la 3ª definición
nInversiones :: Integer -> Int
nInversiones = nInversiones3

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcular cuántos números hay de 4 cifras con más de
-- dos inversiones.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> length [x |x <- [1000 ..9999], nInversiones x > 2]
--    5370

-- ---------------------------------------------------------------------
-- Ejercicio 2. La notas de un examen se pueden representar mediante un
-- vector en el que los valores son los pares formados por los nombres
-- de los alumnos y sus notas.
--
-- Definir la función
--    aprobados :: (Num a, Ord a) => Array Int (String,a) -> Maybe [String]
-- tal que (aprobados p) es la lista de los nombres de los alumnos que
-- han aprobado y Nothing si todos están suspensos. Por ejemplo,
--    ghci> aprobados (listArray (1,3) [("Ana",5),("Pedro",3),("Lucia",6)])
--    Just ["Ana","Lucia"]
--    ghci> aprobados (listArray (1,3) [("Ana",4),("Pedro",3),("Lucia",4.9)])
--    Nothing
-- ---------------------------------------------------------------------

aprobados :: (Num a, Ord a) => Array Int (String,a) -> Maybe [String]
aprobados p | null xs   = Nothing
            | otherwise = Just xs
    where xs = [x | i <- indices p
                  , let (x,n) = p!i
                  , n >= 5]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    refina :: Ord a => Monticulo a -> [a -> Bool] -> Monticulo a
-- tal que (refina m ps) es el formado por los elementos del montículo m
-- que cumplen todos predicados de la lista ps. Por ejemplo,
--    ghci> refina (foldr inserta vacio [1..22]) [(<7), even]
--    M 2 1 (M 4 1 (M 6 1 Vacio Vacio) Vacio) Vacio
--    ghci> refina (foldr inserta vacio [1..22]) [(<1), even]
--    Vacio
-- ---------------------------------------------------------------------

refina :: Ord a => Monticulo a -> [a-> Bool] -> Monticulo a
refina m ps | esVacio m   = vacio
            | cumple x ps = inserta x (refina r ps)
            | otherwise   = refina r ps
            where x = menor m
                  r = resto m

-- (cumple x ps) se verifica si x cumple todos los predicados de ps. Por
-- ejemplo,
--    cumple 2 [(<7), even]  ==  True
--    cumple 3 [(<7), even]  ==  False
--    cumple 8 [(<7), even]  ==  False
cumple :: a -> [a -> Bool] -> Bool
cumple x ps = and [p x | p <- ps]

-- La función ‘cumple‘se puede definir por recursión:
cumple2 x []     = True
cumple2 x (p:ps) = p x && cumple x ps

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    diferencia :: Ord a => Monticulo a -> Monticulo a -> Monticulo a
-- tal que (diferencia m1 m2) es el montículo formado por los elementos
-- de m1 que no están en m2. Por ejemplo,
--    ghci> diferencia (foldr inserta vacio [7,5,6]) (foldr inserta vacio [4,5])
--    M 6 1 (M 7 1 Vacio Vacio) Vacio
-- ---------------------------------------------------------------------

diferencia :: Ord a => Monticulo a -> Monticulo a -> Monticulo a
```

```
diferencia m1 m2
    | esVacio m1          = vacio
    | esVacio m2          = m1
    | menor m1 <  menor m2 = inserta (menor m1) (diferencia (resto m1) m2)
    | menor m1 == menor m2 = diferencia (resto m1) (resto m2)
    | menor m1 >  menor m2 = diferencia m1 (resto m2)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Una matriz latina de orden n es una matriz cuadrada de
-- orden n tal que todos sus elementos son cero salvo los de su fila y
-- columna central, si n es impar; o los de sus dos filas y columnas
-- centrales, si n es par.
--
-- Definir la función
--    latina :: Int -> Array (Int,Int) Int
-- tal que (latina n) es la siguiente matriz latina de orden n:
-- + Para n impar:
--      | 0  0... 0 1   0 ... 0   0|
--      | 0  0... 0 2   0 ... 0   0|
--      | 0  0... 0 3   0 ... 0   0|
--      | .........................|
--      | 1  2.............n-1   n|
--      | .........................|
--      | 0  0... 0 n-2 0 ... 0   0|
--      | 0  0... 0 n-1 0 ... 0   0|
--      | 0  0... 0 n   0 ... 0   0|
-- + Para n par:
--      | 0  0... 0 1   n    0 ...   0   0|
--      | 0  0... 0 2   n-1  0 ...   0   0|
--      | 0  0... 0 3   n-2  0 ...   0   0|
--      | ...............................|
--      | 1  2...................n-1   n|
--      | n n-1 .................. 2   1|
--      | ...............................|
--      | 0  0... 0 n-2  3   0 ...   0   0|
--      | 0  0... 0 n-1  2   0 ...   0   0|
--      | 0  0... 0 n    1   0 ...   0   0|
-- Por ejemplo,
--    ghci> elems (latina 5)
--    [0,0,1,0,0,
```

```
--     0,0,2,0,0,
--     1,2,3,4,5,
--     0,0,4,0,0,
--     0,0,5,0,0]
--    ghci> elems (latina 6)
--    [0,0,1,6,0,0,
--     0,0,2,5,0,0,
--     1,2,3,4,5,6,
--     6,5,4,3,2,1,
--     0,0,5,2,0,0,
--     0,0,6,1,0,0]
-- ---------------------------------------------------------------------------

latina :: Int -> Array (Int,Int) Int
latina n | even n    = latinaPar n
         | otherwise = latinaImpar n

-- (latinaImpar n) es la matriz latina de orden n, siendo n un número
-- impar. Por ejemplo,
--    ghci> elems (latinaImpar 5)
--    [0,0,1,0,0,
--     0,0,2,0,0,
--     1,2,3,4,5,
--     0,0,4,0,0,
--     0,0,5,0,0]
latinaImpar :: Int -> Array (Int,Int) Int
latinaImpar n =
    array ((1,1),(n,n)) [((i,j),f i j) | i <- [1..n], j <- [1..n]]
    where c = 1 + (n `div` 2)
          f i j | i == c    = j
                | j == c    = i
                | otherwise = 0

-- (latinaPar n) es la matriz latina de orden n, siendo n un número
-- par. Por ejemplo,
--    ghci> elems (latinaPar 6)
--    [0,0,1,6,0,0,
--     0,0,2,5,0,0,
--     1,2,3,4,5,6,
--     6,5,4,3,2,1,
```

```
--     0,0,5,2,0,0,
--     0,0,6,1,0,0]
latinaPar :: Int -> Array (Int,Int) Int
latinaPar n =
    array ((1,1),(n,n)) [((i,j),f i j) | i <- [1..n], j <- [1..n]]
    where c = n `div` 2
          f i j | i == c    = j
                | i == c+1  = n-j+1
                | j == c    = i
                | j == c+1  = n-i+1
                | otherwise = 0

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir las funciones
--    grafo   :: [(Int,Int)] -> Grafo Int Int
--    caminos :: Grafo Int Int -> Int -> Int -> [[Int]]
-- tales que
-- + (grafo as) es el grafo no dirigido definido cuyas aristas son as. Por
--   ejemplo,
--      ghci> grafo [(2,4),(4,5)]
--      G ND (array (2,5) [(2,[(4,0)]),(3,[]),(4,[(2,0),(5,0)]),(5,[(4,0)])])
-- + (caminos g a b) es la lista los caminos en el grafo g desde a hasta
--   b sin pasar dos veces por el mismo nodo. Por ejemplo,
--      ghci> caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 7
--      [[1,3,7],[1,3,5,7]]
--      ghci> caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 2 7
--      [[2,5,7],[2,5,3,7]]
--      ghci> caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 2
--      [[1,3,7,5,2],[1,3,5,2]]
--      ghci> caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 4
--      []
-- ---------------------------------------------------------------------

grafo :: [(Int,Int)] -> Grafo Int Int
grafo as = creaGrafo ND (m,n) [(x,y,0) | (x,y) <- as]
    where ns = map fst as ++ map snd as
          m  = minimum ns
          n  = maximum ns

caminos :: Grafo Int Int -> Int -> Int -> [[Int]]
```

```
caminos g a b = reverse (aux b a [])
    where aux a b vs
              | a == b    = [[b]]
              | otherwise = [b:xs | c <- adyacentes g b
                                  , c `notElem` vs
                                  , xs <- aux a c (c:vs)]

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    sumaDePrimos :: Int -> [[Int]]
-- tal que (sumaDePrimos x) es la lista de las listas no crecientes de
-- números primos que suman x. Por ejemplo:
--    sumaDePrimos 10  ==  [[7,3],[5,5],[5,3,2],[3,3,2,2],[2,2,2,2,2]]
-- ---------------------------------------------------------------------

sumaDePrimos :: Integer -> [[Integer]]
sumaDePrimos 1 = []
sumaDePrimos n = aux n (reverse (takeWhile (<=n) primes))
    where aux _ [] = []
          aux n (x:xs) | x > n     = aux n xs
                       | x == n    = [n] : aux n xs
                       | otherwise = map (x:) (aux (n-x) (x:xs)) ++ aux n xs
```

6.2.6. Examen 6 (15 de junio de 2015)

El examen es común con el del grupo 4 (ver página 546).

6.2.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 5 (ver página 582).

6.2.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 5 (ver página 589).

6.2.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 5 (ver página 600).

6.3. Exámenes del grupo 3 (Andrés Cordón)

6.3.1. Examen 1 (4 de Noviembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (4 de noviembre de 2014)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, usando listas por comprensión, la función
--      mulPosC :: Int ->  [Int] -> [Int]
-- tal que (mulPosC x xs) es la lista de los elementos de xs que son
-- múltiplos positivos de x. Por ejemplo,
--    mulPosC 3 [1,6,-5,-9,33] == [6,33]
-- ---------------------------------------------------------------------

mulPosC :: Int -> [Int] -> [Int]
mulPosC x xs = [y | y <- xs, y > 0, rem y x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    mulPosR :: Int -> [Int] -> [Int]
-- tal que (mulPosR x xs) es la lista de los elementos de xs que son
-- múltiplos positivos de x. Por ejemplo,
--    mulPosR 3 [1,6,-5,-9,33] == [6,33]
-- ---------------------------------------------------------------------

mulPosR :: Int -> [Int] -> [Int]
mulPosR _ [] = []
mulPosR x (y:ys) | y > 0 && rem y x == 0 = y : mulPosR x ys
                 | otherwise             = mulPosR x ys

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Diremos que una lista numérica es muy creciente si
-- cada elemento es mayor estricto que el doble del anterior.
--
-- Definir el predicado
--    muyCreciente :: (Ord a, Num a) => [a] -> Bool
-- tal que (muyCreciente xs) se verifica si xs es una lista muy
```

```
-- creciente. Por ejemplo,
--    muyCreciente [3,7,100,220] == True
--    muyCreciente [1,5,7,1000]  == False
-- ---------------------------------------------------------------------

muyCreciente :: (Ord a, Num a) => [a] -> Bool
muyCreciente xs = and [y > 2*x | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Para generar listas muy crecientes, consideramos la
-- función
--    f :: Integer -> Integer -> Integer
-- dada por las ecuaciones recursivas:
--    f(x,0) = x
--    f(x,n) = 2*f(x,n-1) + 1, si n > 0
--
--  Definir la función
--    lista :: Int -> Integer -> [Integer]
-- tal que (lista n x) es la lista [f(x,0),f(x,1),...,f(x,n)]
-- Por ejemplo,
--    lista 5 4 == [4,9,19,39,79]
-- ---------------------------------------------------------------------

f :: Integer -> Integer -> Integer
f x 0 = x
f x n = 2 * f x (n-1) + 1

lista :: Int -> Integer -> [Integer]
lista n x = take n [f x i | i <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. 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 es la masa puntual.
--
-- Definir la función
--    masaTotal ::  [((Float,Float),Float)] -> Float
-- tal que (masaTotal xs) es la masa total del conjunto xs. Por ejemplo,
--    masaTotal  [((-1,3),2),((0,0),5),((1,4),3)] == 10.0
-- ---------------------------------------------------------------------
```

```
masaTotal :: [((Float,Float),Float)] -> Float
masaTotal xs = sum [m | (_,m) <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Se define el diámetro de un conjunto de puntos del
-- plano como la mayor distancia entre dos puntos del conjunto.
--
-- Definir la función
--    diametro :: [[(Float,Float),Float)] -> Float
-- tal que (diametro xs) es el diámetro del conjunto de masas xs. Por
-- ejemplo,
--    diametro  [((-1,3),2),((0,0),5),((1,4),3)] == 4.1231055
-- ---------------------------------------------------------------------

diametro :: [((Float,Float),Float)] -> Float
diametro xs = maximum [dist p q | (p,_) <- xs, (q,_) <- xs]
    where dist (a,b) (c,d) = sqrt ((a-c)^2+(b-d)^2)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se define el grado de similitud entre dos cadenas de
-- texto como la menor posición en la que ambas cadenas difieren, o bien
-- como la longitud de la cadena menor si una cadena es una segmento
-- inicial de la otra.
--
-- Definir la función:
--    grado :: String -> String -> Int
-- tal que (grado xs ys) es el grado de similitud entre las cadenas xs e ys.
-- Por ejemplo,
--    grado "cadiz" "calamar" == 2
--    grado "sevilla" "betis" == 0
--    grado "pi" "pitagoras"  == 2
-- ---------------------------------------------------------------------

-- 1ª definición:
grado :: String -> String -> Int
grado xs ys | null zs   = min (length xs) (length ys)
            | otherwise = head zs
    where zs = [i | ((x,y),i) <- zip (zip xs ys) [0..], x /= y]
```

```
-- 2ª definición:
grado2 :: String -> String -> Int
grado2 xs ys = length (takeWhile (\(x,y) -> x == y) (zip xs ys))
```

6.3.2. Examen 2 (5 de Diciembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (5 de diciembre de 2014)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    cuentaC :: Ord a => a -> a -> [a] -> Int
-- tal que (cuentaC x y xs) es el número de elementos de la lista xs que
-- están en el intervalo [x,y]. Por ejemplo,
--    cuentaC 50 150 [12,3456,100,78,711] == 2
-- ---------------------------------------------------------------------

cuentaC :: Ord a => a -> a -> [a] -> Int
cuentaC x y xs = length [z | z <- xs, x <= z && z <= y]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, usando orden superior (map, filter, ...), la
-- función
--    cuentaS :: Ord a => a -> a -> [a] -> Int
-- tal que (cuentaS x y xs) es el número de elementos de la lista xs que
-- están en el intervalo [x,y]. Por ejemplo,
--    cuentaS 50 150 [12,3456,100,78,711] == 2
-- ---------------------------------------------------------------------

cuentaS ::  Ord a => a -> a -> [a] -> Int
cuentaS x y = length . filter (>=x) . filter (<=y)

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por recursión, la función
--    cuentaR :: Ord a => a -> a -> [a] -> Int
-- tal que (cuentaR x y xs) es el número de elementos de la lista xs que
-- están en el intervalo [x,y]. Por ejemplo,
--    cuentaR 50 150 [12,3456,100,78,711] == 2
```

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

cuentaR ::  Ord a => a -> a -> [a] -> Int
cuentaR _ _ [] = 0
cuentaR x y (z:zs) | x <= z && z <= y = 1 + cuentaR x y zs
                   | otherwise        = cuentaR x y zs

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir, por plegado (foldr), la función
--    cuentaP :: Ord a => a -> a -> [a] -> Int
-- tal que (cuentaP x y xs) es el número de elementos de la lista xs que
-- están en el intervalo [x,y]. Por ejemplo,
--    cuentaP 50 150 [12,3456,100,78,711] == 2
-- ---------------------------------------------------------------------

-- 1ª definición:
cuentaP ::  Ord a => a -> a -> [a] -> Int
cuentaP x y = foldr (\z u -> if x <= z && z <= y then 1 + u else u) 0

-- 2ª definición:
cuentaP2 ::  Ord a => a -> a -> [a] -> Int
cuentaP2 x y = foldr f 0
    where f z u | x <= z && z <= y = 1 + u
                | otherwise        = u

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

-- La propiedad es
prop_cuenta :: Int -> Int -> [Int] -> Bool
prop_cuenta x y zs =
    cuentaS x y zs == n &&
    cuentaR x y zs == n &&
    cuentaP x y zs == n
    where n = cuentaC x y zs

-- La comprobación es
--    ghci> quickCheck prop_cuenta
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    mdp :: Integer -> Integer
-- tal que (mdp x) es el mayor divisor primo del entero positivo x. Por
-- ejemplo,
--    mdp 100    ==  5
--    mdp 45     ==  9
--    mdp 12345  ==  823
-- ---------------------------------------------------------------------

mdp :: Integer -> Integer
mdp x = head [i | i <- [x,x-1..2], rem x i == 0, primo i]

primo :: Integer -> Bool
primo x = divisores x == [1,x]
   where divisores x = [y | y <- [1..x], rem x y == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    busca :: Integer -> Integer -> Integer
-- tal que (busca a b) es el menor entero por encima de a cuyo mayor
-- divisor primo es mayor o igual que b. Por ejemplo,
--    busca 2014  1000 == 2017
--    busca 2014 10000 == 10007
-- ---------------------------------------------------------------------

busca :: Integer -> Integer -> Integer
busca a b = head [i | i <- [max a b..], mdp i >= b]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Consideramos el predicado
--    comun :: Eq b => (a -> b) -> [a] -> Bool
-- tal que (comun f xs) se verifica si al aplicar la función f a los
-- elementos de xs obtenemos siempre el mismo valor. Por ejemplo,
--    comun (^2) [1,-1,1,-1]                 == True
--    comun (+1) [1,2,1]                     == False
--    comun length ["eva","iba","con","ana"] == True
--
```

```
-- Definir, por recursión, el predicado comun.
-- ---------------------------------------------------------------------

-- 1ª definición
comunR ::  Eq b => (a -> b) -> [a] -> Bool
comunR f (x:y:xs) = f x == f y && comunR f (y:xs)
comunR _ _        = True

-- 2ª definición:
comunR2 ::  Eq b => (a -> b) -> [a] -> Bool
comunR2 _ [] = True
comunR2 f (x:xs) = aux xs
    where z          = f x
          aux []     = True
          aux (y:ys) = f y == z && aux ys

-- Comparación de eficiencia:
--    ghci> comunR (\n -> product [1..n]) (replicate 20 20000)
--    True
--    (39.71 secs, 11731056160 bytes)
--
--    ghci> comunR2 (\n -> product [1..n]) (replicate 20 20000)
--    True
--    (20.36 secs, 6175748288 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por comprensión, el predicado comun.
-- ---------------------------------------------------------------------

-- 1ª definición
comunC ::  Eq b => (a -> b) -> [a] -> Bool
comunC f xs = and [f a == f b | (a,b) <- zip xs (tail xs)]

-- 2ª definición
comunC2 ::  Eq b => (a -> b) -> [a] -> Bool
comunC2 _ []     = True
comunC2 f (x:xs) = and [f y == z | y <- xs]
    where z = f x

-- Comparación de eficiencia:
```

```
--    ghci> comunC (\n -> product [1..n]) (replicate 20 20000)
--    True
--    (39.54 secs, 11731056768 bytes)
--
--    ghci> comunC2 (\n -> product [1..n]) (replicate 20 20000)
--    True
--    (20.54 secs, 6175747048 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    extension :: String -> (String,String)
-- tal que (extension cs) es el par (nombre,extensión) del fichero
-- cs. Por ejemplo,
--    extension "examen.hs"    ==  ("examen",hs")
--    extension "index.html"   ==  ("index","html")
--    extension "sinExt"       ==  ("sinExt","")
--    extension "raro.pdf.ps"  ==  ("raro","pdf.ps")
-- ----------------------------------------------------------------

extension :: String -> (String,String)
extension cs
    | '.' `notElem` cs = (cs,"")
    | otherwise        = (takeWhile (/='.') cs, tail (dropWhile (/='.') cs))

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Un entero positivo x es especial si en x y en x^2
-- aparecen los mismos dígitos. Por ejemplo, 10 es especial porque en 10
-- y en 10^2 = 100 aparecen los mismos dígitos (0 y 1). Asimismo,
-- 4762 es especial porque en 4762 y en 4762^2 = 22676644 aparecen los
-- mismos dígitos (2, 4, 6 y 7).
--
-- Definir el predicado
--    especial :: Integer -> Bool
-- tal que (especial x) se verifica si x es un entero positivo especial.
-- ---------------------------------------------------------------------

especial :: Integer -> Bool
especial x = digitos x == digitos (x^2)
    where digitos z = [d | d <- ['0'..'9'], d `elem` show z]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    especiales :: Int -> [Integer]
-- tal que (especiales x) es la lista de los x primeros números
-- especiales que no son potencias de 10. Por ejemplo,
--    espaciales 5 == [4762,4832,10376,10493,11205]
-- ---------------------------------------------------------------------

especiales :: Int -> [Integer]
especiales x = take x [i | i <- [1..], not (pot10 i), especial i]
    where pot10 z = z `elem` takeWhile (<= z) (map (10^) [0 ..])
```

6.3.3. Examen 3 (23 de enero de 2015)

El examen es común con el del grupo 4 (ver página 524).

6.3.4. Examen 4 (18 de marzo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (18 de marzo de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import I1M.Pol
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    agrupa :: Eq a => [a] -> [(a,Int)]
-- tal que (agrupa xs) es la lista obtenida agrupando las ocurrencias
-- consecutivas de elementos de xs junto con el número de dichas
-- ocurrencias. Por ejemplo:
--    agrupa "aaabzzaa" == [('a',3),('b',1),('z',2),('a',2)]
-- ---------------------------------------------------------------------
```

```
-- 1ª definición (por recursión)
agrupa :: Eq a => [a] -> [(a,Int)]
agrupa xs = aux xs 1
    where aux (x:y:zs) n | x == y    = aux (y:zs) (n+1)
                         | otherwise = (x,n) : aux (y:zs) 1
          aux [x]      n             = [(x,n)]

-- 2ª definición (por recursión usando takeWhile):
agrupa2 :: Eq a => [a] -> [(a,Int)]
agrupa2 [] = []
agrupa2 (x:xs) =
    (x,1 + length (takeWhile (==x) xs)) : agrupa2 (dropWhile (==x) xs)

-- 3ª definición (por comprensión usando group):
agrupa3 :: Eq a => [a] -> [(a,Int)]
agrupa3 xs = [(head ys,length ys) | ys <- group xs]

-- 4ª definición (usando map y group):
agrupa4 :: Eq a => [a] -> [(a,Int)]
agrupa4 = map (\xs -> (head xs, length xs)) . group

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función expande
--    expande :: [(a,Int)] -> [a]
-- tal que (expande xs) es la lista expandida correspondiente a ps (es
-- decir, es la lista xs tal que la comprimida de xs es ps. Por ejemplo,
--    expande [('a',2),('b',3),('a',1)] == "aabbba"
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
expande :: [(a,Int)] -> [a]
expande ps = concat [replicate k x | (x,k) <- ps]

-- 2ª definición (por concatMap)
expande2 :: [(a,Int)] -> [a]
expande2 = concatMap (\(x,k) -> replicate k x)

-- 3ª definición (por recursión)
expande3 :: [(a,Int)] -> [a]
expande3 [] = []
```

```
expande3 ((x,n):ps) = replicate n x ++ expande3 ps

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Dos enteros positivos a y b se dirán relacionados
-- si poseen, exactamente, un factor primo en común. Por ejemplo, 12 y
-- 14 están relacionados pero 6 y 30 no lo están.
--
-- Definir la lista infinita
--    paresRel :: [(Int,Int)]
-- tal que paresRel enumera todos los pares (a,b), con 1 <= a < b,
-- tal que a y b están relacionados. Por ejemplo,
--    ghci> take 10 paresRel
--    [(2,4),(2,6),(3,6),(4,6),(2,8),(4,8),(6,8),(3,9),(6,9),(2,10)]
--
-- ¿Qué lugar ocupa el par (51,111) en la lista infinita paresRel?
-- ---------------------------------------------------------------------

paresRel :: [(Int,Int)]
paresRel = [(a,b) | b <- [1..], a <- [1..b-1], relacionados a b]

relacionados :: Int -> Int -> Bool
relacionados a b =
    length (nub (primeFactors a `intersect` primeFactors b)) == 1

-- El cálculo es
--    ghci> 1 + length (takeWhile (/=(51,111)) paresRel)
--    2016

-- ---------------------------------------------------------------------
-- Ejercicio 3.  Representamos árboles binarios con elementos en las
-- hojas y en los nodos mediante el tipo de dato
--    data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
-- Por ejemplo,
--    ej1 :: Arbol Int
--    ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))
--
-- Definir la función
--    ramasCon :: Eq a => Arbol a -> a -> [[a]]
-- tal que (ramasCon a x) es la lista de las ramas del árbol a en las
-- que aparece el elemento x. Por ejemplo,
```

```
--   ramasCon ej1 2 ==  [[5,2,1],[5,2,2],[5,3,2]]
-- ---------------------------------------------------------------------

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

ej1 :: Arbol Int
ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))

-- 1ª definición
-- =============

ramasCon :: Eq a => Arbol a -> a -> [[a]]
ramasCon a x = [ys | ys <- ramas a, x `elem` ys]

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

-- 2ª definición
-- =============

ramasCon2 :: Eq a => Arbol a -> a -> [[a]]
ramasCon2 a x = filter (x `elem`) (ramas2 a)

ramas2 :: Arbol a -> [[a]]
ramas2 (H x)     = [[x]]
ramas2 (N x i d) = map (x:) (ramas2 i ++ ramas2 d)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Representamos matrices mediante el tipo de dato
--    type Matriz a = Array (Int,Int) a
-- Por ejemplo,
--    ejM :: Matriz Int
--    ejM = listArray ((1,1),(2,4)) [1,2,3,0,4,5,6,7]
-- representa la matriz
--    |1 2 3 0|
--    |4 5 6 7|
--
-- Definir la función
--    ampliada :: Num a => Matriz a -> Matriz a
```

```haskell
-- tal que (ampliada p) es la matriz obtenida al añadir una nueva fila
-- a p cuyo elemento i-ésimo es la suma de la columna i-ésima de p.
-- Por ejemplo,
--    |1 2 3 0|        |1 2 3 0|
--    |4 5 6 7| ==>    |4 5 6 7|
--                     |5 7 9 7|
-- En Haskell,
--    ghci> ampliada ejM
--    array ((1,1),(3,4)) [((1,1),1),((1,2),2),((1,3),3),((1,4),0),
--                         ((2,1),4),((2,2),5),((2,3),6),((2,4),7),
--                         ((3,1),5),((3,2),7),((3,3),9),((3,4),7)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

ejM :: Matriz Int
ejM = listArray ((1,1),(2,4)) [1,2,3,0,4,5,6,7]

ampliada :: Num a => Matriz a -> Matriz a
ampliada p =
    array ((1,1),(m+1,n)) [((i,j),f i j) | i <- [1..m+1], j <- [1..n]]
    where (_,(m,n)) = bounds p
          f i j | i <= m    = p!(i,j)
                | otherwise = sum [p!(i,j) | i <- [1..m]]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Un polinomio de coeficientes enteros se dirá par si
-- todos sus coeficientes son números pares. Por ejemplo, el polinomio
-- 2*x^3 - 4*x^2 + 8 es par y el x^2 + 2*x + 10 no lo es.

-- Definir el predicado
--    parPol :: Integral a => Polinomio a -> Bool
-- tal que (parPol p) se verifica si p es un polinomio par. Por ejemplo,
--    ghci> parPol (consPol 3 2 (consPol 2 (-4) (consPol 0 8 polCero)))
--    True
--    ghci> parPol (consPol 2 1 (consPol 1 2 (consPol 0 10 polCero)))
--    False
-- ---------------------------------------------------------------------

parPol :: Integral a => Polinomio a -> Bool
```

```
parPol p = esPolCero p || (even (coefLider p) && parPol (restoPol p))
```

6.3.5. Examen 5 (6 de mayo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (6 de mayo de 2015)
-- ---------------------------------------------------------------------

import Data.List
import I1M.Pol
import Data.Matrix
import I1M.Grafo
import I1M.BusquedaEnEspaciosDeEstados
import qualified Debug.Trace as T

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    conUno :: [Int] -> [Int]
-- tal que (conUno xs) es la lista de los elementos de xs que empiezan
-- por 1. Por ejemplo,
--    conUno [123,51,11,711,52]  == [123,11]
-- ---------------------------------------------------------------------

conUno :: [Int] -> [Int]
conUno xs = [x | x <- xs, head (show x) == '1']

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los árboles binarios con elementos en las
-- hojas y en los nodos mediante el tipo de dato
--    data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
--
-- Definir la función
--    aplica :: (a -> a) -> (a -> a) -> Arbol a -> Arbol a
-- tal que (aplica f g a) devuelve el árbol obtenido al aplicar la
-- función f a las hojas del árbol a y la función g a los nodos
-- interiores. Por ejemplo,
--    ghci> aplica (+1)(*2) (N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2)))
--    N 10 (N 4 (H 2) (H 3)) (N 6 (H 5) (H 3))
-- ---------------------------------------------------------------------

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

```
aplica :: (a -> a) -> (a -> a) -> Arbol a -> Arbol a
aplica f g (H x)     = H (f x)
aplica f g (N x i d) = N (g x) (aplica f g i) (aplica f g d)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Representamos los polinomios mediante el TAD de los
-- Polinomios (I1M.Pol). La parte par de un polinomio de coeficientes
-- enteros es el polinomio formado por sus monomios cuyos coeficientes
-- son números pares. Por ejemplo, la parte par de 4x^3+x^2-7x+6 es
-- 4x^3+6.
--
-- Definir la función
--    partePar :: Integral a => Polinomio a -> Polinomio a
-- tal que (partePar p) es la parte par de p. Por ejemplo,
--    ghci> partePar (consPol 3 4 (consPol 2 1 (consPol 0 6 polCero)))
--    4*x^3 + 6
-- ---------------------------------------------------------------------

partePar :: Integral a => Polinomio a -> Polinomio a
partePar p
    | esPolCero p = polCero
    | even b      = consPol n b (partePar r)
    | otherwise   = partePar r
    where n = grado p
          b = coefLider p
          r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Representaremos las matrices mediante la librería de
-- Haskell Data.Matrix.
--
-- Las posiciones frontera de una matriz de orden mxn son aquellas que
-- están en la fila 1 o la fila m o la columna 1 o la columna n. El
-- resto se dirán posiciones interiores. Observa que cada elemento en
-- una posición interior tiene exactamente 8 vecinos en la matriz.
--
-- Definir la función
--    marco :: Int -> Int -> Integer -> Matrix Integer
-- tal que (marco m n z) genera la matriz de dimensión mxn que
```

```
-- contiene el entero z en las posiciones frontera y 0 en las posiciones
-- interiores. Por ejemplo,
--    ghci> marco 5 5 1
--    ( 1 1 1 1 1 )
--    ( 1 0 0 0 1 )
--    ( 1 0 0 0 1 )
--    ( 1 0 0 0 1 )
--    ( 1 1 1 1 1 )
-- ---------------------------------------------------------------------

marco :: Int -> Int -> Integer -> Matrix Integer
marco m n z  = matrix m n f
    where f (i,j) | frontera m n (i,j) = 1
                  | otherwise          = 0

-- (frontera m n (i,j)) se verifica si (i,j) es una posición de la
-- frontera de las matrices de dimensión mxn.
frontera :: Int -> Int -> (Int,Int) -> Bool
frontera m n (i,j) = i == 1 || i == m || j == 1 || j == n

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Dada una matriz, un paso de transición genera una
-- nueva matriz de la misma dimensión pero en la que se ha sustituido
-- cada elemento interior por la suma de sus 8 vecinos. Los elementos
-- frontera no varían.
--
-- Definir la función
--   paso :: Matrix Integer -> Matrix Integer
-- tal que (paso t) calcula la matriz generada tras aplicar un paso de
-- transición a la matriz t. Por ejemplo,
--    ghci> paso (marco 5 5 1)
--    ( 1 1 1 1 1 )
--    ( 1 5 3 5 1 )
--    ( 1 3 0 3 1 )
--    ( 1 5 3 5 1 )
--    ( 1 1 1 1 1 )
-- ---------------------------------------------------------------------

paso :: Matrix Integer -> Matrix Integer
paso p = matrix m n f where
```

```
    m = nrows p
    n = ncols p
    f (i,j)
        | frontera m n (i,j) = 1
        | otherwise          = sum [p!(u,v) | (u,v) <- vecinos m n (i,j)]

-- (vecinos m n (i,j)) es la lista de las posiciones de los vecinos de
-- (i,j) en las matrices de dimensión mxn.
vecinos :: Int -> Int -> (Int,Int) -> [(Int,Int)]
vecinos m n (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)]
                           , b <- [max 1 (j-1)..min n (j+1)]
                           , (a,b) /= (i,j)]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    itPasos :: Int -> Matrix Integer -> Matrix Integer
-- tal que (itPasos k t) es la matriz obtenida tras aplicar k pasos de
-- transición a partir de la matriz t. Por ejemplo,
--    ghci> itPasos 10 (marco 5 5 1)
--    (       1       1       1       1       1 )
--    (       1 4156075 5878783 4156075       1 )
--    (       1 5878783 8315560 5878783       1 )
--    (       1 4156075 5878783 4156075       1 )
--    (       1       1       1       1       1 )
-- ---------------------------------------------------------------------

itPasos :: Int -> Matrix Integer -> Matrix Integer
itPasos k t = (iterate paso t) !! k

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Definir la función
--    pasosHasta :: Integer -> Int
-- tal que (pasosHasta k) es el número de pasos de transición a partir
-- de la matriz (marco 5 5 1) necesarios para que en la matriz
-- resultante aparezca un elemento mayor que k. Por ejemplo,
--    pasosHasta 4          ==  1
--    pasosHasta 6          ==  2
--    pasosHasta (2^2015)  ==  887
-- ---------------------------------------------------------------------
```

```
pasosHasta :: Integer -> Int
pasosHasta k =
    length (takeWhile (\t -> menores t k) (iterate paso (marco 5 5 1)))

-- (menores p k) se verifica si los elementos de p son menores o
-- iguales que k. Por ejemplo,
--    menores (itPasos 1 (marco 5 5 1)) 6  ==  True
--    menores (itPasos 1 (marco 5 5 1)) 4  ==  False
menores :: Matrix Integer -> Integer -> Bool
menores p k = and [p!(i,j) <= k | i <- [1..m], j <- [1..n]]
    where m = nrows p
          n = ncols p

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Representaremos los grafos mediante el TAD de los
-- grafos (I1M.Grafo).
--
-- Dado un grafo G, un ciclo en G es una secuencia de nodos de G
-- [v(1),v(2),v(3),...,v(n)] tal que:
--    1) (v(1),v(2)), (v(2),v(3)), (v(3),v(4)), ..., (v(n-1),v(n)) son
--       aristas de G,
--    2) v(1) = v(n), y
--    3) salvo v(1) = v(n), todos los v(i) son distintos entre sí.
--
-- Definir la función
--    esCiclo :: [Int] -> Grafo Int Int -> Bool
-- tal que (esCiclo xs g) se verifica si xs es un ciclo de g. Por
-- ejemplo, si g1 es el grafo definido por
--    g1 :: Grafo Int Int
--    g1 = creaGrafo D (1,4) [(1,2,0),(2,3,0),(2,4,0),(4,1,0)]
-- entonces
--    esCiclo [1,2,4,1] g1  ==  True
--    esCiclo [1,2,3,1] g1  ==  False
--    esCiclo [1,2,3] g1    ==  False
--    esCiclo [1,2,1] g1    ==  False
-- ---------------------------------------------------------------------

g1 :: Grafo Int Int
g1 = creaGrafo D (1,4) [(1,2,0),(2,3,0),(2,4,0),(4,1,0)]
```

```
esCiclo :: [Int] -> Grafo Int Int -> Bool
esCiclo vs g =
    all (aristaEn g) (zip vs (tail vs)) &&
    head vs == last vs &&
    length (nub vs) == length vs - 1

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. El grafo rueda de orden k es un grafo no dirigido
-- formado por
--    1) Un ciclo con k nodos [1,2,..,k], y
--    2) un nodo central k+1 unido con cada uno de los k nodos del
--       ciclo;
--    3) y con peso 0 en todas sus aristas.
--
-- Definir la función
--    rueda :: Int -> Grafo Int Int
-- tal que (rueda k) es el grafo rueda de orden k. Por ejemplo,
--    ghci> rueda 3
--    G D (array (1,4) [(1,[(2,0)]),(2,[(3,0)]),(3,[(1,0)]),
--                      (4,[(1,0),(2,0),(3,0)])])
-- ---------------------------------------------------------------------

rueda :: Int -> Grafo Int Int
rueda k = creaGrafo D (1,k+1) ([(i,i+1,0) | i <- [1..k-1]] ++
                               [(k,1,0)] ++
                               [(k+1,i,0) | i <- [1..k]])

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir la función
--    contieneCiclo :: Grafo Int Int -> Int -> Bool
-- tal que (contieneCiclo g k) se verifica si el grafo g contiene algún
-- ciclo de orden k. Por ejemplo,
--    contieneCiclo g1 4          ==  True
--    contieneCiclo g1 3          ==  False
--    contieneCiclo (rueda 5) 6  ==  True
-- ---------------------------------------------------------------------

contieneCiclo :: Grafo Int Int -> Int -> Bool
contieneCiclo g k = not (null (ciclos g k))
```

```
-- (caminosDesde g k v) es la lista de los caminos en el grafo g, de
-- longitud k, a partir del vértice v. Por ejemplo
--    caminosDesde g1 3 1  ==  [[1,2,3],[1,2,4]]
caminosDesde1 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminosDesde1 g k v = map reverse (aux [[v]])
    where aux [] = []
          aux ((x:vs):vss)
              | length (x:vs) == k = (x:vs) : aux vss
              | length (x:vs) >  k = aux vss
              | otherwise      = aux ([y:x:vs | y <- adyacentes g x] ++ vss)

caminosDesde2 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminosDesde2 g k v = map (reverse . snd) (aux [(1,[v])])
    where aux [] = []
          aux ((n,(x:vs)):vss)
              | n == k = (n,(x:vs)) : aux vss
              | n >  k = aux vss
              | otherwise = aux ([(n+1,y:x:vs) | y <- adyacentes g x] ++ vss)

-- 3ª definición de caminosDesde (con búsqueda en espacio de estados):
caminosDesde3 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminosDesde3 g k v = map reverse (buscaEE sucesores esFinal inicial)
    where inicial          = [v]
          esFinal vs       = length vs == k
          sucesores (x:xs) = [y:x:xs | y <- adyacentes g x]

-- 4ª definición de caminosDesde (con búsqueda en espacio de estados):
caminosDesde4 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminosDesde4 g k v = map (reverse . snd) (buscaEE sucesores esFinal inicial)
    where inicial            = (1,[v])
          esFinal (n,_)      = n == k
          sucesores (n,x:xs) = [(n+1,y:x:xs) | y <- adyacentes g x]

-- Comparación
--    ghci> let n = 10000 in length (caminosDesde1 (rueda n) (n+1) 1)
--    1
--    (3.87 secs, 20713864 bytes)
--    ghci> let n = 10000 in length (caminosDesde2 (rueda n) (n+1) 1)
--    1
--    (0.10 secs, 18660696 bytes)
```

```
--    ghci> let n = 10000 in length (caminosDesde3 (rueda n) (n+1) 1)
--    1
--    (0.42 secs, 16611272 bytes)
--    ghci> let n = 10000 in length (caminosDesde4 (rueda n) (n+1) 1)
--    1
--    (0.10 secs, 20118376 bytes)

-- En lo sucesivo usamos la 4ª definición
caminosDesde :: Grafo Int Int -> Int -> Int -> [[Int]]
caminosDesde = caminosDesde4

-- (ciclosDesde g k v) es la lista de los ciclos en el grafo g de orden
-- k a partir del vértice v. Por ejemplo,
--    ciclosDesde g1 4 1  ==  [[1,2,4,1]]
ciclosDesde :: Grafo Int Int -> Int -> Int -> [[Int]]
ciclosDesde g k v = [xs | xs <- caminosDesde g k v
                        , esCiclo xs g]

-- (ciclos g k) es la lista de los ciclos en el grafo g de orden
-- k. Por ejemplo,
--    ciclos g1 4  ==  [[1,2,4,1],[2,4,1,2],[4,1,2,4]]
ciclos :: Grafo Int Int -> Int -> [[Int]]
ciclos g k = concat [ciclosDesde g k v | v <- nodos g]

caminosDesde5 :: Grafo Int Int -> Int -> [[Int]]
caminosDesde5 g v = map (reverse . fst) (buscaEE sucesores esFinal inicial)
    where inicial             = ([v],[v])
          esFinal (x:_,ys)    = all (`elem` ys) (adyacentes g x)
          sucesores (x:xs,ys) = [(z:x:xs,z:ys) | z <- adyacentes g x
                                               , z `notElem` ys]

--    caminos g1  ==  [[1,2,3],[1,2,4],[2,3],[2,4,1],[3],[4,1,2,3]]
caminos :: Grafo Int Int -> [[Int]]
caminos g = concatMap (caminosDesde5 g) (nodos g)

--    todosCiclos g1  ==  [[1,2,4,1],[2,4,1,2]]

todosCiclos :: Grafo Int Int -> [[Int]]
todosCiclos g = [ys | (x:xs) <- caminos g
                    , let ys = (x:xs) ++ [x]
```

```
                    , esCiclo ys g]
```

6.3.6. Examen 6 (15 de junio de 2015)

El examen es común con el del grupo 4 (ver página 546).

6.3.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 5 (ver página 582).

6.3.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 5 (ver página 589).

6.3.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 5 (ver página 600).

6.4. Exámenes del grupo 4 (María J. Hidalgo)

6.4.1. Examen 1 (6 de Noviembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (6 de noviembre de 2014)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. La suma de la serie
--    1/1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 ...
-- es 2.
--
-- Definir la función
--    aproxima2:: Integer -> Float
-- tal que (aproxima2 n) es la aproximación de 2 obtenida mediante n
-- términos de la serie. Por ejemplo,
--    aproxima2 10  == 1.9990234
--    aproxima2 100 == 2.0
```

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

aproxima2:: Integer -> Float
aproxima2 n = sum [1 / (2^k) | k <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Decimos que los números m y n están relacionados si
-- tienen los mismos divisores primos.
--
-- Definir la función
--    relacionados :: Integer -> Integer -> Bool
-- tal que (relacionados m n) se verifica si m y n están relacionados.
-- Por ejemplo,
--    relacionados 24 32 == False
--    relacionados 24 12 == True
--    relacionados 24 2  == False
--    relacionados 18 12 == True
-- ---------------------------------------------------------------------

relacionados:: Integer -> Integer -> Bool
relacionados m n =
    nub (divisoresPrimos n) == nub (divisoresPrimos m)

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

esPrimo :: Integer -> Bool
esPrimo n = divisores n == [1,n]

divisores :: Integer -> [Integer]
divisores n =
    [x | x <- [1..n], n `rem` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. ¿Es cierto que si dos enteros positivos están
-- relacionados entonces uno es múltiplo del otro? Comprobarlo con
-- QuickCheck.
-- ---------------------------------------------------------------------
```

```
-- La propiedad es
prop_rel :: Integer -> Integer -> Property
prop_rel m n =
    m > 0 && n > 0 && relacionados m n ==>
    rem m n == 0 || rem n m == 0

-- La comprobación es
--    ghci> quickCheck prop_rel
--    *** Failed! Falsifiable (after 20 tests):
--    18
--    12

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que si p es primo, los
-- números relacionados con p son las potencias de p.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_rel_primos :: Integer -> Integer -> Property
prop_rel_primos p n =
    esPrimo p ==> esPotencia n p == relacionados n p

esPotencia :: Integer -> Integer -> Bool
esPotencia n p = nub (divisoresPrimos n) == [p]

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    mcdLista :: [Integer] -> Integer
-- tal que (mcdLista xs) es el máximo común divisor de los elementos de
-- xs. Por ejemplo,
--    mcdLista [3,4,5]  == 1
--    mcdLista [6,4,12] == 2
-- ---------------------------------------------------------------------

mcdLista :: [Integer] -> Integer
mcdLista [x]    = x
```

```haskell
mcdLista (x:xs) = gcd x (mcdLista xs)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que el resultado de
-- (mcdLista xs) divide a todos los elementos de xs, para cualquier
-- lista de enteros.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd_1 :: [Integer] -> Bool
prop_mcd_1 xs = and [rem x d == 0 | x <- xs]
    where d = mcdLista xs

-- La comprobación es
--    ghci> quickCheck prop_mcd_1
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck que, para n > 1, el máximo
-- común divisor de los n primeros primos es 1.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd_2 :: Int -> Property
prop_mcd_2 n = n > 1 ==> mcdLista (take n primos) == 1

primos = [x | x <- [2..], esPrimo x]

-- La comprobación es
--    ghci> quickCheck prop_mcd_2
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    menorLex:: (Ord a) => [a] -> [a] -> Bool
-- tal que menorLex sea el orden lexicográfico entre listas. Por
-- ejemplo,
--    menorLex "hola" "adios"         == False
--    menorLex "adios" "antes"        == True
--    menorLex "antes" "adios"        == False
```

```
--    menorLex [] [3,4]                == True
--    menorLex [3,4] []                == False
--    menorLex [1,2,3] [1,3,4,5]       == True
--    menorLex [1,2,3,3,3,3] [1,3,4,5] == True
-- ---------------------------------------------------------------------

menorLex:: Ord a => [a] -> [a] -> Bool
menorLex [] _          = True
menorLex _ []          = False
menorLex (x:xs) (y:ys) = x < y || (x == y && menorLex xs ys)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que menorLex coincide con la
-- relación predefinida <=.
-- ---------------------------------------------------------------------

-- La propiedad es
prop :: Ord a => [a] -> [a] -> Bool
prop xs ys = menorLex xs ys == (xs <= ys)

-- La comprobación es:
--    quickCheck prop
--    +++ OK, passed 100 tests.
```

6.4.2. Examen 2 (4 de Diciembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (4 de diciembre de 2014)
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    seleccionaDiv :: [Integer] -> [Integer] -> [Integer]
-- tal que (seleccionaDiv xs ys) es la lista de los elementos de xs que
-- son divisores de los correspondientes elementos de ys. Por ejemplo,
--    seleccionaDiv [1..5] [7,8,1,2,3]      == [1,2]
--    seleccionaDiv [2,5,3,7] [1,3,4,5,0,9] == []
```

```
--    seleccionaDiv (repeat 1) [3,5,8]      == [1,1,1]
--    seleccionaDiv [1..4] [2,4..]          == [1,2,3,4]
-- ---------------------------------------------------------------------

-- Por comprensión:
seleccionaDiv :: [Integer] -> [Integer] -> [Integer]
seleccionaDiv xs ys = [x | (x,y) <- zip xs ys, rem y x == 0]

-- Por recursión:
seleccionaDivR :: [Integer] -> [Integer] -> [Integer]
seleccionaDivR (x:xs) (y:ys) | rem y x == 0 = x: seleccionaDivR xs ys
                             | otherwise    = seleccionaDivR xs ys
seleccionaDivR _ _ = []

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Un número es especial si se cumple que la suma de cada
-- dos dígitos consecutivos es primo. Por ejemplo,
--    4116743 es especial pues 4+1, 1+1, 1+6, 6+7, 7+4 y 4+3 son primos.
--    41167435 no es especial porque 3+5 no es primo.
--
-- Definir la función
--    especial :: Integer -> Bool
-- tal que (especial n) se verifica si n es especial. Por ejemplo,
--    especial 4116743  == True
--    especial 41167435 == False
-- ---------------------------------------------------------------------

especial :: Integer -> Bool
especial n = all esPrimo (zipWith (+) xs (tail xs))
    where xs = digitos n

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

esPrimo :: Integer -> Bool
esPrimo x = x `elem` takeWhile (<=x) primos

primos:: [Integer]
primos = criba [2..]
    where criba (p:ps) = p: criba [x | x <- ps, rem x p /= 0]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Calcular el menor (y el mayor) número especial con 6
-- cifras.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [10^5..], especial n]
--    111111
--    ghci> head [n | n <- [10^6,10^6-1..], especial n]
--    989898

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    productoDigitosNN :: Integer -> Integer
-- tal que (productoDigitosNN n) es el producto de los dígitos no nulos
-- de n.
--    productoDigitosNN 2014  ==  8
-- ---------------------------------------------------------------------

productoDigitosNN :: Integer -> Integer
productoDigitosNN = product . filter (/=0) . digitos

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Consideremos la sucesión definida a partir de un
-- número d, de forma que cada elemento es la suma del anterior más el
-- producto de sus dígitos no nulos. Por ejemplo,
--    Si d =  1, la sucesión es 1,2,4,8,16,22,26,38,62,74,102,104, ...
--    Si d = 15, la sucesión es 15,20,22,26,38,62,74,102,104,108,...
--
-- Definir, usando iterate, la función
--    sucesion :: Integer -> [Integer]
-- tal que (sucesion d) es la sucesión anterior, empezando por d. Por
-- ejemplo,
--    take 10 (sucesion  1)   ==  [1,2,4,8,16,22,26,38,62,74]
--    take 10 (sucesion 15)   ==  [15,20,22,26,38,62,74,102,104,108]
-- ---------------------------------------------------------------------

sucesion :: Integer -> [Integer]
sucesion = iterate f
```

```
  where f x = x + productoDigitosNN x

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Llamamos sucesionBase a la sucesión que empieza en
-- 1. Probar con QuickCheck que cualquier otra sucesión que empiece en
-- d, con d > 0, tiene algún elemento común con la sucesionBase.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sucesion :: Integer -> Property
prop_sucesion d =
    d > 0 ==>
    [n | n <- sucesion d, n `elem` takeWhile (<=n) sucesionBase] /= []
    where sucesionBase = sucesion 1

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Consideremos el tipo de dato árbol, definido por
--   data Arbol a = H a | N a (Arbol a) (Arbol a)
-- y los siguientes ejemplos de árboles
--    ej1 = N 9 (N 3 (H 2) (H 4)) (H 7)
--    ej2 = N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7)
--
-- Definir la función
--    allArbol :: (t -> Bool) -> Arbol t -> Bool
-- tal que (allArbol p a) se verifica si todos los elementos del árbol
-- verifican p. Por ejemplo,
--    allArbol even ej1 == False
--    allArbol (>0) ej1 == True
-- ---------------------------------------------------------------------

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

ej1 = N 9 (N 3 (H 2) (H 4)) (H 7)
ej2 = N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7)

allArbol :: (t -> Bool) -> Arbol t -> Bool
```

```
allArbol p (H x)     = p x
allArbol p (N r i d) = p r && allArbol p i && allArbol p d

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    listasNoPrimos :: [[Integer]]
-- tal que listasNoPrimos es lista formada por las listas de números
-- no primos consecutivos. Por ejemplo,
--    take 7 listasNoPrimos == [[1],[4],[6],[8,9,10],[12],[14,15,16],[18]]
-- ---------------------------------------------------------------------

listasNoPrimos :: [[Integer]]
listasNoPrimos = aux [1..]
    where aux xs = takeWhile (not . esPrimo) xs :
                   aux (dropWhile esPrimo (dropWhile (not . esPrimo ) xs))
```

6.4.3. Examen 3 (23 de enero de 2015)

```
-- Informática: 3º examen de evaluación continua (23 de enero de 2014)
-- ---------------------------------------------------------------------

-- Puntuación: Cada uno de los 5 ejercicios vale 2 puntos.

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    sumaSinMuliplos :: Int -> [Int] -> Int
-- tal que (sumaSinMuliplos n xs) es la suma de los números menores o
-- iguales a n que no son múltiplos de ninguno de xs. Por ejemplo,
--    sumaSinMuliplos 10 [2,5]    ==  20
--    sumaSinMuliplos 10 [2,5,6]  ==  20
--    sumaSinMuliplos 10 [2,5,3]  ==  8
-- ---------------------------------------------------------------------

sumaSinMuliplos :: Int -> [Int] -> Int
sumaSinMuliplos n xs = sum [x | x <- [1..n], sinMultiplo x xs]

-- 1ª definición
sinMultiplo :: Int -> [Int] -> Bool
```

```
sinMultiplo n xs = all (\x -> n `mod` x /= 0) xs

-- 2ª definición (por comprensión):
sinMultiplo2 :: Int -> [Int] -> Bool
sinMultiplo2 n xs = and [n `mod` x /= 0 | x <- xs]

-- 3ª definición (por recursión):
sinMultiplo3 :: Int -> [Int] -> Bool
sinMultiplo3 n []     = True
sinMultiplo3 n (x:xs) = n `mod` x /= 0 && sinMultiplo3 n xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    menorFactorial :: (Int -> Bool) -> Int
-- tal que (menorFactorial p) es el menor n tal que n! cumple la
-- propiedad p. Por ejemplo,
--    menorFactorialP (>5)                     ==  3
--    menorFactorialP (\x -> x `mod` 21 == 0)  ==  7
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

menorFactorial :: (Int -> Bool) -> Int
menorFactorial p = head [n | (n,m) <- zip [0..] factoriales, p m]
    where factoriales = scanl (*) 1 [1..]

-- 2ª solución
-- ===========

menorFactorial2 :: (Int -> Bool) -> Int
menorFactorial2 p = 1 + length (takeWhile (not . p) factoriales)

factoriales :: [Int]
factoriales = [factorial n | n <- [1..]]

factorial :: Int -> Int
factorial n = product [1..n]

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

```
-- Ejercicio 3. Las expresiones aritméticas se pueden representar como
-- árboles con números en las hojas y operaciones en los nodos. Por
-- ejemplo, la expresión "9-2*4" se puede representar por el árbol
--       -
--      / \
--     9   *
--        / \
--       2   4
--
-- Definiendo el tipo de dato Arbol por
--    data Arbol = H Int | N (Int -> Int -> Int) Arbol Arbol
-- la representación del árbol anterior es
--    N (-) (H 9) (N (*) (H 2) (H 4))
--
-- Definir la función
--    valor :: Arbol -> Int
-- tal que (valor a) es el valor de la expresión aritmética
-- correspondiente al árbol a. Por ejemplo,
--    valor (N (-) (H 9) (N (*) (H 2) (H 4)))    ==  1
--    valor (N (+) (H 9) (N (*) (H 2) (H 4)))    ==  17
--    valor (N (+) (H 9) (N (div) (H 4) (H 2)))  ==  11
--    valor (N (+) (H 9) (N (max) (H 4) (H 2)))  ==  13
-- ---------------------------------------------------------------------

data Arbol = H Int | N (Int -> Int -> Int) Arbol Arbol

valor :: Arbol -> Int
valor (H x)     = x
valor (N f i d) = f (valor i) (valor d)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Se dice que el elemento y es un superior de x en una
-- lista xs si y > x y la posición de y es mayor que la de x en xs. Por
-- ejemplo, los superiores de 5 en [7,3,5,2,8,5,6,9,1] son el 8, el 6 y
-- el 9. El número de superiores de cada uno de sus elementos se
-- representa en la siguiente tabla
--    elementos:             [ 7, 3, 5, 2, 8, 5, 6, 9, 1]
--    número de superiores:    2  5  3  4  1  2  1  0  0
-- El elemento con máximo número de superiores es el 3 que tiene 5
-- superiores.
```

```
--
-- Definir la función
--    maximoNumeroSup :: Ord a => [a] -> Int
-- tal que (maximoNumeroSup xs) es el máximo de los números de
-- superiores de los elementos de xs. Por ejemplo,
--   maximoNumeroSup [7,3,5,2,8,4,6,9,1]  ==  5
--   maximoNumeroSup "manifestacion"      ==  10
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

maximoNumeroSup :: Ord a => [a] -> Int
maximoNumeroSup [] = 0
maximoNumeroSup xs = maximum [length (filter (z<) zs) | z:zs <- tails xs]

-- 2ª solución
-- ===========

maximoNumeroSup2 :: Ord a => [a] -> Int
maximoNumeroSup2 = maximum . numeroSup

-- (numeroSup xs) es la lista de los números de superiores de cada
-- elemento de xs. Por ejemplo,
--    numeroSup [7,3,5,2,8,5,6,9,1] ==  [2,5,3,4,1,2,1,0,0]
numeroSup :: Ord a => [a] -> [Int]
numeroSup []     = []
numeroSup [_]    = [0]
numeroSup (x:xs) = length (filter (>x) xs) : numeroSup xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que (maximoNumeroSup xs) es
-- igual a cero si, y sólo si, xs está ordenada de forma no decreciente.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_maximoNumeroSup :: [Int] ->  Bool
prop_maximoNumeroSup xs = (maximoNumeroSup xs == 0) == ordenada xs
    where ordenada xs = and (zipWith (>=) xs (tail xs))
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. En la siguiente figura, al rotar girando 90º en el
-- sentido del reloj la matriz de la izquierda se obtiene la de la
-- derecha
--    1 2 3        7 4 1
--    4 5 6        8 5 2
--    7 8 9        9 6 3
--
-- Definir la función
--    rota :: [[a]] -> [[a]]
-- tal que (rota xss) es la matriz obtenida girando 90º en el sentido
-- del reloj la matriz xss, Por ejemplo,
--    rota [[1,2,3],[4,5,6],[7,8,9]]  ==  [[7,4,1],[8,5,2],[9,6,3]]
--    rota ["abcd","efgh","ijkl"]     ==  ["iea","jfb","kgc","lhd"]
-- ---------------------------------------------------------------------

rota :: [[a]] -> [[a]]
rota []     = []
rota ([]:_) = []
rota xss    = reverse (map head xss) : rota (map tail xss)
```

6.4.4. Examen 4 (12 de marzo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (12 de marzo de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import I1M.PolOperaciones

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

```
-- Ejercicio 1.1. En este ejercicio, representemos las fracciones
-- mediante pares de números de enteros.
--
-- Definir la función
--    fracciones :: Integer -> [(Integer,Integer)]
-- tal que (fracciones n) es la lista con las fracciones propias
-- positivas, con denominador menor o igual que n. Por ejemplo,
--    fracciones 4 == [(1,2),(1,3),(2,3),(1,4),(3,4)]
--    fracciones 5 == [(1,2),(1,3),(2,3),(1,4),(3,4),(1,5),(2,5),(3,5),(4,5)]
-- ---------------------------------------------------------------------

fracciones :: Integer -> [(Integer,Integer)]
fracciones n = [(x,y) | y <- [2..n], x <- [1..y-1], gcd x y == 1]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    fraccionesOrd :: Integer -> [(Integer,Integer)]
-- tal que (fraccionesOrd n) es la lista con las fracciones propias
-- positivas ordenadas, con denominador menor o igual que n. Por
-- ejemplo,
--    fraccionesOrd 4 == [(1,4),(1,3),(1,2),(2,3),(3,4)]
--    fraccionesOrd 5 == [(1,5),(1,4),(1,3),(2,5),(1,2),(3,5),(2,3),(3,4),(4,5)]
-- ---------------------------------------------------------------------

fraccionesOrd :: Integer -> [(Integer,Integer)]
fraccionesOrd n = sortBy comp (fracciones n)
    where comp (a,b) (c,d) = compare (a*d) (b*c)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Todo número par se puede escribir como suma de números
-- pares de varias formas. Por ejemplo:
--    8 = 8
--      = 6 + 2
--      = 4 + 4
--      = 4 + 2 + 2
--      = 2 + 2 + 2 + 2
--
-- Definir la función
--    descomposicionesDecrecientes:: Integer -> [[Integer]]
-- tal que (descomposicionesDecrecientes n) es la lista con las
```

```
-- descomposiciones de n como suma de pares, en forma decreciente. Por
-- ejemplo,
--    ghci> descomposicionesDecrecientes 8
--    [[8],[6,2],[4,4],[4,2,2],[2,2,2,2]]
--    ghci> descomposicionesDecrecientes 10
--    [[10],[8,2],[6,4],[6,2,2],[4,4,2],[4,2,2,2],[2,2,2,2,2]]
--
-- Calcular el número de descomposiciones de 40.
-- ---------------------------------------------------------------------

descomposicionesDecrecientes:: Integer -> [[Integer]]
descomposicionesDecrecientes 0 = [[0]]
descomposicionesDecrecientes n = aux n [n,n-2..2]
    where aux _ [] = []
          aux n (x:xs) | x > n     = aux n xs
                       | x == n    = [n] : aux n xs
                       | otherwise = map (x:) (aux (n-x) (x:xs)) ++ aux n xs

-- El cálculo es
--    ghci> length (descomposicionesDecrecientes 40)
--    627

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideremos los árboles binarios con etiquetas en las
-- hojas y en los nodos. Por ejemplo,
--          5
--         / \
--        2   4
--          / \
--         7   1
--            / \
--           2   3
--
-- Un camino es una sucesión de nodos desde la raiz hasta una hoja. Por
-- ejemplo, [5,2] y [5,4,1,2] son caminos que llevan a 2, mientras que
-- [5,4,1] no es un camino, pues no lleva a una hoja.
--
-- Definimos el tipo de dato Arbol y el ejemplo por
--    data Arbol = H Int | N Arbol Int Arbol
--                 deriving Show
```

```
--
--    arb1:: Arbol
--    arb1 = N (H 2) 5 (N (H 7) 4 (N (H 2) 1 (H 3)))
--
-- Definir la función
--    maxLong :: Int -> Arbol -> Int
-- tal que (maxLong x a) es la longitud máxima de los caminos que
-- terminan en x. Por ejemplo,
--    maxLong 3 arb1 == 4
--    maxLong 2 arb1 == 4
--    maxLong 7 arb1 == 3
-- ---------------------------------------------------------------------

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

arb1:: Arbol
arb1 = N (H 2) 5 (N (H 7) 4 (N (H 2) 1 (H 3)))

-- 1ª solución (calculando los caminos)
-- ------------------------------------

-- (caminos x a) es la lista de los caminos en el árbol a desde la raíz
-- hasta las hojas x. Por ejemplo,
--    caminos 2 arb1 == [[5,2],[5,4,1,2]]
--    caminos 3 arb1 == [[5,4,1,3]]
--    caminos 1 arb1 == []
caminos :: Int -> Arbol -> [[Int]]
caminos x (H y) | x == y    = [[x]]
                | otherwise = []
caminos x (N i r d) = map (r:) (caminos x i ++ caminos x d)

maxLong1 :: Int -> Arbol -> Int
maxLong1 x a = maximum (0: map length (caminos x a))

-- 2ª solución
-- -----------

maxLong2 :: Int -> Arbol -> Int
maxLong2 x a = maximum (0 : aux x a)
```

```
    where aux x (H y) | x == y    = [1]
                      | otherwise = []
          aux x (N i r d) = map (+1) (aux x i ++ aux x d)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un elemento de una matriz es un máximo local si es un
-- elemento interior, que es mayor que todos sus vecinos. Por ejemplo,
-- en la matriz
--     [[1,0,0,1],
--      [0,2,0,3],
--      [0,0,0,5],
--      [3,5,7,6],
--      [1,2,3,4]]
-- los máximos locales son 2 (en la posición (2,2)) y 7 (en la posición
-- (4,3)).
--
-- Definimos el tipo de las matrices, mediante
--    type Matriz a = Array (Int,Int) a
-- y el ejemplo anterior por
--    ej1 :: Matriz Int
--    ej1 = listArray ((1,1),(5,4)) (concat [[1,0,0,1],
--                                          [0,2,0,3],
--                                          [0,0,0,5],
--                                          [3,5,7,6],
--                                          [1,2,3,4]])
--
-- Definir la función
--    maximosLocales :: Matriz Int -> [((Int,Int),Int)]
-- tal que (maximosLocales p) es la lista de las posiciones en las que
-- hay un máximo local, con el valor correspondiente. Por ejemplo,
--    maximosLocales ej1 == [((2,2),2),((4,3),7)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

ej1 :: Matriz Int
ej1 = listArray ((1,1),(5,4)) (concat [[1,0,0,1],
                                       [0,2,0,3],
                                       [0,0,0,5],
                                       [3,5,7,6],
```

```
                                        [1,2,3,4]])

maximosLocales :: Matriz Int -> [((Int,Int),Int)]
maximosLocales p =
    [((i,j),p!(i,j)) | i <- [1..m], j <- [1..n], posicionMaxLocal (i,j) p]
        where (_,(m,n)) = bounds p


-- (posicionMaxLocal (i,j) p) se verifica si (i,j) es la posición de un
-- máximo local de la matriz p. Por ejemplo,
--    posicionMaxLocal (2,2) ej1  ==  True
--    posicionMaxLocal (2,3) ej1  ==  False
posicionMaxLocal :: (Int,Int) -> Matriz Int -> Bool
posicionMaxLocal (i,j) p =
    esInterior (i,j) p && all (< p!(i,j)) (vecinosInterior (i,j) p)

-- (esInterior (i,j) p) se verifica si (i,j) es una posición interior de
-- la matriz p.
esInterior:: (Int,Int) -> Matriz a -> Bool
esInterior (i,j) p = i /= 1 && i /= m && j /= 1 && j /= n
    where (_,(m,n)) = bounds p

-- (indicesVecinos (i,j)) es la lista de las posiciones de los
-- vecinos de la posición (i,j). Por ejemplo,
--    ghci> indicesVecinos (2,2)
--    [(1,1),(1,2),(1,3),(2,1),(2,3),(3,1),(3,2),(3,3)]
indicesVecinos :: (Int,Int) -> [(Int,Int)]
indicesVecinos (i,j) =
    [(i+a,j+b) | a <- [-1,0,1], b <- [-1,0,1], (a,b) /= (0,0)]

-- (vecinosInterior (i,j) p) es la lista de los valores de los vecinos
-- de la posición (i,j) en la matriz p. Por ejemplo,
--    vecinosInterior (4,3) ej1  ==  [0,0,5,5,6,2,3,4]
vecinosInterior (i,j) p =
    [p!(k,l) | (k,l) <- indicesVecinos (i,j)]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Los polinomios de Bell forman una sucesión de
-- polinomios, definida como sigue:
--    B_0(x) = 1 (polinomio unidad)
```

```
--    B_n(x) = x*[B_n(x) + B_n'(x)]
-- Por ejemplo,
--    B_0(x) = 1
--    B_1(x) = x*(1+0)                      = x
--    B_2(x) = x*(x+1)                      = x^2+x
--    B_3(x) = x*(x^2+x + 2x+1)             = x^3+3x^2+x
--    B_4(x) = x*(x^3+3x^2+x + 3x^2+6x+1) = x^4+6x^3+7x^2+x
--
-- Definir la función
--    polBell :: Int -> Polinomio Int
-- tal que (polBell n) es el polinomio de Bell de grado n. Por ejemplo,
--    polBell1 4  ==  x^4 + 6*x^3 + 7*x^2 + 1*x
--
-- Calcular el coeficiente de x^2 en el polinomio B_30.
-- ---------------------------------------------------------------------

-- 1ª solución (por recursión)
polBell1 :: Integer -> Polinomio Integer
polBell1 0 = polUnidad
polBell1 n = multPol (consPol 1 1 polCero) (sumaPol p (derivada p))
    where p = polBell1 (n-1)

-- 2ª solución (evaluación perezosa)
polBell2 :: Integer -> Polinomio Integer
polBell2 n = sucPolinomiosBell `genericIndex` n

sucPolinomiosBell :: [Polinomio Integer]
sucPolinomiosBell = iterate f polUnidad
    where f p = multPol (consPol 1 1 polCero) (sumaPol p (derivada p))

-- El cálculo es
--    ghci> coeficiente 2 (polBellP1 30)
--    536870911
```

6.4.5. Examen 5 (30 de abril de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (30 de abril de 2015)
-- ---------------------------------------------------------------------

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

```
-- § Librerías auxiliares                                            --
-- ---------------------------------------------------------------------

import Data.Numbers.Primes
import Test.QuickCheck
import Data.List
import Data.Array
import I1M.Grafo
import I1M.Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número n es especial si al unir las cifras de sus
-- factores primos, se obtienen exactamente las cifras de n, aunque
-- puede ser en otro orden. Por ejemplo, 1255 es especial, pues los
-- factores primos de 1255 son 5 y 251.
--
-- Definir la función
--    esEspecial :: Integer -> Bool
-- tal que (esEspecial n) se verifica si un número n es especial. Por
-- ejemplo,
--    esEspecial 1255 == True
--    esEspecial 125  == False
-- ---------------------------------------------------------------------

esEspecial :: Integer -> Bool
esEspecial n =
    sort (show n) == sort (concatMap show (nub (primeFactors n)))

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que todo número primo es
-- especial.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_primos:: Integer -> Property
prop_primos n =
    isPrime (abs n) ==> esEspecial (abs n)

-- La comprobación es
--    ghci> quickCheck prop_primos
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Calcular los 5 primeros números especiales que no son
-- primos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> take 5 [n | n <- [2..], esEspecial n, not (isPrime n)]
--    [735,1255,3792,7236,11913]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Consideremos las relaciones binarias homogéneas,
-- representadas por el siguiente tipo
--    type Rel a = ([a],[(a,a)])
-- y las matrices, representadas por
--    type Matriz a = Array (Int,Int) a
--
-- Dada una relación r sobre un conjunto de números enteros, la matriz
-- asociada a r es una matriz booleana p (cuyos elementos son True o
-- False), tal que p(i,j) = True si y sólo si i está relacionado con j
-- mediante la relación r.
--
-- Definir la función
--    matrizRB:: Rel Int -> Matriz Bool
-- tal que (matrizRB r) es la matriz booleana asociada a r. Por ejemplo,
--    ghci> matrizRB ([1..3],[(1,1), (1,3), (3,1), (3,3)])
--    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 ([1..3],[(1,3), (3,1)])
--    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.
-- ---------------------------------------------------------------------

type Rel a    = ([a],[(a,a)])
```

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

-- 1ª definición (con array, universo y grafo):
matrizRB:: Rel Int -> Matriz Bool
matrizRB r =
    array ((1,1),(n,n))
          [((a,b), (a,b) `elem` grafo r) | a <- [1..n], b <- [1..n]]
    where n = maximum (universo r)

universo :: Eq a => Rel a -> [a]
universo (us,_) = us

grafo :: Eq a => Rel a -> [(a,a)]
grafo (_,ps) = ps

-- 2ª definición (con listArray y sin universo ni grafo):
matrizRB2:: Rel Int -> Matriz Bool
matrizRB2 r =
    listArray ((1,1),(n,n))
              [(a,b) `elem` snd r | a <- [1..n], b <- [1..n]]
    where n = maximum (fst r)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Dado un grafo G = (V, E),
--    + la distancia entre dos nodos de G es el valor absoluto de su
--      diferencia,
--    + la anchura de un nodo x es la máxima distancia entre x y todos
--      los nodos adyacentes y
--    + la anchura del grafo es la máxima anchura de sus nodos.
--
-- Definir la función
--    anchuraG :: Grafo Int Int -> Int
-- tal que (anchuraG g) es la anchura del grafo g. Por ejemplo, si g es
-- el grafo definido a continuación,
--    g :: Grafo Int Int
--    g = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
--                            (2,4,55),(2,5,32),
--                            (3,4,61),(3,5,44),
--                            (4,5,93)]
-- entonces
```

```
--    anchuraG g == 4
-- ---------------------------------------------------------------------

g :: Grafo Int Int
g = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                        (2,4,55),(2,5,32),
                        (3,4,61),(3,5,44),
                        (4,5,93)]

anchuraG :: Grafo Int Int -> Int
anchuraG g = maximum [abs(a-b) | (a,b,_) <- aristas g]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar experimentalmente que la anchura del grafo
-- cíclico de orden n (para n entre 1 y 20) es n-1.
-- ---------------------------------------------------------------------

-- La propiedad es
propG :: Int -> Bool
propG n = anchuraG (grafoCiclo n) == n-1

grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n = creaGrafo ND (1,n) ([(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)])

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. 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 [1,8,2,4,5])  ==  8
-- ---------------------------------------------------------------------

-- 1ª solución
mayor :: Ord a => Monticulo a -> a
mayor m | esVacio r = menor m
        | otherwise = mayor r
        where r = resto m
```

```
-- 2ª solución
mayor2 :: Ord a => Monticulo a -> a
mayor2 m = last (monticulo2Lista m)

monticulo2Lista :: Ord a => Monticulo a -> [a]
monticulo2Lista m | esVacio m = []
                  | otherwise = menor m : monticulo2Lista (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    minMax :: Ord a => Monticulo a -> Maybe (a, a)
-- tal que (minMax m) es justamente el par formado por el menor y el
-- mayor elemento de m, si el montículo m es no vacío. Por ejemplo,
--    minMax (foldr inserta vacio [4,8,2,1,5])  ==  Just (1,8)
--    minMax (foldr inserta vacio [4])          ==  Just (4,4)
--    minMax vacio                              ==  Nothing
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Dada una lista de números naturales xs, la
-- codificación de Gödel de xs se obtiene multiplicando las potencias de
-- los primos sucesivos, siendo los exponentes los elementos de xs. Por
-- ejemplo, si xs = [6,0,4], la codificación de xs es
--    2^6 * 3^0 * 5^4 = 64 * 1 * 625 = 40000.
--
-- Definir la función
--    codificaG :: [Integer] -> Integer
-- tal que (codificaG xs) es la codificación de Gödel de xs. Por
-- ejemplo,
--    codificaG [6,0,4]           == 40000
--    codificaG [3,1,1]           == 120
--    codificaG [3,1,0,0,0,0,0,1] == 456
--    codificaG [1..6]            == 4199506113235182750
-- ---------------------------------------------------------------------
```

```
codificaG :: [Integer] -> Integer
codificaG xs = product (zipWith (^) primes xs)

-- Se puede eliminar el argumento:
codificaG2   :: [Integer] -> Integer
codificaG2 = product . zipWith (^) primes

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--     decodificaG :: Integer -> [Integer]
-- tal que (decodificaG n) es la lista xs cuya codificación es n. Por
-- ejemplo,
--    decodificaG 40000               == [6,0,4]
--    decodificaG 120                 == [3,1,1]
--    decodificaG 456                 == [3,1,0,0,0,0,0,1]
--    decodificaG 4199506113235182750 == [1,2,3,4,5,6]
-- ---------------------------------------------------------------------

decodificaG :: Integer -> [Integer]
decodificaG n = aux primes (group $ primeFactors n)
  where aux _ [] = []
        aux (x:xs) (y:ys) | x == head y = genericLength y : aux xs ys
                          | otherwise   = 0 : aux xs (y:ys)

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Comprobar con QuickCheck que ambas funciones son
-- inversas.
-- ---------------------------------------------------------------------

-- Las propiedades son
propCodifica1 :: [Integer] -> Bool
propCodifica1 xs =
    decodificaG (codificaG ys) == ys
    where ys = map ((+1) . abs) xs

propCodifica2:: Integer -> Property
propCodifica2 n =
    n > 0 ==> codificaG (decodificaG n) == n

-- Las comprobaciones son
```

```
--    ghci> quickCheck propCodifica1
--    +++ OK, passed 100 tests.
--
--    ghci> quickCheck propCodifica2
--    +++ OK, passed 100 tests.
```

6.4.6. Examen 6 (15 de junio de 2015)

```
-- Informática: 6º examen de evaluación continua (15 de junio de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                             --
-- ---------------------------------------------------------------------

import Data.Numbers.Primes
import Data.List
import I1M.PolOperaciones
import Test.QuickCheck
import Data.Array
import Data.Char
import Data.Matrix

-- ---------------------------------------------------------------------
-- Ejercicio 1. Sea p(n) el n-ésimo primo y sea r el resto de dividir
-- (p(n)-1)^n + (p(n)+1)^n por p(n)^2. Por ejemplo,
--    si n = 3, entonces p(3) =  5 y r = ( 4^3 +  6^3) mod  (5^2) =   5
--    si n = 7, entonces p(7) = 17 y r = (16^7 + 18^7) mod (17^2) = 238
--
-- Definir la función
--    menorPR :: Integer -> Integer
-- tal que (menorPR x) es el menor n tal que el resto de dividir
-- (p(n)-1)^n + (p(n)+1)^n por p(n)^2 es mayor que x. Por ejemplo,
--    menorPR 100     == 5
--    menorPR 345     == 9
--    menorPR 1000    == 13
--    menorPR (10^9)  == 7037.
--    menorPR (10^10) == 21035
--    menorPR (10^12) == 191041
-- ---------------------------------------------------------------------
```

```
-- 1ª solución
-- ===========

menorPR1 :: Integer -> Integer
menorPR1 x =
    head [n | (n,p) <- zip [1..] primes
            , (((p-1)^n + (p+1)^n) `mod` (p^2)) > x]

-- Segunda solución (usando el binomio de Newton)
-- ==============================================

-- Desarrollando por el binomio de Newton
--    (p+1)^n = C(n,0)p^n + C(n,1)p^(n-1) +...+ C(n,n-1)p + 1
--    (p-1)^n = C(n,0)p^n - C(n,1)p^(n-1) +...+ C(n,n-1)p + (-1)^n
-- Sumando se obtiene (según n sea par o impar)
--    2*C(n,0)p^n + 2*C(n,n-2)p^(n-1) +...+ 2*C(n,2)p^2 + 2
--    2*C(n,0)p^n + 2*C(n,n-2)p^(n-1) +...+ 2*C(n,1)p^1
-- Al dividir por p^2, el resto es (según n sea par o impar) 2 ó 2*C(n,1)p

-- (restoM n p) es el resto de de dividir (p-1)^n + (p+1)^n por p^2.
restoM :: Integer -> Integer -> Integer
restoM n p | even n    = 2
           | otherwise = 2*n*p `mod`(p^2)

menorPR2 :: Integer -> Integer
menorPR2 x = head [n | (n,p) <- zip [1..] primes, restoM n p > x]

-- Comparación de eficiencia
--    ghci> menorPR1 (3*10^8)
--    3987
--    (2.44 secs, 120291676 bytes)
--    ghci> menorPR2 (3*10^8)
--    3987
--    (0.04 secs, 8073900 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumaPosteriores :: [Int] -> [Int]
-- tal que (sumaPosteriores xs) es la lista obtenida sustituyendo cada
-- elemento de xs por la suma de los elementos posteriores. Por ejemplo,
```

```
--    sumaPosteriores [1..8]        == [35,33,30,26,21,15,8,0]
--    sumaPosteriores [1,-3,2,5,-8] == [-4,-1,-3,-8,0]
--
-- Comprobar con QuickCheck que el último elemento de la lista
-- (sumaPosteriores xs) siempre es 0.
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
sumaPosteriores1 :: [Int] -> [Int]
sumaPosteriores1 []     = []
sumaPosteriores1 (x:xs) = sum xs : sumaPosteriores1 xs

-- 2ª definición (sin argumentos)
sumaPosteriores2 :: [Int] -> [Int]
sumaPosteriores2 = map sum . tail . tails

-- 3ª definición (con scanr)
sumaPosteriores3 :: [Int] -> [Int]
sumaPosteriores3 = tail . scanr (+) 0

-- La propiedad es
propSumaP:: [Int] -> Property
propSumaP xs = not (null xs) ==> last (sumaPosteriores1 xs) == 0

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la constante
--    sucesionD :: String
-- tal que su valor es la cadena infinita "1234321234321234321..."
-- formada por la repetición de los dígitos 123432. Por ejemplo,
--    ghci> take 50 sucesionD
--    "12343212343212343212343212343212343212343212343212"
-- ---------------------------------------------------------------------

-- 1ª definición (con cycle):
sucesionD :: String
sucesionD = cycle "123432"
```

```
-- 2ª definición (con repeat):
sucesionD2 :: String
sucesionD2 = concat $ repeat "123432"

-- 3ª definición (por recursión):
sucesionD3 :: String
sucesionD3 = "123432" ++ sucesionD4

-- Comparación de eficiencia
--    ghci> sucesionD !! (2*10^7)
--    '3'
--    (0.16 secs, 1037132 bytes)
--    ghci> sucesionD2 !! (2*10^7)
--    '3'
--    (3.28 secs, 601170876 bytes)
--    ghci> sucesionD3 !! (2*10^7)
--    '3'
--    (0.17 secs, 1033344 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. La sucesión anterior se puede partir en una sucesión
-- de números, de forma que la suma de los dígitos de dichos números
-- forme la sucesión de los números naturales, como se observa a
-- continuación:
--     1, 2, 3, 4, 32, 123, 43, 2123, 432, 1234, 32123, ...
--     1, 2, 3, 4,  5,   6,  7,    8,   9,   10,    11, ...
--
-- Definir la sucesión
--    sucesionN :: [Integer]
-- tal que sus elementos son los números de la partición anterior. Por
-- ejemplo,
--    ghci> take 11 sucesionN
--    [1,2,3,4,32,123,43,2123,432,1234,32123]
-- ---------------------------------------------------------------------

sucesionN :: [Int]
sucesionN = aux [1..] sucesionD
    where aux (n:ns) xs = read ys : aux ns zs
              where (ys,zs) = prefijoSuma n xs
```

```
-- (prefijoSuma n xs) es el par formado por el primer prefijo de xs cuyo
-- suma es n y el resto de xs. Por ejemplo,
--    prefijoSuma 6 "12343"  ==  ("123","43")
prefijoSuma :: Int -> String -> (String,String)
prefijoSuma n xs =
    head [(us,vs) | (us,vs) <- zip (inits xs) (tails xs)
                  , sumaD us == n]

-- (sumaD xs) es la suma de los dígitos de xs. Por ejemplo,
--    sumaD "123"  ==  6
sumaD :: String -> Int
sumaD = sum . map digitToInt

-- ---------------------------------------------------------------------
-- Ejercicio 4. El polinomio cromático de un grafo calcula el número de
-- maneras en las cuales puede ser coloreado el grafo usando un número
-- de colores dado, de forma que dos vértices adyacentes no tengan el
-- mismo color.
--
-- En el caso del grafo completo de n vértices, su polinomio cromático
-- es P(n,x) = x(x-1)(x-2) ... (x-(n-1)). Por ejemplo,
--    P(3,x) = x(x-1)(x-2)      = x^3 - 3*x^2 + 2*x
--    P(4,x) = x(x-1)(x-2)(x-3) = x^4 - 6*x^3 + 11*x^2 - 6*x
-- Lo que significa que P(4)(x) es el número de formas de colorear el
-- grafo completo de 4 vértices con x colores. Por tanto,
--    P(4,2) =  0 (no se puede colorear con 2 colores)
--    P(4,4) = 24 (hay 24 formas de colorearlo con 4 colores)
--
-- Definir la función
--      polGC:: Int -> Polinomio Int
-- tal que (polGC n) es el polinomio cromático del grafo completo de n
-- vértices. Por ejemplo,
--      polGC 4  ==  x^4 + -6*x^3 + 11*x^2 + -6*x
--      polGC 5  ==  x^5 + -10*x^4 + 35*x^3 + -50*x^2 + 24*x
--
-- Comprobar con QuickCheck que si el número de colores (x) coincide con
-- el número de vértices del grafo (n), el número de maneras de colorear
-- el grafo es n!.
--
```

```
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_polGC
--    +++ OK, passed 100 tests.
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

polGC :: Int -> Polinomio Int
polGC 0 = consPol 0 1 polCero
polGC n = multPol (polGC (n-1)) (consPol 1 1 (consPol 0 (-n+1) polCero))

-- 2ª solución
-- ===========

polGC2 :: Int -> Polinomio Int
polGC2 n = multLista (map polMon [0..n-1])

-- (polMon n) es el monomio x-n. Por ejemplo,
--    polMon 3  ==  1*x + -3
polMon:: Int -> Polinomio Int
polMon n = consPol 1 1 (consPol 0 (-n) polCero)

-- (multLista ps) es el producto de la lista de polinomios ps.
multLista :: [Polinomio Int] -> Polinomio Int
multLista []     = polUnidad
multLista (p:ps) = multPol p (multLista ps)

-- La función multLista se puede definir por plegado
multLista2 :: [Polinomio Int] -> Polinomio Int
multLista2 = foldr multPol polUnidad

-- La propiedad es
prop_polGC :: Int -> Property
prop_polGC n =
    n > 0 ==> valor (polGC n) n == product [1..n]

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_polGC
```

```
--    +++ OK, passed 100 tests.
--    (0.04 secs, 7785800 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 5:

-- Consideramos un tablero de ajedrez en el que hay un único caballo y
-- lo representamos por una matriz con ceros en todas las posiciones,
-- excepto en la posición del caballo que hay un 1.

-- Definimos el tipo de las matrices:

type Matriz a = Array (Int,Int) a

-- (a) Definir una función
--     matrizC:: (Int,Int) -> Matriz Int
-- tal que, dada la posición (i,j) donde está el caballo, obtiene la
-- matriz correspondiente. Por ejemplo,
--   elems (matrizC (1,1))
--  [1,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0,
--   0,0,0,0,0,0,0,0]

matrizC:: (Int,Int) -> Matrix Int
matrizC (i,j) = setElem 1 (i,j) (zero 8 8)

-- (b) Definir una función
--     posicionesC :: (Int,Int) -> [(Int,Int)]
-- tal que dada la posición (i,j) de un caballo, obtiene la lista
-- con las posiciones posibles a las que se puede mover el caballo.
-- Por ejemplo,
-- posicionesC (1,1) == [(2,3),(3,2)]
-- posicionesC (3,4) == [(2,2),(2,6),(4,2),(4,6),(1,3),(1,5),(5,3),(5,5)]

posicionesC :: (Int,Int) -> [(Int,Int)]
```

```
posicionesC (i,j) =
    filter p [(i-1,j-2),(i-1,j+2),(i+1,j-2),(i+1,j+2),
              (i-2,j-1),(i-2,j+1),(i+2,j-1),(i+2,j+1)]
    where p (x,y) = x >= 1 && x <= 8 && y >= 1 && y <= 8

-- (c) Definir una función
--     saltoC:: Matriz Int -> (Int,Int) -> [Matriz Int]
-- tal que, dada una matriz m con un caballo en la posición (i,j),
-- obtiene la lista con las matrices que representan cada una de los
-- posibles movimientos del caballo.

saltoC:: Matrix Int -> (Int,Int) -> [Matrix Int]
saltoC m (i,j) = map matrizC (posicionesC (i,j))

-- o bien, sin usar matrizC

saltoC':: Matrix Int -> (Int,Int) -> [Matrix Int]
saltoC' m (i,j) = map f (posicionesC (i,j))
    where f (k,l) = setElem 0 (i,j) (setElem 1 (k,l) m)

-- También se puede definir obviando la matriz:

saltoCI:: (Int,Int) -> [Matrix Int]
saltoCI = map matrizC . posicionesC


-- saltoC m1 (1,1)
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 1 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ,
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 1 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
```

```
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )
-- ( 0 0 0 0 0 0 0 0 )

-- (d) Definir una función
--       juego:: IO ()
-- que realice lo siguiente: pregunte por la posición del caballo en el
-- tablero y por la casilla hacia la que queremos moverlo y nos devuelva
-- en pantalla "Correcta" o "Incorrecta". Por ejemplo,

--    juego
--    Introduce la posición actual del caballo
--    fila: 1
--    columna: 1
--    Introduce la posición hacia la que quieres moverlo
--    fila: 4
--    columna: 2
--    Incorrecta

--    juego
--    Introduce la posición actual del caballo
--    fila: 3
--    columna: 4
--    Introduce la posición hacia la que quieres moverlo
--    fila: 1
--    columna: 5
--    Correcta

juego :: IO ()
juego = do putStrLn "Introduce la posición actual del caballo "
           putStr "fila: "
           a <- getLine
           let i = read a
           putStr "columna: "
           b <- getLine
           let j = read b
           putStrLn "Introduce la posición hacia la que quieres moverlo "
           putStr "fila: "
           a2 <- getLine
```

```
           let k = read a2
           putStr "columna: "
           b2 <- getLine
           let l = read b
           putStrLn (if (k,l) `elem` posicionesC (i,j)
                     then "Correcta"
                     else "Incorrecta")

-- ================================================
-- Ejercicio 5: Con Array. Matrices. Entrada/salida
-- ================================================

-- Los mismos ejercicios, pero usando Array en vez de la librería de
-- matrices.

matrizC2:: (Int,Int) -> Array (Int,Int) Int
matrizC2 (i,j) = array ((1,1), (8,8)) [((k,l), f (k,l)) | k <-[1..8],
                                                           l <- [1..8]]
    where f (k,l) | (k,l) == (i,j) = 1
                  | otherwise      = 0

-- Ejemplo:
m1_2:: Array (Int,Int) Int
m1_2 = matrizC2 (1,1)

saltoC2:: Array (Int,Int) Int -> (Int,Int) -> [Array (Int,Int) Int]
saltoC2 m (i,j) = map matrizC2 (posicionesC (i,j))

saltoCI2:: (Int,Int) -> [Array (Int,Int) Int]
saltoCI2 = map matrizC2 . posicionesC
```

6.4.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 5 (ver página 582).

6.4.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 5 (ver página 582).

6.4.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 5 (ver página 600).

6.5. Exámenes del grupo 5 (José A. Alonso y Luis Valencia)

6.5.1. Examen 1 (5 de Noviembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (5 de noviembre de 2014)
-- ---------------------------------------------------------------------

import Data.Char
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    esPotencia :: Integer -> Integer -> Bool
-- tal que (esPotencia x a) se verifica si x es una potencia de a. Por
-- ejemplo,
--    esPotencia 32 2  ==  True
--    esPotencia 42 2  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
esPotencia :: Integer -> Integer -> Bool
esPotencia x a = x `elem` [a^n | n <- [0..x]]

-- 2ª definición (por recursión):
esPotencia2 :: Integer -> Integer -> Bool
esPotencia2 x a = aux x a 0
    where aux x a b | b > x     = False
                    | otherwise = x == a ^ b || aux x a (b+1)

-- 3ª definición (por recursión):
esPotencia3 :: Integer -> Integer -> Bool
esPotencia3 0 _ = False
esPotencia3 1 a = True
esPotencia3 _ 1 = False
```

```
esPotencia3 x a = rem x a == 0 && esPotencia3 (div x a) a

-- La propiedad de equivalencia es
prop_equiv_esPotencia :: Integer -> Integer -> Property
prop_equiv_esPotencia x a =
    x > 0 && a > 0 ==>
    esPotencia2 x a == b &&
    esPotencia3 x a == b
    where b = esPotencia x a

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que, para cualesquiera números
-- enteros positivos x y a, x es potencia de a si y sólo si x^2 es
-- potencia de a^2.
-- ---------------------------------------------------------------------

-- Propiedad de potencia
prop_potencia :: Integer -> Integer -> Property
prop_potencia x a =
    x > 0 && a > 0 ==> esPotencia x a == esPotencia (x*x) (a*a)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    intercambia :: String -> String
-- tal que (intercambia xs) es la cadena obtenida poniendo la mayúsculas
-- de xs en minúscula y las minúsculas en mayúscula. Por ejemplo,
--    intercambia "Hoy es 5 de Noviembre"  ==  "hOY ES 5 DE nOVIEMBRE"
--    intercambia "hOY ES 5 DE nOVIEMBRE"  ==  "Hoy es 5 de Noviembre"
-- ---------------------------------------------------------------------

intercambia :: String -> String
intercambia xs = [intercambiaCaracter x | x <- xs]

intercambiaCaracter :: Char -> Char
intercambiaCaracter c | isLower c = toUpper c
                      | otherwise = toLower c
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que, para cualquier cadena xs
-- se tiene que (intercambia (intercambia ys)) es igual a ys, siendo ys
-- la lista de las letras (mayúsculas o minúsculas no acentuadas) de xs.
-- ---------------------------------------------------------------------

prop_intercambia :: String -> Bool
prop_intercambia xs = intercambia (intercambia ys) == ys
    where ys = [x | x <- xs, x `elem` ['a'..'z'] ++ ['A'..'Z']]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    primosEquidistantes :: Integer -> [(Integer,Integer)]
-- tal que (primosEquidistantes n) es la lista de pares de primos
-- equidistantes de n y con la primera componente menor que la
-- segunda. Por ejemplo,
--    primosEquidistantes 8   ==  [(3,13),(5,11)]
--    primosEquidistantes 12  ==  [(5,19),(7,17),(11,13)]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
primosEquidistantes :: Integer -> [(Integer,Integer)]
primosEquidistantes n =
    [(x,n+(n-x)) | x <- [2..n-1], esPrimo x, esPrimo (n+(n-x))]

esPrimo :: Integer -> Bool
esPrimo n = [x | x <- [1..n], rem n x == 0] == [1,n]

-- 2ª definición (con zip):
primosEquidistantes2 :: Integer -> [(Integer,Integer)]
primosEquidistantes2 n =
    reverse [(x,y) | (x,y) <- zip [n-1,n-2..1] [n+1..], esPrimo x, esPrimo y]

-- Propiedad de equivalencia de las definiciones:
prop_equiv_primosEquidistantes :: Integer -> Property
prop_equiv_primosEquidistantes n =
    n > 0 ==> primosEquidistantes n == primosEquidistantes2 n

-- La comprobación es
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck si se cumple la siguiente
-- propiedad: "Todo número entero positivo mayor que 4 es equidistante
-- de dos primos"
-- ---------------------------------------------------------------------

-- La propiedad es
prop_suma2Primos :: Integer -> Property
prop_suma2Primos n =
    n > 4 ==> primosEquidistantes n /= []

-- La comprobación es
--    ghci> quickCheck prop_suma2Primos
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    triangulo :: [a] -> [(a,a)]
-- tal que (triangulo xs) es la lista de los pares formados por cada uno
-- de los elementos de xs junto con sus siguientes en xs. Por ejemplo,
--    ghci> triangulo [3,2,5,9,7]
--    [(3,2),(3,5),(3,9),(3,7),
--           (2,5),(2,9),(2,7),
--                 (5,9),(5,7),
--                       (9,7)]
-- ---------------------------------------------------------------------

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

-- 2ª solución
triangulo2 :: [a] -> [(a,a)]
triangulo2 []     = []
triangulo2 (x:xs) = zip (repeat x) xs ++ triangulo2 xs
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que la longitud de
-- (triangulo xs) es la suma desde 1 hasta n-1, donde n es el número de
-- elementos de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_triangulo :: [Int] -> Bool
prop_triangulo xs =
    length (triangulo xs) == sum [1..length xs - 1]

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

6.5.2. Examen 2 (3 de Diciembre de 2014)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (3 de diciembre de 2014)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    trenza :: [a] -> [a] -> [a]
-- tal que (trenza xs ys) es la lista obtenida intercalando los
-- elementos de xs e ys. Por ejemplo,
--    trenza [5,1] [2,7,4]             ==  [5,2,1,7]
--    trenza [5,1,7] [2..]             ==  [5,2,1,3,7,4]
--    trenza [2..] [5,1,7]             ==  [2,5,3,1,4,7]
--    take 8 (trenza [2,4..] [1,5..])  ==  [2,1,4,5,6,9,8,13]
-- ---------------------------------------------------------------------

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

-- 2ª definición (por zipWith):
trenza2 :: [a] -> [a] -> [a]
```

```
trenza2 xs ys = concat (zipWith par xs ys)
    where par x y = [x,y]

-- 3ª definición (por zipWith y sin argumentos):
trenza3 :: [a] -> [a] -> [a]
trenza3 = (concat .) . zipWith par
    where par x y = [x,y]

-- 4ª definición (por recursión):
trenza4 :: [a] -> [a] -> [a]
trenza4 (x:xs) (y:ys) = x : y : trenza xs ys
trenza4 _      _      = []

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que el número de elementos de
-- (trenza xs ys) es el doble del mínimo de los números de elementos de
-- xs e ys.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_trenza :: [Int] -> [Int] -> Bool
prop_trenza xs ys =
    length (trenza xs ys) == 2 * min (length xs) (length ys)

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Dado un número cualquiera, llamamos MDI de ese número
-- a su mayor divisor impar. Así, el MDI de 12 es 3 y el MDI de 15 es 15.
--
-- Definir la función
--    mdi :: Int -> Int
-- tal que (mdi n) es el mayor divisor impar de n. Por ejemplo,
--    mdi 12  ==  3
--    mdi 15  ==  15
-- ----------------------------------------------------------------------

mdi :: Int -> Int
```

```
mdi n | odd n     = n
      | otherwise = head [x | x <- [n-1,n-3..1], n `rem` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que la suma de los MDI de los
-- números n+1, n+2, ..., 2n de cualquier entero positivo n siempre da
-- n^2.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    ghci> quickCheckWith (stdArgs {maxSize=5}) prop_mdi
--    +++ OK, passed 100 tests.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mdi :: Int -> Property
prop_mdi n =
    n > 0 ==> sum [mdi x | x <- [n+1..2*n]] == n^2


prop_mdi2 :: Int -> Property
prop_mdi2 n =
    n > 0 ==> sum (map mdi [n+1..2*n]) == n^2

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=5}) prop_mdi
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    reiteracion :: Int -> (a -> a) -> a -> a
-- tal que (reiteracion n f x) es el resultado de aplicar n veces la
-- función f a x. Por ejemplo,
--    reiteracion 10 (+1) 5  ==  15
--    reiteracion 10 (+5) 0  ==  50
--    reiteracion  4 (*2) 1  ==  16
--    reiteracion  4 (5:) [] ==  [5,5,5,5]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
```

```
reiteracion :: Int -> (a -> a) -> a -> a
reiteracion 0 f x = x
reiteracion n f x = f (reiteracion (n-1) f x)

-- 2ª definición (por recursión sin el 3ª argumento):
reiteracion2 :: Int -> (a -> a) -> a -> a
reiteracion2 0 f = id
reiteracion2 n f = f . reiteracion2 (n-1) f

-- 3ª definición (con iterate):
reiteracion3 :: Int -> (a -> a) -> a -> a
reiteracion3 n f x = (iterate f x) !! n

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que se verifican las
-- siguientes propiedades
--    reiteracion 10 (+1) x  == 10 + x
--    reiteracion 10 (+x) 0  == 10 * x
--    reiteracion 10 (x:) [] == replicate 10 x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_reiteracion :: Int -> Bool
prop_reiteracion x =
    reiteracion 10 (+1) x  == 10 + x &&
    reiteracion 10 (+x) 0  == 10 * x &&
    reiteracion 10 (x:) [] == replicate 10 x

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la constante
--    cadenasDe0y1 :: [String]
-- tal que cadenasDe0y1 es la lista de todas las cadenas de ceros y
-- unos. Por ejemplo,
--    ghci> take 10 cadenasDe0y1
--    ["","0","1","00","10","01","11","000","100","010"]
-- ---------------------------------------------------------------------
```

```
cadenasDe0y1 :: [String]
cadenasDe0y1 = "" : concat [['0':cs, '1':cs] | cs <- cadenasDe0y1]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--     posicion :: String -> Int
-- tal que (posicion cs) es la posición de la cadena cs en la lista
-- cadenasDe0y1. Por ejemplo,
--     posicion "1"   == 2
--     posicion "010" == 9
-- ---------------------------------------------------------------------

posicion :: String -> Int
posicion cs =
    length (takeWhile (/= cs) cadenasDe0y1)

-- ---------------------------------------------------------------------
-- Ejercicio 5. El siguiente tipo de dato representa expresiones
-- construidas con números, variables, sumas y productos
--    data Expr = N Int
--              | V String
--              | S Expr Expr
--              | P Expr Expr
-- Por ejemplo, x*(5+z) se representa por (P (V "x") (S (N 5) (V "z")))
--
-- Definir la función
--    reducible :: Expr -> Bool
-- tal que (reducible a) se verifica si a es una expresión reducible; es
-- decir, contiene una operación en la que los dos operandos son números.
-- Por ejemplo,
--    reducible (S (N 3) (N 4))             == True
--    reducible (S (N 3) (V "x"))           == False
--    reducible (S (N 3) (P (N 4) (N 5)))   == True
--    reducible (S (V "x") (P (N 4) (N 5))) == True
--    reducible (S (N 3) (P (V "x") (N 5))) == False
--    reducible (N 3)                       == False
--    reducible (V "x")                     == False
-- ---------------------------------------------------------------------
```

```
data Expr = N Int
          | V String
          | S Expr Expr
          | P Expr Expr

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

6.5.3. Examen 3 (23 de enero de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 3º examen de evaluación continua (23 de enero de 2015)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    divisiblesPorAlguno :: [Int] -> [Int]
-- tal que (divisiblesPorAlguno xs) es la lista de los números que son
-- divisibles por algún elemento de xs. Por ejemplo,
--    take 10 (divisiblesPorAlguno [2,3])    ==  [2,3,4,6,8,9,10,12,14,15]
--    take 10 (divisiblesPorAlguno [2,4,3])  ==  [2,3,4,6,8,9,10,12,14,15]
--    take 10 (divisiblesPorAlguno [2,5,3])  ==  [2,3,4,5,6,8,9,10,12,14]
-- ---------------------------------------------------------------------

divisiblesPorAlguno :: [Int] -> [Int]
divisiblesPorAlguno xs = [n | n <- [1..], divisiblePorAlguno xs n]

-- 1ª definición (con any)
divisiblePorAlguno :: [Int] -> Int -> Bool
divisiblePorAlguno xs n = any (\x -> n `mod` x == 0) xs

-- 2ª definición (por comprensión)
divisiblePorAlguno1 :: [Int] -> Int -> Bool
```

```
divisiblePorAlguno1 xs n = or [n `mod` x == 0 | x <- xs]

-- 3ª definición (por recursión)
divisiblePorAlguno2 :: [Int] -> Int -> Bool
divisiblePorAlguno2 [] _     = False
divisiblePorAlguno2 (x:xs) n = n `mod` x == 0 || divisiblePorAlguno2 xs n

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las matrices pueden representarse mediante tablas cuyos
-- índices son pares de números naturales:
--    type Matriz = Array (Int,Int) Int
--
-- Definir la función
--    ampliada :: Matriz -> Matriz
-- tal que (ampliada p) es la matriz obtenida ampliando p añadiéndole
-- al final una columna con la suma de los elementos de cada fila y
-- añadiéndole al final una fila con la suma de los elementos de cada
-- columna. Por ejemplo, al ampliar las matrices
--    |1 2 3|      |1 2|
--    |4 5 6|      |3 4|
--                 |5 6|
-- se obtienen, respectivamente
--    |1 2 3  6|   |1  2  3|
--    |4 5 6 15|   |3  4  7|
--    |5 7 9 21|   |5  6 11|
--                 |9 12 21|
-- En Haskell,
--    ghci> ampliada (listArray ((1,1),(2,3)) [1,2,3, 4,5,6])
--    array ((1,1),(3,4)) [((1,1),1),((1,2),2),((1,3),3),((1,4),6),
--                         ((2,1),4),((2,2),5),((2,3),6),((2,4),15),
--                         ((3,1),5),((3,2),7),((3,3),9),((3,4),21)]
--    ghci> ampliada (listArray ((1,1),(3,2)) [1,2, 3,4, 5,6])
--    array ((1,1),(4,3)) [((1,1),1),((1,2),2),((1,3),3),
--                         ((2,1),3),((2,2),4),((2,3),7),
--                         ((3,1),5),((3,2),6),((3,3),11),
--                         ((4,1),9),((4,2),12),((4,3),21)]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int
```

```
ampliada :: Matriz -> Matriz
ampliada p = array ((1,1),(m+1,n+1))
                   [((i,j),f i j) | i <- [1..m+1], j <- [1..n+1]]
    where
      (_,(m,n)) = bounds p
      f i j | i <= m   && j <= n   = p ! (i,j)
            | i <= m   && j == n+1 = sum [p!(i,j) | j <- [1..n]]
            | i == m+1 && j <= n   = sum [p!(i,j) | i <- [1..m]]
            | i == m+1 && j == n+1 = sum [p!(i,j) | i <- [1..m], j <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. El siguiente tipo de dato representa expresiones
-- construidas con variables, sumas y productos
--    data Expr = Var String
--              | S Expr Expr
--              | P Expr Expre
--              deriving (Eq, Show)
-- Por ejemplo, x*(y+z) se representa por (P (V "x") (S (V "y") (V "z")))
--
-- Una expresión está en forma normal si es una suma de términos. Por
-- ejemplo, x*(y*z) y x+(y*z) está en forma normal; pero x*(y+z) y
-- (x+y)*(x+z) no lo están.
--
-- Definir la función
--    normal :: Expr -> Expr
-- tal que (normal e) es la forma normal de la expresión e obtenida
-- aplicando, mientras que sea posible, las propiedades distributivas:
--    (a+b)*c = a*c+b*c
--    c*(a+b) = c*a+c*b
-- Por ejemplo,
--    ghci> normal (P (S (V "x") (V "y")) (V "z"))
--    S (P (V "x") (V "z")) (P (V "y") (V "z"))
--    ghci> normal (P (V "z") (S (V "x") (V "y")))
--    S (P (V "z") (V "x")) (P (V "z") (V "y"))
--    ghci> normal (P (S (V "x") (V "y")) (S (V "u") (V "v")))
--    S (S (P (V "x") (V "u")) (P (V "x") (V "v")))
--      (S (P (V "y") (V "u")) (P (V "y") (V "v")))
--    ghci> normal (S (P (V "x") (V "y")) (V "z"))
--    S (P (V "x") (V "y")) (V "z")
--    ghci> normal (V "x")
```

```
--    V "x"
-- ---------------------------------------------------------------------

data Expr = V String
          | S Expr Expr
          | P Expr Expr
          deriving (Eq, Show)

normal :: Expr -> Expr
normal (V v)   = V v
normal (S a b) = S (normal a) (normal b)
normal (P a b) = p (normal a) (normal b)
    where p (S a b) c = S (p a c) (p b c)
          p a (S b c) = S (p a b) (p a c)
          p a b       = P a b

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los primeros números de Fibonacci son
--    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
-- tales que los dos primeros son iguales a 1 y los siguientes se
-- obtienen sumando los dos anteriores.
--
-- El teorema de Zeckendorf establece que todo entero positivo n se
-- puede representar, de manera única, como la suma de números de
-- Fibonacci no consecutivos decrecientes. Dicha suma se llama la
-- representación de Zeckendorf de n. Por ejemplo, la representación de
-- Zeckendorf de 100 es
--    100 = 89 + 8 + 3
-- Hay otras formas de representar 100 como sumas de números de
-- Fibonacci; por ejemplo,
--    100 = 89 +  8 + 2 + 1
--    100 = 55 + 34 + 8 + 3
-- pero no son representaciones de Zeckendorf porque 1 y 2 son números
-- de Fibonacci consecutivos, al igual que 34 y 55.
--
-- Definir la función
--    zeckendorf :: Integer -> [Integer]
-- tal que (zeckendorf n) es la representación de Zeckendorf de n. Por
-- ejemplo,
--    zeckendorf 100       == [89,8,3]
```

```
--    zeckendorf 2014      == [1597,377,34,5,1]
--    zeckendorf 28656     == [17711,6765,2584,987,377,144,55,21,8,3,1]
--    zeckendorf 14930396  == [14930352,34,8,2]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========
zeckendorf1 :: Integer -> [Integer]
zeckendorf1 n = reverse (head (aux n (tail fibs)))
    where aux 0 _ = [[]]
          aux n (x:y:zs)
              | x <= n     = [x:xs | xs <- aux (n-x) zs] ++ aux n (y:zs)
              | otherwise  = []

-- fibs es la sucesión de los números de Fibonacci. Por ejemplo,
--    take 14 fibs  == [1,1,2,3,5,8,13,21,34,55,89,144,233,377]
fibs :: [Integer]
fibs = 1 : scanl (+) 1 fibs

-- 2ª solución
-- ===========
zeckendorf2 :: Integer -> [Integer]
zeckendorf2 n = aux n (reverse (takeWhile (<= n) fibs))
    where aux 0 _ = []
          aux n (x:xs) = x : aux (n-x) (dropWhile (>n-x) xs)

-- 3ª solución
-- ===========
zeckendorf3 :: Integer -> [Integer]
zeckendorf3 0 = []
zeckendorf3 n = x : zeckendorf3 (n - x)
    where x = last (takeWhile (<= n) fibs)

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

-- La comparación es
--    ghci> zeckendorf1 300000
--    [196418,75025,17711,6765,2584,987,377,89,34,8,2]
--    (0.72 secs, 58478576 bytes)
```

```
--    ghci> zeckendorf2 300000
--    [196418,75025,17711,6765,2584,987,377,89,34,8,2]
--    (0.00 secs, 517852 bytes)
--    ghci> zeckendorf3 300000
--    [196418,75025,17711,6765,2584,987,377,89,34,8,2]
--    (0.00 secs, 515360 bytes)
-- Se observa que las definiciones más eficientes son la 2ª y la 3ª.

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    maximoIntercambio :: Int -> Int
-- tal que (maximoIntercambio x) es el máximo número que se puede
-- obtener intercambiando dos dígitos de x. Por ejemplo,
--    maximoIntercambio 983562  ==  986532
--    maximoIntercambio 31524   ==  51324
--    maximoIntercambio 897     ==  987
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============
maximoIntercambio :: Int -> Int
maximoIntercambio = maximum . intercambios

-- (intercambios x) es la lista de los números obtenidos intercambiando
-- dos dígitos de x. Por ejemplo,
--    intercambios 1234  ==  [2134,3214,4231,1324,1432,1243]
intercambios :: Int -> [Int]
intercambios x = [intercambio i j x | i <- [0..n-2], j <- [i+1..n-1]]
    where n = length (show x)

-- (intercambio i j x) es el número obtenido intercambiando las cifras
-- que ocupan las posiciones i y j (empezando a contar en cero) del
-- número x. Por ejemplo,
--    intercambio 2 5 123456789  ==  126453789
intercambio :: Int -> Int -> Int -> Int
intercambio i j x = read (concat [as,[d],cs,[b],ds])
    where xs        = show x
          (as,b:bs) = splitAt i xs
          (cs,d:ds) = splitAt (j-i-1) bs
```

```
-- 2ª definición (con vectores)
-- ===========================

maximoIntercambio2 :: Int -> Int
maximoIntercambio2 = read . elems . maximum . intercambios2

-- (intercambios2 x) es la lista de los vectores obtenidos
-- intercambiando dos elementos del vector de dígitos de x. Por ejemplo,
--    ghci> intercambios2 1234
--    [array (0,3) [(0,'2'),(1,'1'),(2,'3'),(3,'4')],
--     array (0,3) [(0,'3'),(1,'2'),(2,'1'),(3,'4')],
--     array (0,3) [(0,'4'),(1,'2'),(2,'3'),(3,'1')],
--     array (0,3) [(0,'1'),(1,'3'),(2,'2'),(3,'4')],
--     array (0,3) [(0,'1'),(1,'4'),(2,'3'),(3,'2')],
--     array (0,3) [(0,'1'),(1,'2'),(2,'4'),(3,'3')]]
intercambios2 :: Int -> [Array Int Char]
intercambios2 x = [intercambioV i j v | i <- [0..n-2], j <- [i+1..n-1]]
    where xs = show x
          n  = length xs
          v  = listArray (0,n-1) xs

-- (intercambioV i j v) es el vector obtenido intercambiando los
-- elementos de v que ocupan las posiciones i y j. Por ejemplo,
--    ghci> intercambioV 2 4 (listArray (0,4) [3..8])
--    array (0,4) [(0,3),(1,4),(2,7),(3,6),(4,5)]
intercambioV i j v = v // [(i,v!j),(j,v!i)]
```

6.5.4. Examen 4 (9 de marzo de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (9 de marzo de 2015)
-- ---------------------------------------------------------------------

import I1M.PolOperaciones
import Data.List
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un capicúa es un número que es igual leído de izquierda
-- a derecha que de derecha a izquierda.
```

```
--
-- Definir la función
--    mayorCapicuaP :: Integer -> Integer
-- tal que (mayorCapicuaP n) es el mayor capicúa que es el producto de
-- dos números de n cifras. Por ejemplo,
--    mayorCapicuaP 2  ==  9009
--    mayorCapicuaP 3  ==  906609
--    mayorCapicuaP 4  ==  99000099
--    mayorCapicuaP 5  ==  9966006699
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

mayorCapicuaP1 :: Integer -> Integer
mayorCapicuaP1 n = maximum [x*y | x <- [a,a-1..b],
                                  y <- [a,a-1..b],
                                  esCapicua (x*y)]
    where a = 10^n-1
          b = 10^(n-1)

-- (esCapicua x) se verifica si x es capicúa. Por ejemplo,
--    esCapicua 353  ==  True
--    esCapicua 357  ==  False
esCapicua :: Integer -> Bool
esCapicua n = xs == reverse xs
    where xs = show n

-- 2ª solución
-- ===========

mayorCapicuaP2 :: Integer -> Integer
mayorCapicuaP2 n = maximum [x | y <- [a..b],
                                z <- [y..b],
                                let x = y * z,
                                let s = show x,
                                s == reverse s]
     where a = 10^(n-1)
           b = 10^n-1
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Sea (b(i) | i ≥ 1) una sucesión infinita de números
-- enteros mayores que 1. Entonces todo entero x mayor que cero se puede
-- escribir de forma única como
--    x = x(0) + x(1)b(1) +x(2)b(1)b(2) + ... + x(n)b(1)b(2)...b(n)
-- donde cada x(i) satisface la condición 0 ≤ x(i) < b(i+1). Se dice
-- que [x(n),x(n-1),...,x(2),x(1),x(0)] es la representación de x en la
-- base (b(i)). Por ejemplo, la representación de 377 en la base
-- (2*i | i >= 1) es [7,5,0,1] ya que
--    377 = 1 + 0*2 + 5*2*4 + 7*2*4*6
-- y, además, 0 ≤ 1 < 2, 0 ≤ 0 < 4, 0 ≤ 5 < 6 y 0 ≤ 7 < 8.
--
-- Definir las funciones
--    decimalAmultiple :: [Integer] -> Integer -> [Integer]
--    multipleAdecimal :: [Integer] -> [Integer] -> Integer
-- tales que (decimalAmultiple bs x) es la representación del número x
-- en la base bs y (multipleAdecimal bs cs) es el número decimal cuya
-- representación en la base bs es cs. Por ejemplo,
--    decimalAmultiple [2,4..] 377                      ==  [7,5,0,1]
--    multipleAdecimal [2,4..] [7,5,0,1]                ==  377
--    decimalAmultiple [2,5..] 377                      ==  [4,5,3,1]
--    multipleAdecimal [2,5..] [4,5,3,1]                ==  377
--    decimalAmultiple [2^n | n <- [1..]] 2015          ==  [1,15,3,3,1]
--    multipleAdecimal [2^n | n <- [1..]] [1,15,3,3,1]  ==  2015
--    decimalAmultiple (repeat 10) 2015                 ==  [2,0,1,5]
--    multipleAdecimal (repeat 10) [2,0,1,5]            ==  2015
-- ---------------------------------------------------------------------

-- 1ª definición de decimalAmultiple (por recursión)
decimalAmultiple :: [Integer] -> Integer -> [Integer]
decimalAmultiple bs n = reverse (aux bs n)
    where aux _ 0 = []
          aux (b:bs) n = r : aux bs q
              where (q,r) = quotRem n b

-- 2ª definición de decimalAmultiple (con acumulador)
decimalAmultiple2 :: [Integer] -> Integer -> [Integer]
decimalAmultiple2 bs n = aux bs n []
    where aux _ 0  xs = xs
          aux (b:bs) n xs = aux bs q (r:xs)
```

```
             where (q,r) = quotRem n b

-- 1ª definición multipleAdecimal (por recursión)
multipleAdecimal :: [Integer] -> [Integer] -> Integer
multipleAdecimal xs ns = aux xs (reverse ns)
    where aux (x:xs) (n:ns) = n + x * (aux xs ns)
          aux _ _           = 0

-- 2ª definición multipleAdecimal (con scanl1)
multipleAdecimal2 :: [Integer] -> [Integer] -> Integer
multipleAdecimal2 bs xs =
    sum (zipWith (*) (reverse xs) (1 : scanl1 (*) bs))

-- ---------------------------------------------------------------------
-- Ejercicio 3. Las expresiones aritméticas pueden representarse usando
-- el siguiente tipo de datos
--    data Expr = N Int | S Expr Expr | P Expr Expr
--                deriving (Eq, Show)
-- Por ejemplo, la expresión 2*(3+7) se representa por
--    P (N 2) (S (N 3) (N 7))
--
-- Definir la función
--    subexpresiones :: Expr -> [Expr]
-- tal que (subexpresiones e) es el conjunto de las subexpresiones de
-- e. Por ejemplo,
--    ghci> subexpresiones (S (N 2) (N 3))
--    [S (N 2) (N 3),N 2,N 3]
--    ghci> subexpresiones (P (S (N 2) (N 2)) (N 7))
--    [P (S (N 2) (N 2)) (N 7),S (N 2) (N 2),N 2,N 7]
-- ---------------------------------------------------------------------

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

subexpresiones :: Expr -> [Expr]
subexpresiones = nub . aux
    where aux (N x) = [N x]
          aux (S i d) = S i d : (subexpresiones i ++ subexpresiones d)
          aux (P i d) = P i d : (subexpresiones i ++ subexpresiones d)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    diagonalesPrincipales :: Array (Int,Int) a -> [[a]]
-- tal que (diagonalesPrincipales p) es la lista de las diagonales
-- principales de p. Por ejemplo, para la matriz
--    1  2  3  4
--    5  6  7  8
--    9 10 11 12
-- la lista de sus diagonales principales es
--    [[9],[5,10],[1,6,11],[2,7,12],[3,8],[4]]
-- En Haskell,
--    ghci> diagonalesPrincipales (listArray ((1,1),(3,4)) [1..12])
--    [[9],[5,10],[1,6,11],[2,7,12],[3,8],[4]]
-- ---------------------------------------------------------------------

diagonalesPrincipales :: Array (Int,Int) a -> [[a]]
diagonalesPrincipales p =
    [[p!ij1 | ij1 <- extension ij] | ij <- iniciales]
    where (_,(m,n)) = bounds p
          iniciales = [(i,1) | i <- [m,m-1..2]] ++ [(1,j) | j <- [1..n]]
          extension (i,j) = [(i+k,j+k) | k <- [0..min (m-i) (n-j)]]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Dado un polinomio p no nulo con coeficientes enteros, se
-- llama contenido de p al máximo común divisor de sus coeficientes. Se
-- dirá que p es primitivo si su contenido es 1.
--
-- Definir la función
--    primitivo :: Polinomio Int -> Bool
-- tal que (primitivo p) se verifica si el polinomio p es primitivo. Por
-- ejemplo,
--    ghci> let listaApol xs = foldr (\(n,b) -> consPol n b) polCero xs
--    ghci> primitivo (listaApol [(6,2),(4,3)])
--    True
--    ghci> primitivo (listaApol [(6,2),(5,3),(4,8)])
--    True
--    ghci> primitivo (listaApol [(6,2),(5,6),(4,8)])
--    False
-- ---------------------------------------------------------------------
```

```
primitivo :: Polinomio Int -> Bool
primitivo p = contenido p == 1

contenido :: Polinomio Int -> Int
contenido p
    | n == 0 = b
    | otherwise = gcd b (contenido r)
    where n = grado p
          b = coefLider p
          r = restoPol p
```

6.5.5. Examen 5 (29 de abril de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (29 de abril de 2015)
-- ---------------------------------------------------------------------

import Data.Array
import Data.List
import Data.Numbers.Primes
import I1M.Grafo
import I1M.Monticulo
import I1M.PolOperaciones
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una propiedad del 2015 es que la suma de sus dígitos
-- coincide con el número de sus divisores; en efecto, la suma de sus
-- dígitos es 2+0+1+5=8 y tiene 8 divisores (1, 5, 13, 31, 65, 155, 403
-- y 2015).
--
-- Definir la sucesión
--    especiales :: [Int]
-- formada por los números n tales que la suma de los dígitos de n
-- coincide con el número de divisores de n. Por ejemplo,
--    take 12 especiales == [1,2,11,22,36,84,101,152,156,170,202,208]
--
-- Calcular la posición de 2015 en la sucesión de especiales.
-- ---------------------------------------------------------------------
```

```
especiales :: [Int]
especiales = [n | n <- [1..], sum (digitos n) == length (divisores n)]

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

divisores :: Int -> [Int]
divisores n = n : [x | x <- [1..n `div` 2], n `mod` x == 0]

-- El cálculo de número de años hasta el 2015 inclusive que han cumplido
-- la propiedad es
--    ghci> length (takeWhile (<=2015) especiales)
--    59

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    posicion :: Array Int Bool -> Maybe Int
-- tal que (posicion v) es la menor posición del vector de booleanos v
-- cuyo valor es falso y es Nothing si todos los valores son
-- verdaderos. Por ejemplo,
--    posicion (listArray (0,4) [True,True,False,True,False]) == Just 2
--    posicion (listArray (0,4) [i <= 2 | i <- [0..4]])       == Just 3
--    posicion (listArray (0,4) [i <= 7 | i <- [0..4]])       == Nothing
-- ---------------------------------------------------------------------

-- 1ª solución
posicion :: Array Int Bool -> Maybe Int
posicion v | p > n     = Nothing
           | otherwise = Just p
    where p = (length . takeWhile id . elems) v
          (_,n) = bounds v

-- 2ª solución:
posicion2 :: Array Int Bool -> Maybe Int
posicion2 v | null xs   = Nothing
            | otherwise = Just (head xs)
    where xs = [i | i <- indices v, v!i]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
```

```
--    todos :: Ord a => (a -> Bool) -> Monticulo a -> Bool
-- tal que (todos p m) se verifica si todos los elementos del montículo
-- m cumple la propiedad p, Por ejemplo,
--    todos (>2) (foldr inserta vacio [6,3,4,8])  ==  True
--    todos even (foldr inserta vacio [6,3,4,8])  ==  False
-- ---------------------------------------------------------------------

todos :: Ord a => (a -> Bool) -> Monticulo a -> Bool
todos p m
    | esVacio m = True
    | otherwise = p (menor m) && todos p (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 4. El complementario del grafo G es un grafo G' del mismo
-- tipo que G (dirigido o no dirigido), con el mismo conjunto de nodos y
-- tal que dos nodos de G' son adyacentes si y sólo si no son adyacentes
-- en G. Los pesos de todas las aristas del complementario es igual a 0.
--    ghci> complementario (creaGrafo D (1,3) [(1,3,0),(3,2,0),(2,2,0),(2,1,0)])
--    G D (array (1,3) [(1,[(1,0),(2,0)]),(2,[(3,0)]),(3,[(1,0),(3,0)])])
--    ghci> complementario (creaGrafo D (1,3) [(3,2,0),(2,2,0),(2,1,0)])
--    G D (array (1,3) [(1,[(1,0),(2,0),(3,0)]),(2,[(3,0)]),(3,[(1,0),(3,0)])])
-- ---------------------------------------------------------------------

complementario :: Grafo Int Int -> Grafo Int Int
complementario g =
    creaGrafo d (1,n) [(x,y,0) | x <- xs, y <- xs, not (aristaEn g (x,y))]
    where d  = if dirigido g then D else ND
          xs = nodos g
          n  = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 5. En 1772, Euler publicó que el polinomio n^2 + n + 41
-- genera 40 números primos para todos los valores de n entre 0 y
-- 39. Sin embargo, cuando n=40, 40^2+40+41 = 40(40+1)+41 es divisible
-- por 41.
--
-- Definir la función
--    generadoresMaximales :: Integer -> (Int,[(Integer,Integer)])
-- tal que (generadoresMaximales n) es el par (m,xs) donde
--    + xs es la lista de pares (x,y) tales que n^2+xn+y es uno de los
```

```
--      polinomios que genera un número máximo de números primos
--      consecutivos a partir de cero entre todos los polinomios de la
--      forma n²+an+b, con |a| ≤ n y |b| ≤ n y
--    + m es dicho número máximo.
-- Por ejemplo,
--    generadoresMaximales    4  ==  ( 3,[(-2,3),(-1,3),(3,3)])
--    generadoresMaximales    6  ==  ( 5,[(-1,5),(5,5)])
--    generadoresMaximales   50  ==  (43,[(-5,47)])
--    generadoresMaximales  100  ==  (48,[(-15,97)])
--    generadoresMaximales  200  ==  (53,[(-25,197)])
--    generadoresMaximales 1650  ==  (80,[(-79,1601)])
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

generadoresMaximales1 :: Integer -> (Int,[(Integer,Integer)])
generadoresMaximales1 n =
    (m,[((a,b)) | a <- [-n..n], b <- [-n..n], nPrimos a b == m])
    where m = maximum $ [nPrimos a b | a <- [-n..n], b <- [-n..n]]

-- (nPrimos a b) es el número de primos consecutivos generados por el
-- polinomio n² + an + b a partir de n=0. Por ejemplo,
--    nPrimos 1 41        ==  40
--    nPrimos (-79) 1601  ==  80
nPrimos :: Integer -> Integer -> Int
nPrimos a b =
    length $ takeWhile isPrime [n*n+a*n+b | n <- [0..]]

-- 2ª solución (reduciendo las cotas)
-- ==================================

-- Notas:
-- 1. Se tiene que b es primo, ya que para n=0, se tiene que 0²+a*0+b =
--    b es primo.
-- 2. Se tiene que 1+a+b es primo, ya que es el valor del polinomio para
--    n=1.

generadoresMaximales2 :: Integer -> (Int,[(Integer,Integer)])
generadoresMaximales2 n = (m,map snd zs)
```

```
    where xs = [(nPrimos a b,(a,b)) | b <- takeWhile (<=n) primes,
                                      a <- [-n..n],
                                      isPrime(1+a+b)]
          ys = reverse (sort xs)
          m  = fst (head ys)
          zs = takeWhile (\(k,_) -> k == m) ys

-- 3ª solución (con la librería de polinomios)
-- ==============================================

generadoresMaximales3 :: Integer -> (Int,[(Integer,Integer)])
generadoresMaximales3 n = (m,map snd zs)
    where xs = [(nPrimos2 a b,(a,b)) | b <- takeWhile (<=n) primes,
                                       a <- [-n..n],
                                       isPrime(1+a+b)]
          ys = reverse (sort xs)
          m  = fst (head ys)
          zs = takeWhile (\(k,_) -> k == m) ys

-- (nPrimos2 a b) es el número de primos consecutivos generados por el
-- polinomio n^2 + an + b a partir de n=0. Por ejemplo,
--    nPrimos2 1 41        ==  40
--    nPrimos2 (-79) 1601  ==  80
nPrimos2 :: Integer -> Integer -> Int
nPrimos2 a b =
    length $ takeWhile isPrime [valor p n | n <- [0..]]
    where p = consPol 2 1 (consPol 1 a (consPol 0 b polCero))

-- Comparación de eficiencia
--    ghci> generadoresMaximales1 200
--    (53,[(-25,197)])
--    (3.06 secs, 720683776 bytes)
--    ghci> generadoresMaximales1 300
--    (56,[(-31,281)])
--    (6.65 secs, 1649274220 bytes)
--
--    ghci> generadoresMaximales2 200
--    (53,[(-25,197)])
--    (0.25 secs, 94783464 bytes)
--    ghci> generadoresMaximales2 300
```

```
--    (56,[(-31,281)])
--    (0.51 secs, 194776708 bytes)
--
--    ghci> generadoresMaximales3 200
--    (53,[(-25,197)])
--    (0.20 secs, 105941096 bytes)
--    ghci> generadoresMaximales3 300
--    (56,[(-31,281)])
--    (0.35 secs, 194858344 bytes)
```

6.5.6. Examen 6 (15 de junio de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (15 de junio de 2015)
-- ---------------------------------------------------------------------

import Data.List
import qualified Data.Map as M
import I1M.BusquedaEnEspaciosDeEstados
import I1M.Grafo

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una inversión de una lista xs es un par de elementos
-- (x,y) de xs tal que y está a la derecha de x en xs y además y es
-- menor que x. Por ejemplo, en la lista [1,7,4,9,5] hay tres
-- inversiones: (7,4), (7,5) y (9,5).
--
-- Definir la función
--    inversiones :: Ord a -> [a] -> [(a,a)]
-- tal que (inversiones xs) es la lista de las inversiones de xs. Por
-- ejemplo,
--    inversiones [1,7,4,9,5]  ==  [(7,4),(7,5),(9,5)]
--    inversiones "esto"       ==  [('s','o'),('t','o')]
-- ---------------------------------------------------------------------

inversiones :: Ord a => [a] -> [(a,a)]
inversiones []     = []
inversiones (x:xs) = [(x,y) | y <- xs, y < x] ++ inversiones xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las expresiones aritméticas se pueden representar como
```

```
-- árboles con números en las hojas y operaciones en los nodos. Por
-- ejemplo, la expresión "9-2*4" se puede representar por el árbol
--       -
--      / \
--     9   *
--        / \
--       2   4
--
-- Definiendo el tipo de dato Arbol por
--     data Arbol = H Int | N (Int -> Int -> Int) Arbol Arbol
-- la representación del árbol anterior es
--     N (-) (H 9) (N (*) (H 2) (H 4))
--
-- Definir la función
--     valor :: Arbol -> Int
-- tal que (valor a) es el valor de la expresión aritmética
-- correspondiente al árbol a. Por ejemplo,
--     valor (N (-) (H 9) (N (*) (H 2) (H 4)))    ==  1
--     valor (N (+) (H 9) (N (*) (H 2) (H 4)))    ==  17
--     valor (N (+) (H 9) (N (div) (H 4) (H 2)))  ==  11
--     valor (N (+) (H 9) (N (max) (H 4) (H 2)))  ==  13
-- ---------------------------------------------------------------------

data Arbol = H Int | N (Int -> Int -> Int) Arbol Arbol

valor :: Arbol -> Int
valor (H x)     = x
valor (N f i d) = f (valor i) (valor d)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--     agrupa :: Ord c => (a -> c) -> [a] -> M.Map c [a]
-- tal que (agrupa f xs) es el diccionario obtenido agrupando los
-- elementos de xs según sus valores mediante la función f. Por ejemplo,
--     ghci> agrupa length ["hoy", "ayer", "ana", "cosa"]
--     fromList [(3,["hoy","ana"]),(4,["ayer","cosa"])]
--     ghci> agrupa head ["claro", "ayer", "ana", "cosa"]
--     fromList [('a',["ayer","ana"]),('c',["claro","cosa"])]
--     ghci> agrupa length (words "suerte en el examen")
--     fromList [(2,["en","el"]),(6,["suerte","examen"])]
```

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

-- 1ª definición (por recursión)
agrupa1 :: Ord c => (a -> c) -> [a] -> M.Map c [a]
agrupa1 _ []     = M.empty
agrupa1 f (x:xs) = M.insertWith (++) (f x) [x] (agrupa1 f xs)

-- 2ª definición (por plegado)
agrupa2 :: Ord c => (a -> c) -> [a] -> M.Map c [a]
agrupa2 f = foldr (\x -> M.insertWith (++) (f x) [x]) M.empty

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los primeros términos de la sucesión de Fibonacci son
--    0, 1, 1, 2, 3, 5, 8, 13, 21, 34
-- Se observa que el 6º término de la sucesión (comenzando a contar en
-- 0) es el número 8.
--
-- Definir la función
--    indiceFib :: Integer -> Maybe Integer
-- tal que (indiceFib x) es justo el número n si x es el n-ésimo
-- términos de la sucesión de Fibonacci o Nothing en el caso de que x no
-- pertenezca a la sucesión. Por ejemplo,
--    indiceFib 8        ==  Just 6
--    indiceFib 9        ==  Nothing
--    indiceFib 21       ==  Just 8
--    indiceFib 22       ==  Nothing
--    indiceFib 9227465  ==  Just 35
--    indiceFib 9227466  ==  Nothing
-- ---------------------------------------------------------------------

indiceFib :: Integer -> Maybe Integer
indiceFib x | y == x    = Just n
            | otherwise = Nothing
    where (y,n) = head (dropWhile (\(z,m) -> z < x) fibsNumerados)

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

```
-- fibsNumerados es la lista de los términos de la sucesión de Fibonacci
-- juntos con sus posiciones. Por ejemplo,
--    ghci> take 10 fibsNumerados
--    [(0,0),(1,1),(1,2),(2,3),(3,4),(5,5),(8,6),(13,7),(21,8),(34,9)]
fibsNumerados :: [(Integer,Integer)]
fibsNumerados = zip fibs [0..]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir las funciones
--    grafo   :: [(Int,Int)] -> Grafo Int Int
--    caminos :: Grafo Int Int -> Int -> Int -> [[Int]]
-- tales que
-- + (grafo as) es el grafo no dirigido definido cuyas aristas son as. Por
--   ejemplo,
--      ghci> grafo [(2,4),(4,5)]
--      G ND (array (2,5) [(2,[(4,0)]),(3,[]),(4,[(2,0),(5,0)]),(5,[(4,0)])])
-- + (caminos g a b) es la lista los caminos en el grafo g desde a hasta
--   b sin pasar dos veces por el mismo nodo. Por ejemplo,
--      ghci> sort (caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 7)
--      [[1,3,5,7],[1,3,7]]
--      ghci> sort (caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 2 7)
--      [[2,5,3,7],[2,5,7]]
--      ghci> sort (caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 2)
--      [[1,3,5,2],[1,3,7,5,2]]
--      ghci> caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 4
--      []
-- ---------------------------------------------------------------------

grafo :: [(Int,Int)] -> Grafo Int Int
grafo as = creaGrafo ND (m,n) [(x,y,0) | (x,y) <- as]
    where ns = map fst as ++ map snd as
          m  = minimum ns
          n  = maximum ns

-- 1ª solución (mediante espacio de estados)
caminos1 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminos1 g a b = buscaEE sucesores esFinal inicial
    where inicial          = [b]
          sucesores (x:xs) = [z:x:xs | z <- adyacentes g x
```

```
                                , z `notElem` (x:xs)]
          esFinal (x:xs)   = x == a

-- 2ª solución (sin espacio de estados)
caminos2 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminos2 g a b = aux [[b]] where
    aux [] = []
    aux ((x:xs):yss)
        | x == a    = (x:xs) : aux yss
        | otherwise = aux ([z:x:xs | z <- adyacentes g x
                                  , z `notElem` (x:xs)]
                           ++ yss)

-- ============================================================================

-- Tipos de ejercicios
--    |--------------------------------+----+-----+----+----+----|
--    |                                | E1 | E2  | E3 | E4 | E5 |
--    |--------------------------------+----+-----+----+----+----|
--    | R: Recursión                   | R  | R   | R  | R  | R  |
--    | C: Comprensión                 | C  |     |    |    |    |
--    | TDA: Tipo de datos algebraicos |    | TDA |    |    |    |
--    | OS: Orden superior             |    |     | OS |    | OS |
--    | D: Diccionarios                |    |     | D  |    |    |
--    | P: Plegado                     |    |     | P  |    |    |
--    | M: Maybe                       |    |     |    | M  |    |
--    | LI: Listas infinitas           |    |     |    | LI |    |
--    | PD: Programación dinámica      |    |     |    | PD |    |
--    | E: Emparejamiento con zip      |    |     |    | E  |    |
--    | G: Grafos                      |    |     |    |    | G  |
--    | EE: Espacio de estados         |    |     |    |    | EE |
--    |--------------------------------+----+-----+----+----+----|
```

6.5.7. Examen 7 (3 de julio de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de julio (3 de julio de 2015)
-- ---------------------------------------------------------------------

import Data.List
```

```
import Data.Matrix
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    minimales :: Eq a => [[a]] -> [[a]]
-- tal que (minimales xss) es la lista de los elementos de xss que no
-- están contenidos en otros elementos de xss. Por ejemplo,
--    minimales [[1,3],[2,3,1],[3,2,5]]        ==  [[2,3,1],[3,2,5]]
--    minimales [[1,3],[2,3,1],[3,2,5],[3,1]]  ==  [[2,3,1],[3,2,5]]
-- ---------------------------------------------------------------------

minimales :: Eq a => [[a]] -> [[a]]
minimales xss =
    [xs | xs <- xss, null [ys | ys <- xss, subconjuntoPropio xs ys]]

-- (subconjuntoPropio xs ys) se verifica si xs es un subconjunto propio
-- de ys. Por ejemplo,
--    subconjuntoPropio [1,3] [3,1,3]    ==  False
--    subconjuntoPropio [1,3,1] [3,1,2]  ==  True
subconjuntoPropio :: Eq a => [a] -> [a] -> Bool
subconjuntoPropio xs ys = subconjuntoPropio' (nub xs) (nub ys)
    where
      subconjuntoPropio' _xs [] = False
      subconjuntoPropio' [] _ys = True
      subconjuntoPropio' (x:xs) ys =
          x `elem` ys && subconjuntoPropio xs (delete x ys)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un mínimo local de una lista es un elemento de la lista
-- que es menor que su predecesor y que su sucesor en la lista. Por
-- ejemplo, 1 es un mínimo local de [8,2,1,3,7,6,4,0,5] ya que es menor
-- que 2 (su predecesor) y que 3 (su sucesor).
--
-- Análogamente se definen los máximos locales. Por ejemplo, 7 es un
-- máximo local de [8,2,1,3,7,6,4,0,5] ya que es mayor que 7 (su
-- predecesor) y que 6 (su sucesor).
--
-- Los extremos locales están formados por los mínimos y máximos
-- locales. Por ejemplo, los extremos locales de [8,2,1,3,7,6,4,0,5] son
```

```
-- el 1, el 7 y el 0.
--
-- Definir la función
--    extremos :: Ord a => [a] -> [a]
-- tal que (extremos xs) es la lista de los extremos locales de la
-- lista xs. Por ejemplo,
--    extremos [8,2,1,3,7,6,4,0,5]  ==  [1,7,0]
--    extremos [8,2,1,3,7,7,4,0,5]  ==  [1,7,0]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
-- ===============================
extremos1 :: Ord a => [a] -> [a]
extremos1 xs =
    [y | (x,y,z) <- zip3 xs (tail xs) (drop 2 xs), extremo x y z]

-- (extremo x y z) se verifica si y es un extremo local de [x,y,z]. Por
-- ejemplo,
--    extremo 2 1 3  ==  True
--    extremo 3 7 6  ==  True
--    extremo 7 6 4  ==  False
--    extremo 5 6 7  ==  False
--    extremo 5 5 7  ==  False
extremo :: Ord a => a -> a -> a -> Bool
extremo x y z = (y < x && y < z) || (y > x && y > z)

-- 2ª definición (por recursión)
-- =============================

extremos2 :: Ord a => [a] -> [a]
extremos2 (x:y:z:xs)
    | extremo x y z = y : extremos2 (y:z:xs)
    | otherwise     = extremos2 (y:z:xs)
extremos2 _ = []

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles, con un número variable de hijos, se pueden
-- representar mediante el siguiente tipo de dato
--    data Arbol a = N a [Arbol a]
--                   deriving Show
```

```
-- Por ejemplo, los árboles
--      1              3
--     / \            /|\
--    6   3          / | \
--        |         5  4  7
--        5         |     /\
--                  6    2  1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    ej1 = N 1 [N 6 [],N 3 [N 5 []]]
--    ej2 = N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]]
--
-- Definir la función
--    emparejaArboles :: (a -> b -> c) -> Arbol a -> Arbol b -> Arbol c
-- tal que (emparejaArboles f a1 a2) es el árbol obtenido aplicando la
-- función f a los elementos de los árboles a1 y a2 que se encuentran en
-- la misma posición. Por ejemplo,
--    ghci> emparejaArboles (+) (N 1 [N 2 [], N 3[]]) (N 1 [N 6 []])
--    N 2 [N 8 []]
--    ghci> emparejaArboles (+) ej1 ej2
--    N 4 [N 11 [],N 7 []]
--    ghci> emparejaArboles (+) ej1 ej1
--    N 2 [N 12 [],N 6 [N 10 []]]
-- ---------------------------------------------------------------------

data Arbol a = N a [Arbol a]
               deriving (Show, Eq)

emparejaArboles :: (a -> b -> c) -> Arbol a -> Arbol b -> Arbol c
emparejaArboles f (N x l1) (N y l2) =
    N (f x y) (zipWith (emparejaArboles f) l1 l2)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la lista
--    antecesoresYsucesores :: [[Integer]]
-- cuyos elementos son
--    [[1],[0,2],[-1,1,1,3],[-2,2,0,0,2,0,2,2,4],...]
-- donde cada una de las listas se obtiene de la anterior sustituyendo
-- cada elemento por su antecesor y su sucesor; es decir, el 1 por el 0
-- y el 2, el 0 por el -1 y el 1, el 2 por el 1 y el 3, etc. Por
```

```
-- ejemplo,
--    ghci> take 4 antecesoresYsucesores
--    [[1],[0,2],[-1,1,1,3],[-2,0,0,2,0,2,2,4]]
--
-- Comprobar con Quickcheck que la suma de los elementos de la lista
-- n-ésima de antecesoresYsucesores es 2^n.
--
-- Nota. Limitar la búsqueda a ejemplos pequeños usando
--    quickCheckWith (stdArgs {maxSize=7}) prop_suma
-- ---------------------------------------------------------------------

-- 1ª solución
antecesoresYsucesores :: [[Integer]]
antecesoresYsucesores =
    [1] : map (concatMap (\x -> [x-1,x+1])) antecesoresYsucesores

-- 2ª solución
antecesoresYsucesores2 :: [[Integer]]
antecesoresYsucesores2 =
    iterate (concatMap (\x -> [x-1,x+1])) [1]

-- La propiedad es
prop_suma :: Int -> Property
prop_suma n =
    n >= 0 ==> sum (antecesoresYsucesores2 !! n) == 2^n

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_suma
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Los grafos no dirigidos puede representarse mediante
-- matrices de adyacencia y también mediante listas de adyacencia. Por
-- ejemplo, el grafo
--    1 ----- 2
--    | \     |
--    |  3    |
--    | /     |
--    4 ----- 5
-- se puede representar por la matriz de adyacencia
```

```
--    |0 1 1 1 0|
--    |1 0 0 0 1|
--    |1 0 0 1 0|
--    |1 0 1 0 1|
--    |0 1 0 1 0|
-- donde el elemento (i,j) es 1 si hay una arista entre los vértices i y
-- j y es 0 si no la hay. También se puede representar por la lista de
-- adyacencia
--    [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3,5]),(5,[2,4])]
-- donde las primeras componentes son los vértices y las segundas
-- la lista de los vértices conectados.
--
-- Definir la función
--    matrizAlista :: Matrix Int -> [(Int,[Int])]
-- tal que (matrizAlista a) es la lista de adyacencia correspondiente a
-- la matriz de adyacencia a. Por ejemplo, definiendo la matriz anterior
-- por
--    ejMatriz :: Matrix Int
--    ejMatriz = fromLists [[0,1,1,1,0],
--                          [1,0,0,0,1],
--                          [1,0,0,1,0],
--                          [1,0,1,0,1],
--                          [0,1,0,1,0]]
-- se tiene que
--    ghci> matrizAlista ejMatriz
--    [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3,5]),(5,[2,4])]
-- ---------------------------------------------------------------------

ejMatriz :: Matrix Int
ejMatriz = fromLists [[0,1,1,1,0],
                      [1,0,0,0,1],
                      [1,0,0,1,0],
                      [1,0,1,0,1],
                      [0,1,0,1,0]]

matrizAlista :: Matrix Int -> [(Int,[Int])]
matrizAlista a =
    [(i,[j | j <- [1..n], a!(i,j) == 1]) | i <- [1..n]]
    where n = nrows a
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    listaAmatriz :: [(Int,[Int])] -> Matrix Int
-- tal que (listaAmatriz ps) es la matriz de adyacencia correspondiente
-- a la lista de adyacencia ps. Por ejemplo,
--    ghci> listaAmatriz [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3,5]),(5,[2,4])]
--    ( 0 1 1 1 0 )
--    ( 1 0 0 0 1 )
--    ( 1 0 0 1 0 )
--    ( 1 0 1 0 1 )
--    ( 0 1 0 1 0 )
--    ghci> matrizAlista it
--    [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3,5]),(5,[2,4])]
-- ---------------------------------------------------------------------

listaAmatriz :: [(Int,[Int])] -> Matrix Int
listaAmatriz ps = fromLists [fila n xs | (_,xs) <- sort ps]
    where n = length ps
          fila n xs = [f i | i <- [1..n]]
              where f i | i `elem` xs = 1
                        | otherwise   = 0
```

6.5.8. Examen 8 (4 de septiembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de septiembre (4 de septiembre de 2015)
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import Test.QuickCheck
import I1M.PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    numeroBloquesRepeticion :: Eq a => [a] -> Int
-- tal que (numeroBloquesRepeticion xs) es el número de bloques de
-- elementos consecutivos repetidos en 'xs'. Por ejemplo,
--    numeroBloquesRepeticion [1,1,2,2,3,3]  ==  3
--    numeroBloquesRepeticion [1,1,1,2,3,3]  ==  2
```

```
--    numeroBloquesRepeticion [1,1,2,3]      ==  1
--    numeroBloquesRepeticion [1,2,3]        ==  0
-- ---------------------------------------------------------------------

-- 1ª definición
numeroBloquesRepeticion1 :: Eq a => [a] -> Int
numeroBloquesRepeticion1 xs =
    length (filter (\ys -> length ys > 1) (group xs))

-- 2ª definición (por recursión):
numeroBloquesRepeticion2 :: Eq a => [a] -> Int
numeroBloquesRepeticion2 (x:y:zs)
    | x == y    = 1 + numeroBloquesRepeticion2 (dropWhile (==x) zs)
    | otherwise = numeroBloquesRepeticion2 (y:zs)
numeroBloquesRepeticion2 _ = 0

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Los grafos se pueden representar mediante una lista de
-- pares donde las primeras componentes son los vértices y las segundas
-- la lista de los vértices conectados. Por ejemplo, el grafo
--    1 ----- 2
--    | \     |
--    |  3    |
--    | /     |
--    4 ----- 5
-- se representa por
--    [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3,5]),(5,[2,4])]
-- En Haskell se puede definir el tipo de los grafos por
--    type Grafo a = [(a,[a])]
-- y el ejemplo anterior se representa por
--    ejGrafo :: Grafo Int
--    ejGrafo = [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3]),(5,[2,4])]
--
-- Definir la función
--    aristas :: Ord a => Grafo a -> [(a,a)]
-- tal que (aristas g) es la lista de aristas del grafo g. Por ejemplo,
--    aristas ejGrafo  ==  [(1,2),(1,3),(1,4),(2,5),(3,4)]
-- ---------------------------------------------------------------------

type Grafo a = [(a,[a])]
```

```
ejGrafo :: Grafo Int
ejGrafo = [(1,[2,3,4]),(2,[1,5]),(3,[1,4]),(4,[1,3]),(5,[2,4])]

aristas :: Ord a => Grafo a -> [(a,a)]
aristas g = [(x,y) | (x,ys) <- g, y <- ys, x < y]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. El grafo línea de un grafo G es el grafo L(G) tal que
-- + los vértices de L(G) son las aristas de G y
-- + dos vértices de L(G) son adyacentes si y sólo si sus aristas
--   correspondientes tienen un extremo común en G.
--
-- Definir la función
--    grafoLinea :: Ord a => Grafo a -> Grafo (a,a)
-- tal que (grafoLinea g) es el grafo línea de g. Por ejemplo
--    ghci> grafoLinea ejGrafo
--    [((1,2),[(1,3),(1,4),(2,5)]),
--     ((1,3),[(1,2),(1,4),(3,4)]),
--     ((1,4),[(1,2),(1,3),(3,4)]),
--     ((2,5),[(1,2)]),
--     ((3,4),[(1,3),(1,4)])]
-- ---------------------------------------------------------------------

grafoLinea :: Ord a => Grafo a -> Grafo (a,a)
grafoLinea g =
    [(a1,[a2 | a2 <- as, conExtremoComun a1 a2, a1 /= a2]) | a1 <- as]
    where as = aristas g

conExtremoComun :: Eq a => (a,a) -> (a,a) -> Bool
conExtremoComun (x1,y1) (x2,y2) =
    not (null ([x1,y1] `intersect` [x2,y2]))

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. La sucesión de polinomios de Fibonacci se define por
--    p(0) = 0
--    p(1) = 1
--    p(n) = x*p(n-1) + p(n-2)
-- Los primeros términos de la sucesión son
--    p(2) = x
```

```
--    p(3) = x^2 + 1
--    p(4) = x^3 + 2*x
--    p(5) = x^4 + 3*x^2 + 1
--
-- Definir la lista
--    sucPolFib :: [Polinomio Integer]
-- tal que sus elementos son los polinomios de Fibonacci. Por ejemplo,
--    ghci> take 6 sucPolFib
--    [0,1,1*x,x^2 + 1,x^3 + 2*x,x^4 + 3*x^2 + 1]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

sucPolFib :: [Polinomio Integer]
sucPolFib = [polFibR n | n <- [0..]]

polFibR :: Integer -> Polinomio Integer
polFibR 0 = polCero
polFibR 1 = polUnidad
polFibR n =
    sumaPol (multPol (consPol 1 1 polCero) (polFibR (n-1)))
            (polFibR (n-2))

-- 2ª definición (dinámica)
-- ========================

sucPolFib2 :: [Polinomio Integer]
sucPolFib2 =
    polCero : polUnidad : zipWith f (tail sucPolFib2) sucPolFib2
    where f p = sumaPol (multPol (consPol 1 1 polCero) p)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que el valor del n-ésimo
-- término de sucPolFib para x=1 es el n-ésimo término de la sucesión de
-- Fibonacci 0, 1, 1, 2, 3, 5, 8, ...
--
-- Nota. Limitar la búsqueda a ejemplos pequeños usando
--    quickCheckWith (stdArgs {maxSize=5}) prop_polFib
-- ---------------------------------------------------------------------
```

```
-- La propiedad es
prop_polFib :: Integer -> Property
prop_polFib n =
    n >= 0 ==> valor (polFib n) 1 == fib n
    where polFib n = sucPolFib2 `genericIndex` n
          fib n    = fibs `genericIndex` n

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=5}) prop_polFib
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los números triangulares se forman como sigue
--
--    *     *      *
--         * *    * *
--               * * *
--    1     3      6
--
-- La sucesión de los números triangulares se obtiene sumando los
-- números naturales. Así, los 5 primeros números triangulares son
--     1 = 1
--     3 = 1+2
--     6 = 1+2+3
--    10 = 1+2+3+4
--    15 = 1+2+3+4+5
--
-- Definir la función
--    descomposicionesTriangulares :: Int -> [(Int, Int, Int)]
-- tal que (descomposicionesTriangulares n) es la lista de las
-- ternas correspondientes a las descomposiciones de n en tres sumandos,
-- como máximo, formados por números triangulares. Por ejemplo,
--    ghci> descomposicionesTriangulares 6
--    [(0,0,6),(0,3,3)]
--    ghci> descomposicionesTriangulares 26
--    [(1,10,15),(6,10,10)]
```

```
--    ghci> descomposicionesTriangulares 96
--    [(3,15,78),(6,45,45),(15,15,66),(15,36,45)]
-- ---------------------------------------------------------------------

descomposicionesTriangulares :: Int -> [(Int, Int, Int)]
descomposicionesTriangulares n =
    [(x,y,n-x-y) | x <- xs,
                   y <- dropWhile (<x) xs,
                   n-x-y `elem` dropWhile (<y) xs]
    where xs = takeWhile (<=n) triangulares

-- triangulares es la lista de los números triangulares. Por ejemplo,
--    take 10 triangulares  ==  [0,1,3,6,10,15,21,28,36,45]
triangulares :: [Int]
triangulares = scanl (+) 0 [1..]

-- ---------------------------------------------------------------------
-- Ejercicio 5. En este problema se consideran matrices cuyos elementos
-- son 0 y 1.  Los valores 1 aparecen en forma de islas rectangulares
-- separadas por 0 de forma que como máximo las islas son diagonalmente
-- adyacentes. Por ejemplo,
--    ej1, ej2 :: Array (Int,Int) Int
--    ej1 = listArray ((1,1),(6,3))
--                    [0,0,0,
--                     1,1,0,
--                     1,1,0,
--                     0,0,1,
--                     0,0,1,
--                     1,1,0]
--    ej2 = listArray ((1,1),(6,6))
--                    [1,0,0,0,0,0,
--                     1,0,1,1,1,1,
--                     0,0,0,0,0,0,
--                     1,1,1,0,1,1,
--                     1,1,1,0,1,1,
--                     0,0,0,0,1,1]
--
-- Definir la función
--    numeroDeIslas :: Array (Int,Int) Int -> Int
-- tal que (numeroDeIslas p) es el número de islas de la matriz p. Por
```

```
-- ejemplo,
--    numeroDeIslas ej1  ==  3
--    numeroDeIslas ej2  ==  4
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

ej1, ej2 :: Array (Int,Int) Int
ej1 = listArray ((1,1),(6,3))
                [0,0,0,
                 1,1,0,
                 1,1,0,
                 0,0,1,
                 0,0,1,
                 1,1,0]
ej2 = listArray ((1,1),(6,6))
                [1,0,0,0,0,0,
                 1,0,1,1,1,1,
                 0,0,0,0,0,0,
                 1,1,1,0,1,1,
                 1,1,1,0,1,1,
                 0,0,0,0,1,1]

numeroDeIslas :: Array (Int,Int) Int -> Int
numeroDeIslas p =
    length [(i,j) | (i,j) <- indices p,
                     verticeSuperiorIzquierdo p (i,j)]

-- (verticeSuperiorIzquierdo p (i,j)) se verifica si (i,j) es el
-- vértice superior izquierdo de algunas de las islas de la matriz p,
-- Por ejemplo,
--    ghci> [(i,j) | (i,j) <- indices ej1, verticeSuperiorIzquierdo ej1 (i,j)]
--    [(2,1),(4,3),(6,1)]
--    ghci> [(i,j) | (i,j) <- indices ej2, verticeSuperiorIzquierdo ej2 (i,j)]
--    [(1,1),(2,3),(4,1),(4,5)]
verticeSuperiorIzquierdo :: Matriz -> (Int,Int) -> Bool
verticeSuperiorIzquierdo p (i,j) =
    enLadoSuperior p (i,j) && enLadoIzquierdo p (i,j)

-- (enLadoSuperior p (i,j)) se verifica si (i,j) está en el lado
```

```
-- superior de algunas de las islas de la matriz p, Por ejemplo,
--    ghci> [(i,j) | (i,j) <- indices ej1, enLadoSuperior ej1 (i,j)]
--    [(2,1),(2,2),(4,3),(6,1),(6,2)]
--    ghci> [(i,j) | (i,j) <- indices ej2, enLadoSuperior ej2 (i,j)]
--    [(1,1),(2,3),(2,4),(2,5),(2,6),(4,1),(4,2),(4,3),(4,5),(4,6)]
enLadoSuperior :: Matriz -> (Int,Int) -> Bool
enLadoSuperior p (1,j) = p!(1,j) == 1
enLadoSuperior p (i,j) = p!(i,j) == 1 && p!(i-1,j) == 0

-- (enLadoIzquierdo p (i,j)) se verifica si (i,j) está en el lado
-- izquierdo de algunas de las islas de la matriz p, Por ejemplo,
--    ghci> [(i,j) | (i,j) <- indices ej1, enLadoIzquierdo ej1 (i,j)]
--    [(2,1),(3,1),(4,3),(5,3),(6,1)]
--    ghci> [(i,j) | (i,j) <- indices ej2, enLadoIzquierdo ej2 (i,j)]
--    [(1,1),(2,1),(2,3),(4,1),(4,5),(5,1),(5,5),(6,5)]
enLadoIzquierdo :: Matriz -> (Int,Int) -> Bool
enLadoIzquierdo p (i,1) = p!(i,1) == 1
enLadoIzquierdo p (i,j) = p!(i,j) == 1 && p!(i,j-1) == 0

-- 2ª solución
-- ===========

numeroDeIslas2 :: Array (Int,Int) Int -> Int
numeroDeIslas2 p =
    length [(i,j) | (i,j) <- indices p,
                    p!(i,j) == 1,
                    i == 1 || p!(i-1,j) == 0,
                    j == 1 || p!(i,j-1) == 0]
```

6.5.9. Examen 9 (4 de diciembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 3º convocatoria (4 de diciembre de 2015)
-- ---------------------------------------------------------------------

-- Puntuación: Cada uno de los 5 ejercicios vale 2 puntos.

-- ---------------------------------------------------------------------
-- § Librerías auxiliares                                              --
```

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

import Data.List
import Data.Matrix
import Data.Numbers.Primes
import I1M.BusquedaEnEspaciosDeEstados
import I1M.Grafo
import I1M.PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número tiene factorización capicúa si puede escribir
-- como un producto de números primos tal que la concatenación de sus
-- dígitos forma un número capicúa. Por ejemplo, el 2015 tiene
-- factorización capicúa ya que 2015 = 13*5*31, los factores son primos
-- y su concatenación es 13531 que es capicúa.
--
-- Definir la sucesión
--    conFactorizacionesCapicuas :: [Int]
-- formada por los números que tienen factorización capicúa. Por
-- ejemplo,
--    ghci> take 20 conFactorizacionesCapicuas
--    [1,2,3,4,5,7,8,9,11,12,16,18,20,25,27,28,32,36,39,44]
--
-- Usando conFactorizacionesCapicuas calcular cuál será el siguiente año
-- con factorización capicúa.
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

conFactorizacionesCapicuas :: [Int]
conFactorizacionesCapicuas =
    [n | n <- [1..], not (null (factorizacionesCapicua n))]

-- (factorizacionesCapicua n) es la lista de las factorizaciones
-- capicúas de n. Por ejemplo,
--    factorizacionesCapicua 2015  ==  [[13,5,31],[31,5,13]]
factorizacionesCapicua :: Int -> [[Int]]
factorizacionesCapicua n =
    [xs | xs <- permutations (factorizacion n),
```

```
          esCapicuaConcatenacion xs]

-- (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]

-- (esCapicuaConcatenacion xs) se verifica si la concatenación de los
-- números de xs es capicúa. Por ejemplo,
--    esCapicuaConcatenacion [13,5,31]   ==  True
--    esCapicuaConcatenacion [135,31]    ==  True
--    esCapicuaConcatenacion [135,21]    ==  False
esCapicuaConcatenacion :: [Int] -> Bool
esCapicuaConcatenacion xs = ys == reverse ys
    where ys = concatMap show xs

-- 2ª definición
-- =============

conFactorizacionesCapicuas2 :: [Int]
conFactorizacionesCapicuas2 =
    [n | n <- [1..], not (null (factorizacionesCapicua2 n))]

-- (factorizacionesCapicua2 n) es la lista de las factorizaciones
-- capicúas de n. Por ejemplo,
--    factorizacionesCapicua2 2015  ==  [[13,5,31],[31,5,13]]
factorizacionesCapicua2 :: Int -> [[Int]]
factorizacionesCapicua2 n =
    [xs | xs <- permutations (primeFactors n),
          esCapicuaConcatenacion xs]
```

```
-- 3ª definición
-- =============

conFactorizacionesCapicuas3 :: [Int]
conFactorizacionesCapicuas3 =
    [n | n <- [1..], conFactorizacionCapicua n]

-- (conFactorizacionCapicua n) se verifica si n tiene factorización
-- capicúa. Por ejemplo,
--    factorizacionesCapicua2 2015  ==  [[13,5,31],[31,5,13]]
conFactorizacionCapicua :: Int -> Bool
conFactorizacionCapicua n =
    any listaCapicua (permutations (primeFactors n))

listaCapicua :: Show a => [a] -> Bool
listaCapicua xs = ys == reverse ys
    where ys = concatMap show xs

-- El cálculo es
--    ghci> head (dropWhile (<=2015) conFactorizacionesCapicuas)
--    2023

-- ---------------------------------------------------------------------
-- Ejercicio 2. 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
--          "C"
--          / \
--         /   \
--        /     \
--      "B"     "A"
--      / \     / \
--    "A" "B" "B" "C"
-- se puede definir por
--    ej1 :: Arbol String
--    ej1 = N "C" (N "B" (H "A") (H "B")) (N "A" (H "B") (H "C"))
```

```
--
-- Definir la función
--    renombraArbol :: Arbol t -> Arbol Int
-- tal que (renombraArbol a) es el árbol obtenido sustituyendo el valor
-- de los nodos y hojas por números tales que tengan el mismo valor si y
-- sólo si coincide su contenido. Por ejemplo,
--    ghci> renombraArbol ej1
--    N 2 (N 1 (H 0) (H 1)) (N 0 (H 1) (H 2))
-- Gráficamente,
--          2
--         / \
--        /   \
--       /     \
--      1       0
--     / \     / \
--    0   1   1   2
-- Nótese que los elementos del árbol pueden ser de cualquier tipo. Por
-- ejemplo,
--    ghci> renombraArbol (N 9 (N 4 (H 8) (H 4)) (N 8 (H 4) (H 9)))
--    N 2 (N 0 (H 1) (H 0)) (N 1 (H 0) (H 2))
--    ghci> renombraArbol (N True (N False (H True) (H False)) (H True))
--    N 1 (N 0 (H 1) (H 0)) (H 1)
--    ghci> renombraArbol (N False (N False (H True) (H False)) (H True))
--    N 0 (N 0 (H 1) (H 0)) (H 1)
--    ghci> renombraArbol (H False)
--    H 0
--    ghci> renombraArbol (H True)
--    H 0
-- ---------------------------------------------------------------------

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

ej1 :: Arbol String
ej1 = N "C" (N "B" (H "A") (H "B")) (N "A" (H "B") (H "C"))

renombraArbol :: Ord t => Arbol t -> Arbol Int
renombraArbol a = aux a
    where ys            = valores a
```

```
          aux (H x)     = H (posicion x ys)
          aux (N x i d) = N (posicion x ys) (aux i) (aux d)

-- (valores a) es la lista de los valores en los nodos y las hojas del
-- árbol a. Por ejemplo,
--    valores ej1  ==  ["A","B","C"]
valores :: Ord a => Arbol a -> [a]
valores a = sort (nub (aux a))
    where aux (H x)     = [x]
          aux (N x i d) = x : (aux i ++ aux d)

-- (posicion x ys) es la posición de x en ys. Por ejemplo.
--    posicion 7 [5,3,7,8]  ==  2
posicion :: Eq a => a -> [a] -> Int
posicion x ys = head [n | (y,n) <- zip ys [0..], y == x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. El buscaminas es un juego cuyo objetivo es despejar un
-- campo de minas sin detonar ninguna.
--
-- El campo de minas se representa mediante un cuadrado con NxN
-- casillas. Algunas casillas tienen un número, este número indica las
-- minas que hay en todas las casillas vecinas. Cada casilla tiene como
-- máximo 8 vecinas. Por ejemplo, el campo 4x4 de la izquierda
-- contiene dos minas, cada una representada por el número 9, y a la
-- derecha se muestra el campo obtenido anotando las minas vecinas de
-- cada casilla
--    9 0 0 0       9 1 0 0
--    0 0 0 0       2 2 1 0
--    0 9 0 0       1 9 1 0
--    0 0 0 0       1 1 1 0
-- de la misma forma, la anotación del siguiente a la izquierda es el de
-- la derecha
--    9 9 0 0 0     9 9 1 0 0
--    0 0 0 0 0     3 3 2 0 0
--    0 9 0 0 0     1 9 1 0 0
--
-- Utilizando la librería Data.Matrix, los campos de minas se
-- representan mediante matrices:
--    type Campo = Matrix Int
```

```
-- Por ejemplo, los anteriores campos de la izquierda se definen por
--    ejCampo1, ejCampo2 :: Campo
--    ejCampo1 = fromLists [[9,0,0,0],
--                          [0,0,0,0],
--                          [0,9,0,0],
--                          [0,0,0,0]]
--    ejCampo2 = fromLists [[9,9,0,0,0],
--                          [0,0,0,0,0],
--                          [0,9,0,0,0]]
--
-- Definir la función
--    buscaminas :: Campo -> Campo
-- tal que (buscaminas c) es el campo obtenido anotando las minas
-- vecinas de cada casilla. Por ejemplo,
--    ghci> buscaminas ejCampo1
--    ( 9 1 0 0 )
--    ( 2 2 1 0 )
--    ( 1 9 1 0 )
--    ( 1 1 1 0 )
--
--    ghci> buscaminas ejCampo2
--    ( 9 9 1 0 0 )
--    ( 3 3 2 0 0 )
--    ( 1 9 1 0 0 )
-- ---------------------------------------------------------------------

type Campo   = Matrix Int
type Casilla = (Int,Int)

ejCampo1, ejCampo2 :: Campo
ejCampo1 = fromLists [[9,0,0,0],
                      [0,0,0,0],
                      [0,9,0,0],
                      [0,0,0,0]]
ejCampo2 = fromLists [[9,9,0,0,0],
                      [0,0,0,0,0],
                      [0,9,0,0,0]]

-- 1ª solución
-- ===========
```

```
buscaminas1 :: Campo -> Campo
buscaminas1 c = matrix m n (\(i,j) -> minas c (i,j))
    where m = nrows c
          n = ncols c

-- (minas c (i,j)) es el número de minas en las casillas vecinas de la
-- (i,j) en el campo de mina c y es 9 si en (i,j) hay una mina. Por
-- ejemplo,
--    minas ejCampo (1,1)  ==  9
--    minas ejCampo (1,2)  ==  1
--    minas ejCampo (1,3)  ==  0
--    minas ejCampo (2,1)  ==  2
minas :: Campo -> Casilla -> Int
minas c (i,j)
    | c!(i,j) == 9 = 9
    | otherwise    = length (filter (==9)
                              [c!(x,y) | (x,y) <- vecinas m n (i,j)])
                     where m = nrows c
                           n = ncols c

-- (vecinas m n (i,j)) es la lista de las casillas vecinas de la (i,j) en
-- un campo de dimensiones mxn. Por ejemplo,
--    vecinas 4 (1,1)  ==  [(1,2),(2,1),(2,2)]
--    vecinas 4 (1,2)  ==  [(1,1),(1,3),(2,1),(2,2),(2,3)]
--    vecinas 4 (2,3)  ==  [(1,2),(1,3),(1,4),(2,2),(2,4),(3,2),(3,3),(3,4)]
vecinas :: Int -> Int -> Casilla -> [Casilla]
vecinas m n (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)]

-- 2ª solución
-- ===========

buscaminas2 :: Campo -> Campo
buscaminas2 c = matrix m n (\(i,j) -> minas (i,j))
    where m = nrows c
          n = ncols c
          minas :: Casilla -> Int
          minas (i,j)
```

```
                | c!(i,j) == 9 = 9
                | otherwise    =
                    length (filter (==9) [c!(x,y) | (x,y) <- vecinas (i,j)])
          vecinas :: Casilla -> [Casilla]
          vecinas (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                                   b <- [max 1 (j-1)..min n (j+1)],
                                   (a,b) /= (i,j)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. La codificación de Fibonacci de un número n es una
-- cadena d = d(0)d(1)...d(k-1)d(k) de ceros y unos tal que
--    n = d(0)*F(2) + d(1)*F(3) +...+ d(k-1)*F(k+1)
--    d(k-1) = d(k) = 1
-- donde F(i) es el i-ésimo término de la sucesión de Fibonacci
--    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
-- Por ejemplo, la codificación de Fibonacci de 4 es "1011" ya que los
-- dos últimos elementos son iguales a 1 y
--    1*F(2) + 0*F(3) + 1*F(4) = 1*1 + 0*2 + 1*3 = 4
-- La codificación de Fibonacci de los primeros números se muestra en la
-- siguiente tabla
--     1  = 1     = F(2)           ≡       11
--     2  = 2     = F(3)           ≡      011
--     3  = 3     = F(4)           ≡     0011
--     4  = 1+3   = F(2)+F(4)      ≡     1011
--     5  = 5     = F(5)           ≡    00011
--     6  = 1+5   = F(2)+F(5)      ≡    10011
--     7  = 2+5   = F(3)+F(5)      ≡    01011
--     8  = 8     = F(6)           ≡   000011
--     9  = 1+8   = F(2)+F(6)      ≡   100011
--    10  = 2+8   = F(3)+F(6)      ≡   010011
--    11  = 3+8   = F(4)+F(6)      ≡   001011
--    12  = 1+3+8 = F(2)+F(4)+F(6) ≡   101011
--    13  = 13    = F(7)           ≡  0000011
--    14  = 1+13  = F(2)+F(7)      ≡  1000011
--
-- Definir la función
--    codigoFib :: Integer -> String
-- tal que (codigoFib n) es la codificación de Fibonacci del número
-- n. Por ejemplo,
--    ghci> codigoFib 65
```

```
--    "0100100011"
--    ghci> [codigoFib n | n <- [1..7]]
--    ["11","011","0011","1011","00011","10011","01011"]
-- ---------------------------------------------------------------------

codigoFib :: Integer -> String
codigoFib = (concatMap show) . codificaFibLista

-- (codificaFibLista n) es la lista correspondiente a la codificación de
-- Fibonacci del número n. Por ejemplo,
--    ghci> codificaFibLista 65
--    [0,1,0,0,1,0,0,0,1,1]
--    ghci> [codificaFibLista n | n <- [1..7]]
--    [[1,1],[0,1,1],[0,0,1,1],[1,0,1,1],[0,0,0,1,1],[1,0,0,1,1],[0,1,0,1,1]]
codificaFibLista :: Integer -> [Integer]
codificaFibLista n = map f [2..head xs] ++ [1]
    where xs = map fst (descomposicion n)
          f i | elem i xs = 1
              | otherwise = 0

-- (descomposicion n) es la lista de pares (i,f) tales que f es el
-- i-ésimo número de Fibonacci y las segundas componentes es una
-- sucesión decreciente de números de Fibonacci cuya suma es n. Por
-- ejemplo,
--    descomposicion 65  ==  [(10,55),(6,8),(3,2)]
--    descomposicion 66  ==  [(10,55),(6,8),(4,3)]
descomposicion :: Integer -> [(Integer, Integer)]
descomposicion 0 = []
descomposicion 1 = [(2,1)]
descomposicion n = (i,x) : descomposicion (n-x)
    where (i,x) = fibAnterior n

-- (fibAnterior n) es el mayor número de Fibonacci menor o igual que
-- n. Por ejemplo,
--    fibAnterior 33  ==  (8,21)
--    fibAnterior 34  ==  (9,34)
fibAnterior :: Integer -> (Integer, Integer)
fibAnterior n = last (takeWhile p fibsConIndice)
    where p (i,x) = x <= n
```

```
-- fibsConIndice es la sucesión de los números de Fibonacci junto con
-- sus índices. Por ejemplo,
--    ghci> take 10 fibsConIndice
--    [(0,0),(1,1),(2,1),(3,2),(4,3),(5,5),(6,8),(7,13),(8,21),(9,34)]
fibsConIndice :: [(Integer, Integer)]
fibsConIndice = zip [0..] fibs

-- fibs es la sucesión de Fibonacci. Por ejemplo,
--    take 10 fibs  ==  [0,1,1,2,3,5,8,13,21,34]
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir las funciones
--    grafo   :: [(Int,Int)] -> Grafo Int Int
--    caminos :: Grafo Int Int -> Int -> Int -> [[Int]]
-- tales que
-- + (grafo as) es el grafo no dirigido definido cuyas aristas son as. Por
--   ejemplo,
--      ghci> grafo [(2,4),(4,5)]
--      G ND (array (2,5) [(2,[(4,0)]),(3,[]),(4,[(2,0),(5,0)]),(5,[(4,0)])])
-- + (caminos g a b) es la lista los caminos en el grafo g desde a hasta
--   b sin pasar dos veces por el mismo nodo. Por ejemplo,
--      ghci> sort (caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 7)
--      [[1,3,5,7],[1,3,7]]
--      ghci> sort (caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 2 7)
--      [[2,5,3,7],[2,5,7]]
--      ghci> sort (caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 2)
--      [[1,3,5,2],[1,3,7,5,2]]
--      ghci> caminos (grafo [(1,3),(2,5),(3,5),(3,7),(5,7)]) 1 4
--      []
--      ghci> length (caminos (grafo [(i,j) | i <- [1..10], j <- [i..10]]) 1 10)
--      109601
-- ---------------------------------------------------------------------

grafo :: [(Int,Int)] -> Grafo Int Int
grafo as = creaGrafo ND (m,n) [(x,y,0) | (x,y) <- as]
    where ns = map fst as ++ map snd as
          m  = minimum ns
          n  = maximum ns
```

```
-- 1ª solución
-- ===========

caminos :: Grafo Int Int -> Int -> Int -> [[Int]]
caminos g a b = aux [[b]] where
    aux [] = []
    aux ((x:xs):yss)
        | x == a    = (x:xs) : aux yss
        | otherwise = aux ([z:x:xs | z <- adyacentes g x
                                   , z `notElem` (x:xs)]
                           ++ yss)

-- 2ª solución (mediante espacio de estados)
-- =========================================

caminos2 :: Grafo Int Int -> Int -> Int -> [[Int]]
caminos2 g a b = buscaEE sucesores esFinal inicial
    where inicial          = [b]
          sucesores (x:xs) = [z:x:xs | z <- adyacentes g x
                                     , z `notElem` (x:xs)]
          esFinal (x:xs)   = x == a

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

--    ghci> length (caminos (grafo [(i,j) | i <- [1..10], j <- [i..10]]) 1 10)
--    109601
--    (3.57 secs, 500533816 bytes)
--    ghci> length (caminos2 (grafo [(i,j) | i <- [1..10], j <- [i..10]]) 1 10)
--    109601
--    (3.53 secs, 470814096 bytes)
```

7

Exámenes del curso 2015–16

7.1. Exámenes del grupo 1 (María J. Hidalgo)

7.1.1. Examen 1 (3 de Noviembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (3 de noviembre de 2015)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La suma de la serie
--    1/3 + 1/15 + 1/35 + 1/63 + ...+ 1/(4*x^2-1) + ...
-- es 1/2.
--
-- Definir la función
--    sumaS2:: Double -> Double
-- tal que (sumaS2 n) es la aproximación de 1/2 obtenida mediante n
-- términos de la serie. Por ejemplo,
--    sumaS2 10  == 0.4761904761904761
--    sumaS2 100 == 0.49751243781094495
-- ---------------------------------------------------------------------

sumaS2 :: Double -> Double
sumaS2 n = sum [1/(4*x^2-1) | x <- [1..n]]

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

```
-- Ejercicio 1.2. Comprobar con QuickCheck que la sumas finitas de la
-- serie siempre son menores que 1/2.
-- ---------------------------------------------------------------------

-- La propiedad es
propSumaS2 :: Double -> Bool
propSumaS2 n = sumaS2 n < 0.5

-- La comprobación es
--    ghci> quickCheck propSumaS2
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    menorNError :: Double -> Double
-- tal que (menorNError x) es el menor número de términos de la serie
-- anterior necesarios para obtener 1/2 con un error menor que x. Por
-- ejemplo,
--    menorNError 0.01  == 25.0
--    menorNError 0.001 == 250.0
-- ---------------------------------------------------------------------

menorNError :: Double -> Double
menorNError x =
    head [n | n <- [1..], 0.5 - sumaS2 n < x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Decimos que un número n es "muy divisible por 3" si es
-- divisible por 3 y sigue siendo divisible por 3 si vamos quitando
-- dígitos por la derecha. Por ejemplo, 96060 es muy divisible por 3
-- porque 96060, 9606, 960, 96 y 9 son todos divisibles por 3.
--
-- Definir la función
--    muyDivPor3 :: Integer -> Bool
-- tal que (muyDivPor3 n) se verifica si n es muy divisible por 3. Por
-- ejemplo,
--    muyDivPor3 96060 == True
--    muyDivPor3 90616 == False
-- ---------------------------------------------------------------------
```

```
muyDivPor3 :: Integer -> Bool
muyDivPor3 n
    | n < 10    = n `rem` 3 == 0
    | otherwise = n `rem` 3 == 0 && muyDivPor3 (n `div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    numeroMuyDivPor3Cifras :: Integer -> Integer
-- tal que (numeroMuyDivPor3CifrasC k) es la cantidad de números de k
-- cifras muy divisibles por 3. Por ejemplo,
--    numeroMuyDivPor3Cifras 5 == 768
--    numeroMuyDivPor3Cifras 7 == 12288
--    numeroMuyDivPor3Cifras 9 == 196608
-- ---------------------------------------------------------------------

-- 1ª definición
numeroMuyDivPor3Cifras :: Integer -> Integer
numeroMuyDivPor3Cifras k =
    genericLength [x | x <- [10^(k-1)..10^k-1], muyDivPor3 x]

-- 2ª definición
numeroMuyDivPor3Cifras2 :: Integer -> Integer
numeroMuyDivPor3Cifras2 k =
    genericLength [x | x <- [n,n+3..10^k-1], muyDivPor3 x]
    where n = k*10^(k-1)

-- 3ª definición
-- =============

numeroMuyDivPor3Cifras3 :: Integer -> Integer
numeroMuyDivPor3Cifras3 k = genericLength (numeroMuyDivPor3Cifras3' k)

numeroMuyDivPor3Cifras3' :: Integer -> [Integer]
numeroMuyDivPor3Cifras3' 1 = [3,6,9]
numeroMuyDivPor3Cifras3' k =
    [10*x+y | x <- numeroMuyDivPor3Cifras3' (k-1),
              y <- [0,3..9]]

-- 4ª definición
-- =============
```

```
numeroMuyDivPor3Cifras4 :: Integer -> Integer
numeroMuyDivPor3Cifras4 1 = 3
numeroMuyDivPor3Cifras4 k = 4 * numeroMuyDivPor3Cifras4 (k-1)

-- 5ª definición
numeroMuyDivPor3Cifras5 :: Integer -> Integer
numeroMuyDivPor3Cifras5 k = 3 * 4^(k-1)

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

--    ghci> numeroMuyDivPor3Cifras 6
--    3072
--    (3.47 secs, 534,789,608 bytes)
--    ghci> numeroMuyDivPor3Cifras2 6
--    2048
--    (0.88 secs, 107,883,432 bytes)
--    ghci> numeroMuyDivPor3Cifras3 6
--    3072
--    (0.01 secs, 0 bytes)
--
--    ghci> numeroMuyDivPor3Cifras2 7
--    0
--    (2.57 secs, 375,999,336 bytes)
--    ghci> numeroMuyDivPor3Cifras3 7
--    12288
--    (0.02 secs, 0 bytes)
--    ghci> numeroMuyDivPor3Cifras4 7
--    12288
--    (0.00 secs, 0 bytes)
--    ghci> numeroMuyDivPor3Cifras5 7
--    12288
--    (0.01 secs, 0 bytes)
--
--    ghci> numeroMuyDivPor3Cifras4 (10^5) `rem` 100000
--    32032
--    (5.74 secs, 1,408,600,592 bytes)
--    ghci> numeroMuyDivPor3Cifras5 (10^5) `rem` 100000
--    32032
--    (0.02 secs, 0 bytes)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir una función
--    intercala:: [a] -> [a] -> [a]
-- tal que (intercala xs ys) es la lista que resulta de intercalar los
-- elementos de xs con los de ys. Por ejemplo,
--    intercala [1..7] [9,2,0,3] == [1,9,2,2,3,0,4,3,5,6,7]
--    intercala [9,2,0,3] [1..7] == [9,1,2,2,0,3,3,4,5,6,7]
--    intercala "hola" "adios"   == "haodliaos"
-- ----------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. La diferencia simétrica de dos conjuntos es el conjunto
-- cuyos elementos son aquellos que pertenecen a alguno de los conjuntos
-- iniciales, sin pertenecer a ambos a la vez. Por ejemplo, la
-- diferencia simétrica de {2,5,3} y {4,2,3,7} es {5,4,7}.

-- Definir la función
--    diferenciaSimetrica :: Eq a => [a] -> [a] -> [a]
-- tal que (diferenciaSimetrica xs ys) es la diferencia simétrica de xs
-- e ys. Por ejemplo,
--    diferenciaSimetrica [2,5,3] [4,2,3,7]    ==  [5,4,7]
--    diferenciaSimetrica [2,5,3] [5,2,3]      ==  []
--    diferenciaSimetrica [2,5,2] [4,2,3,7]    ==  [5,4,3,7]
--    diferenciaSimetrica [2,5,2] [4,2,4,7]    ==  [5,4,4,7]
--    diferenciaSimetrica [2,5,2,4] [4,2,4,7]  ==  [5,7]
-- ----------------------------------------------------------------------

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

-- ----------------------------------------------------------------------
-- Ejercicio 5. (Basado en el problema 4 del Proyecto Euler)
-- El número 9009 es capicúa y es producto de dos números de dos dígitos,
```

```
-- pues 9009 = 91*99.
--
-- Definir la función
--    numerosC2Menores :: Int -> [Int]
-- tal que (numerosC2Menores n) es la lista de números capicúas menores
-- que n que son producto de 2 números de dos dígitos. Por ejemplo,
--    numerosC2Menores 100           == []
--    numerosC2Menores 400           == [121,242,252,272,323,363]
--    length (numerosC2Menores 1000) == 38
-- ---------------------------------------------------------------------

numerosC2Menores :: Int -> [Int]
numerosC2Menores n = [x | x <- productos n, esCapicua x]

-- (productos n) es la lista de números menores que n que son productos
-- de 2 números de dos dígitos.
productos :: Int -> [Int]
productos n =
    sort (nub [x*y | x <- [10..99], y <- [x..99], x*y < n])

-- 2ª definición de productos
productos2 :: Int -> [Int]
productos2 n =
    init (nub (sort [x*y | x <- [10..min 99 (n `div` 10)],
                           y <- [x..min 99 (n `div` x)]]))

-- (esCapicua x) se verifica si x es capicúa.
esCapicua :: Int -> Bool
esCapicua x = xs == reverse xs
    where xs = show x
```

7.1.2. Examen 2 (3 de Diciembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (3 de diciembre de 2015)
-- ---------------------------------------------------------------------

import Data.List
import Data.Numbers.Primes
import Test.QuickCheck
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir una función
--    sumaCuadradosDivisores1 :: Integer -> Integer
-- que calcule la suma de los cuadrados de los divisores de n. Por
-- ejemplo:
--    sumaCuadradosDivisores1 6  ==  50
-- ---------------------------------------------------------------------

sumaCuadradosDivisores1 :: Integer -> Integer
sumaCuadradosDivisores1 n = sum [x^2 | x <- divisores n]

sumaCuadradosDivisores1' :: Integer -> Integer
sumaCuadradosDivisores1' = sum . map (^2) . divisores

-- (divisores n) es la lista de los divisores de n.
divisores :: Integer -> [Integer]
divisores n = [x | x <- [1..n], n `rem` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. La suma de los cuadrados de los divisores de un número
-- se puede calcular a partir de su factorización prima. En efecto, si
-- la factorización prima de n es
--    a^x*b^y*...*c^z
-- entonces, la suma de los divisores de n es
--   (1+a+a^2+...+a^x) * (1+b+b^2+...+b^y) *...* (1+c+c^2+...+c^z)
-- es decir,
--   ((a^(x+1)-1)/(a-1)) * ((b^(y+1)-1)/(b-1)) *...* ((c^(z+1)-1)/(c-1))
-- Por tanto, la suma de sus cuadrados de los divisores de n
--   ((a^(2*(x+1))-1)/(a^2-1)) * ((b^(2*(y+1))-1)/(b^2-1)) *...*
--
-- Definir, a partir de la nota anterior, la función
--    sumaCuadradosDivisores2 :: Int -> Integer
-- tal que (sumaCuadradosDivisores2 n) es la suma de los cuadrados de
-- los divisores de n. Por ejemplo,
--    sumaCuadradosDivisores2 6  ==  50
-- ---------------------------------------------------------------------

sumaCuadradosDivisores2 :: Integer -> Integer
sumaCuadradosDivisores2 n =
    product [(a^(2*(m+1))-1) `div` (a^2-1) | (a,m) <- factorizacion n]
```

```
-- (factorizacion n) es la factorización prima de n.
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n =
    [(head xs, genericLength xs) | xs <- group (primeFactors n)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comparar las estadísticas del cálculo de las
-- siguientes expresiones
--    sumaCuadradosDivisores1 1000000
--    sumaCuadradosDivisores2 1000000
-- El cálculo es
--    ghci> sumaCuadradosDivisores1 1000000
--    1388804117611
--    (2.91 secs, 104321520 bytes)
--    ghci> sumaCuadradosDivisores2 1000000
--    1388804117611
--    (0.01 secs, 550672 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    cerosDelFactorial :: Integer -> Integer
-- tal que (cerosDelFactorial n) es el número de ceros en que termina el
-- factorial de n. Por ejemplo,
--    cerosDelFactorial 24                         ==  4
--    cerosDelFactorial 25                         ==  6
--    length (show (cerosDelFactorial (1234^5678)))  ==  17552
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

cerosDelFactorial1 :: Integer -> Integer
cerosDelFactorial1 n = ceros (factorial n)

-- (factorial n) es el factorial n. Por ejemplo,
--    factorial 3  ==  6
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
ceros :: Integer -> Integer
ceros n | rem n 10 /= 0 = 0
        | otherwise     = 1 + ceros (div n 10)

-- 2ª definición
-- =============

cerosDelFactorial2 :: Integer -> Integer
cerosDelFactorial2 n = ceros2 (factorial n)

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
ceros2 :: Integer -> Integer
ceros2 n = genericLength (takeWhile (=='0') (reverse (show n)))

-- 3ª definición
-- =============

cerosDelFactorial3 :: Integer -> Integer
cerosDelFactorial3 n | n < 5     = 0
                     | otherwise = m + cerosDelFactorial3 m
                     where m = n `div` 5

-- Comparación de la eficiencia
--    ghci> cerosDelFactorial1 (3*10^4)
--    7498
--    (3.96 secs, 1,252,876,376 bytes)
--    ghci> cerosDelFactorial2 (3*10^4)
--    7498
--    (3.07 secs, 887,706,864 bytes)
--    ghci> cerosDelFactorial3 (3*10^4)
--    7498
--    (0.03 secs, 9,198,896 bytes)


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

```
-- Ejercicio 3.1. El Triángulo de Floyd, llamado así en honor a Robert
-- Floyd, es un triángulo rectángulo formado con números naturales. Para
-- crear un triángulo de Floyd, se comienza con un 1 en la esquina
-- superior izquierda, y se continúa escribiendo la secuencia de los
-- números naturales de manera que cada línea contenga un número más que
-- la anterior:
--    1
--    2    3
--    4    5       6
--    7    8       9       10
--    11   12      13      14      15
--
-- Definir la función
--    trianguloFloyd :: [[Int]]
-- tal que trianguloFloyd es la lista formada por todas las líneas del
-- triángulo. Por ejemplo,
--    ghci> take 10 trianguloFloyd
--    [[1],
--     [2,3],
--     [4,5,6],
--     [7,8,9,10],
--     [11,12,13,14,15],
--     [16,17,18,19,20,21],
--     [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]]
-- ---------------------------------------------------------------------

trianguloFloyd :: [[Integer]]
trianguloFloyd = iterate siguiente [1]

siguiente :: [Integer] -> [Integer]
siguiente xs = [a..a+n]
  where a = 1+last xs
        n = genericLength xs

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

```
-- tal que (filaTrianguloFloyd n) es la n-sima fila del triángulo de
-- Floyd. Por ejemplo,
--   filaTrianguloFloyd 6 == [16,17,18,19,20,21]
-- ---------------------------------------------------------------------

filaTrianguloFloyd :: Integer -> [Integer]
filaTrianguloFloyd n = trianguloFloyd `genericIndex` (n - 1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck la siguiente propiedad: la
-- suma de los números de la línea n es n(n^2 + 1)/2 .
-- ---------------------------------------------------------------------

-- La propiedad es
prop_trianguloFloyd n =
  n > 0 ==> sum (filaTrianguloFloyd n) == (n*(n^2+1)) `div` 2

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo, los árboles
--         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 pueden representar por
--    ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
--    ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
--    ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
--    ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
--    ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))
```

```
--
-- 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 ej3arbol1 ej3arbol2 == True
--    igualEstructura ej3arbol1 ej3arbol3 == False
--    igualEstructura ej3arbol1 ej3arbol4 == False
-- ---------------------------------------------------------------------

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

ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))

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

7.2. Exámenes del grupo 2 (Antonia M. Chávez)

7.3. Exámenes del grupo 3 (Francisco J. Martín)

7.4. Exámenes del grupo 4 (José A. Alonso y Luis Valencia)

7.5. Exámenes del grupo 5 (Andrés Cordón)

Apéndice A

Resumen de funciones predefinidas de Haskell

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

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

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

A.1. Resumen de funciones sobre TAD en Haskell

A.1.1. Polinomios

1. `polCero` es el polinomio cero.
2. `(esPolCero p)` se verifica si `p` es el polinomio cero.
3. `(consPol n b p)` es el polinomio $bx^n + p$.
4. `(grado p)` es el grado del polinomio `p`.
5. `(coefLider p)` es el coeficiente líder del polinomio `p`.
6. `(restoPol p)` es el resto del polinomio `p`.

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

1. `(range m n)` es la lista de los índices del `m` al `n`.

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

A.1.3. Tablas

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

A.1.4. Grafos

1. `(creaGrafo d cs as)` es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso).
2. `(dirigido g)` se verifica si g es dirigido.
3. `(nodos g)` es la lista de todos los nodos del grafo g.

4. `(aristas g)` es la lista de las aristas del grafo `g`.
5. `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`.
6. `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`.
7. `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo `g`.

Apéndice B

Método de Pólya para la resolución de problemas

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

Para resolver un problema se necesita:

Paso 1: Entender el problema

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

Paso 2: Configurar un plan

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

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

Paso 3: Ejecutar el plan

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

Paso 4: Examinar la solución obtenida

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

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

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

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?

- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

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

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.

- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

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

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

Bibliografía

[1] J. A. Alonso. Temas de programación funcional. Technical report, Univ. de Sevilla, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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