Exámenes de "Programación funcional con Haskell"

Vol. 6 (Curso 2014–15)

José A. Alonso Jiménez

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

Sevilla, 20 de diciembre de 2015

Índice general

Introducción

Este libro es una recopilación de las soluciones de ejercicios de los exámenes de programación funcional con Haskell de la asignatura de Informática (curso 2014–15) del Grado en Matemática de la Universidad de Sevilla.

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

- Temas de programación funcional (curso 2014–15) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2014–15) [2]
- Piensa en Haskell (Ejercicios de programación funcional con Haskell) [3]

El libro consta de 5 capítulos correspondientes a 5 grupos de la asignatura. En cada capítulo hay una sección por cada uno de los exámenes del grupo. Los ejercicios de cada examen han sido propuestos por los profesores de su grupo (cuyos nombres aparecen en el título del capítulo). Sin embargo, los he modificado para unificar el estilo de su presentación.

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

Los códigos del libro están disponibles en GitHub [4]

Este libro es el cuarto volumen de la serie de recopilaciones de exámenes de programación funcional con Haskell. Los volúmenes anteriores son

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

[1] https://www.cs.us.es/~jalonso/cursos/i1m-14/temas/2014-15-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-14/ejercicios/ejercicios-I1M-2014.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol6
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

- Exámenes de “Programación funcional con Haskell”. Vol. 2 (Curso 2010–11) [6]
- Exámenes de “Programación funcional con Haskell”. Vol. 3 (Curso 2011–12) [7]
- Exámenes de “Programación funcional con Haskell”. Vol. 4 (Curso 2012–13) [8]
- Exámenes de “Programación funcional con Haskell”. Vol. 5 (Curso 2013–14) [9]

José A. Alonso
Sevilla, 20 de diciembre de 2015

[6] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol2
[7] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol3
[8] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol4
[9] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol5

1

Exámenes del grupo 1

José A. Alonso y Luis Valencia

1.1. 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 x² es
-- potencia de a².
-- ---------------------------------------------------------------------

-- 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"
-- ---------------------------------------------------------------------
```

```haskell
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]
```

```haskell
-- 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 []     = []
```

```haskell
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.
```

1.2. Examen 2 (3 de Diciembre de 2014)

```haskell
-- 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]
-- ---------------------------------------------------------------------
```

```haskell
-- 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,
```

```haskell
--    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.

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

```haskell
-- 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
```

```haskell
--    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
```

1.3. Examen 3 (23 de enero de 2015)

```haskell
-- 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]
```

```haskell
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])
```

```haskell
--    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
```

```haskell
-- 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)
```

```haskell
-- 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
```

```haskell
-- 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)]
```

1.4. Examen 4 (9 de marzo de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (9 de marzo de 2015)
```

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

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

```haskell
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]
-- ---------------------------------------------------------------------
```

```haskell
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)

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

1.5. 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).
--
```

```haskell
-- 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
```

```haskell
-- 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.
--
-- Definir la función
--    complementario :: Grafo Int Int -> Grafo Int Int
-- tal que (complementario g) es el complementario de g. Por ejemplo,
--    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 n² + n + 41
-- genera 40 números primos para todos los valores de n entre 0 y
-- 39. Sin embargo, cuando n=40, 40²+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 n²+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 n² + 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)
```

1.6. 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).
--
```

```haskell
-- 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
```

```haskell
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"])]
-- ---------------------------------------------------------------------

-- 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
```

```haskell
--    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
```

```haskell
--      []
-- ---------------------------------------------------------------------

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 |
--    |--------------------------------+----+-----+----+----+----|
```

1.7. 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)
```

```haskell
-- 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])]
```

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

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
```

1.8. Examen 8 (4 de septiembre de 2015)

```haskell
-- 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)])]
```

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

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

```haskell
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]
```

1.9. Examen 9 (4 de diciembre de 2015)

```haskell
-- 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
```

```haskell
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)
```

```haskell
--    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
```

```haskell
-- 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
```

```haskell
-- 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
```

```haskell
--   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)
```

2

Exámenes del grupo 2

Antonia M. Chávez

2.1. Examen 1 (6 de Noviembre de 2014)

```haskell
-- 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)
```

```haskell
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):
```

```haskell
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
```

2.2. Examen 2 (4 de Diciembre de 2014)

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

```haskell
-- ---------------------------------------------------------------------
-- 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]
```

```haskell
-- 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)
```

2.3. Examen 3 (23 de enero de 2015)

El examen es común con el del grupo 4 (ver página 115).

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

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

```haskell
-- 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)
```

```haskell
-- ---------------------------------------------------------------------
-- 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))
--
```

```haskell
-- 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]
```

2.5. Examen 5 (7 de mayo de 2015)

```haskell
-- 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
```

```haskell
--    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
-- =========================
```

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

```haskell
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
```

```haskell
-- 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
```

```haskell
-- (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
--      []
-- ---------------------------------------------------------------------
```

```haskell
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
```

2.6. Examen 6 (15 de junio de 2015)

El examen es común con el del grupo 4 (ver página 137).

2.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 1 (ver página 42).

2.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 1 (ver página 50).

2.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 1 (ver página 61).

3

Exámenes del grupo 3

Andrés Cordón

3.1. Examen 1 (4 de Noviembre de 2014)

```haskell
-- 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]
-- ---------------------------------------------------------------------
```

```haskell
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
```

```haskell
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
```

```haskell
-- 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))
```

3.2. Examen 2 (5 de Diciembre de 2014)

```haskell
-- 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
```

```haskell
-- 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

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

```haskell
-- 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
```

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

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)

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

```haskell
-- 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
```

```haskell
-- 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 ..])
```

3.3. Examen 3 (23 de enero de 2015)

El examen es común con el del grupo 4 (ver página 115).

3.4. Examen 4 (18 de marzo de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (18 de marzo de 2015)
-- ---------------------------------------------------------------------

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

```haskell
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
```

```haskell
-- 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]]
```

```haskell
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
-- 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]]
```

```haskell
-- ---------------------------------------------------------------------
-- 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))
```

3.5. Examen 5 (6 de mayo de 2015)

```haskell
-- 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']
```

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

```haskell
    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 )
```

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

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]
```

```haskell
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]
```

3.6. Examen 6 (15 de junio de 2015)

El examen es común con el del grupo 4 (ver página 137).

3.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 1 (ver página 42).

3.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 1 (ver página 50).

3.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 1 (ver página 61).

4

Exámenes del grupo 4

María J. Hidalgo

4.1. Examen 1 (6 de Noviembre de 2014)

```haskell
-- 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]]

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

```haskell
-- 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
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.
```

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

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

```haskell
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.
```

4.2. Examen 2 (4 de Diciembre de 2014)

```haskell
-- 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:
```

```haskell
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
```

```haskell
-- 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))
```

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]
```

```haskell
-- 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,
```

```haskell
--   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)
```

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
```

```haskell
--    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)))
--
```

```haskell
-- 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
```

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

```haskell
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):
```

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

```haskell
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
```

```haskell
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)
```

```haskell
-- 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.
```

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

```haskell
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]
```

```haskell
--
-- 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):
```

```haskell
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
```

```haskell
-- 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)
```

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

4.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 1 (ver página 42).

4.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 1 (ver página 42).

4.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 1 (ver página 61).

5

Exámenes del grupo 5

Francisco J. Martín

5.1. 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
```

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

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
```

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

-- 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
```

5.2. 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
```

```haskell
-- 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 _    = []
```

```haskell
-- 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
```

```haskell
-- 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
```

5.3. Examen 3 (23 de enero de 2015)

El examen es común con el del grupo 1 (ver página 22).

5.4. Examen 4 (16 de marzo de 2015)

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

```haskell
-- ---------------------------------------------------------------------
-- § 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
```

```haskell
-- 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
```

```haskell
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]
```

```haskell
-- 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
```

```haskell
-- 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]
```

```haskell
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)
```

5.5. Examen 5 (5 de mayo de 2015)

```haskell
-- 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
```

```haskell
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)]
```

```haskell
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))
```

```haskell
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]
```

```haskell
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
```

```haskell
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])
```

5.6. Examen 6 (15 de junio de 2015)

El examen es común con el del grupo 4 (ver página 137).

5.7. Examen 7 (3 de julio de 2015)

El examen es común con el del grupo 1 (ver página 42).

5.8. Examen 8 (4 de septiembre de 2015)

El examen es común con el del grupo 1 (ver página 50).

5.9. Examen 9 (4 de diciembre de 2015)

El examen es común con el del grupo 1 (ver página 61).

Apéndice A

Resumen de funciones predefinidas de Haskell

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

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

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

A.1. Resumen de funciones sobre TAD en Haskell

A.1.1. Polinomios

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

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

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

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

A.1.3. Tablas

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

A.1.4. Grafos

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

Apéndice B

Método de Pólya para la resolución de problemas

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

Para resolver un problema se necesita:

Paso 1: Entender el problema

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

Paso 2: Configurar un plan

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

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

Paso 3: Ejecutar el plan

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

Paso 4: Examinar la solución obtenida

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

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

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

Para resolver un problema se necesita:

Paso 1: Entender el problema

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

Paso 2: Diseñar el programa

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

Paso 3: Escribir el programa

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

Paso 4: Examinar la solución obtenida

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

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

Bibliografía

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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