Exámenes de "Programación funcional con Haskell"

Vol. 7 (Curso 2015–16)

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 2016

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 2015–16) 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 2015–16) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2015–16) [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 7º 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-15/temas/2015-16-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-15/ejercicios/ejercicios-I1M-2015.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol7
[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]
- Exámenes de “Programación funcional con Haskell”. Vol. 6 (Curso 2014–15) [10]

José A. Alonso
Sevilla, 20 de diciembre de 2016

[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
[10] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol6

1

Exámenes del grupo 1

José A. Alonso y Luis Valencia

1.1. Examen 1 (6 de Noviembre de 2015)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (6 de noviembre de 2015)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    repiteElementos :: Int -> [a] -> [a]
-- tal que (repiteElementos k xs) es la lista obtenida repitiendo cada
-- elemento de xs k veces. Por ejemplo,
--    repiteElementos 3 [5,2,7,4]  ==  [5,5,5,2,2,2,7,7,7,4,4,4]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
repiteElementos1 :: Int -> [a] -> [a]
repiteElementos1 k xs = concat [replicate k x | x <- xs]

-- 2ª definición (con map)
repiteElementos2 :: Int -> [a] -> [a]
```

```haskell
repiteElementos2 k xs = concat (map (replicate k) xs)

-- 3ª definición (con concatMap):
repiteElementos3 :: Int -> [a] -> [a]
repiteElementos3 k = concatMap (replicate k)

-- 4ª definición (por recursión):
repiteElementos4 :: Int -> [a] -> [a]
repiteElementos4 k []     = []
repiteElementos4 k (x:xs) = replicate k x ++ repiteElementos4 k xs

-- 5ª definición (por plegado):
repiteElementos5 :: Int -> [a] -> [a]
repiteElementos5 k = foldr ((++) . replicate k) []

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que, para todo número natural
-- k y toda lista xs, el número de elementos de (repiteElementos k xs)
-- es k veces el número de elementos de xs.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=7}) prop_repiteElementos
-- ---------------------------------------------------------------------

-- La propiedad es
prop_repiteElementos :: Int -> [Int] -> Property
prop_repiteElementos k xs =
    k >= 0 ==> length (repiteElementos1 k xs) == k * length xs

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_repiteElementos
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. Todo número entero positivo n se puede escribir como
-- 2^k*m, con m impar. Se dice que m es la parte impar de n. Por
-- ejemplo, la parte impar de 40 es 5 porque 40 = 5*2^3.
--
-- Definir la función
```

```
--    parteImpar :: Integer -> Integer
-- tal que (parteImpar n) es la parte impar de n. Por ejemplo,
--    parteImpar 40  ==  5
-- ---------------------------------------------------------------------

parteImpar :: Integer -> Integer
parteImpar n | even n    = parteImpar (n `div` 2)
             | otherwise = n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    refinada :: [Float] -> [Float]
-- tal que (refinada xs) es la lista obtenida intercalando entre cada
-- dos elementos consecutivos de xs su media aritmética. Por ejemplo,
--    refinada [2,7,1,8]  ==  [2.0,4.5,7.0,4.0,1.0,4.5,8.0]
--    refinada [2]        ==  [2.0]
--    refinada []         ==  []
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
refinada :: [Float] -> [Float]
refinada (x:y:zs) = x : (x+y)/2 : refinada (y:zs)
refinada xs       = xs

-- 2ª definición (por comprensión);
refinada2 :: [Float] -> [Float]
refinada2 []     = []
refinada2 (x:xs) = x : concat [[(a+b)/2,b] | (a,b) <- zip (x:xs) xs]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se dice que en una sucesión de números x(1),x(2),...,x(n)
-- hay una inversión cuando existe un par de números x(i) > x(j), siendo
-- i < j. Por ejemplo, en la sucesión 2, 1, 4, 3 hay dos inversiones (2
-- antes que 1 y 4 antes que 3) y en la sucesión 4, 3, 1, 2 hay cinco
-- inversiones (4 antes 3, 4 antes 1, 4 antes 2, 3 antes 1, 3 antes 2).
--
-- Definir la función
--    numeroInversiones :: Ord a => [a] -> Int
-- tal que (numeroInversiones xs) es el número de inversiones de xs. Por
-- ejemplo,
```

```haskell
--    numeroInversiones [2,1,4,3]  ==  2
--    numeroInversiones [4,3,1,2]  ==  5
-- ---------------------------------------------------------------------

-- 1ª solución (por recursión)
numeroInversiones1 :: Ord a => [a] -> Int
numeroInversiones1 [] = 0
numeroInversiones1 (x:xs) =
    length [y | y <- xs, y < x] + numeroInversiones1 xs

-- 2ª solución (por comprensión)
numeroInversiones2 :: Ord a => [a] -> Int
numeroInversiones2 xs =
    length [(i,j) | i <- [0..n-2], j <- [i+1..n-1], xs!!i > xs!!j]
    where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir la función
--    producto :: [[a]] -> [[a]]
-- tal que (producto xss) es el producto cartesiano de los conjuntos xss.
-- Por ejemplo,
--    ghci> producto [[1,3],[2,5]]
--    [[1,2],[1,5],[3,2],[3,5]]
--    ghci> producto [[1,3],[2,5],[6,4]]
--    [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
--    ghci> producto [[1,3,5],[2,4]]
--    [[1,2],[1,4],[3,2],[3,4],[5,2],[5,4]]
--    ghci> producto []
--    [[]]
--    ghci> producto [[x] | x <- [1..10]]
--    [[1,2,3,4,5,6,7,8,9,10]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que el número de elementos de
-- (producto xss) es el producto de los números de elementos de los
```

```
-- elementos de xss.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=7}) prop_producto
-- ---------------------------------------------------------------------

-- La propiedad es
prop_producto :: [[Int]] -> Bool
prop_producto xss =
    length (producto xss) == product [length xs | xs <- xss]

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_producto
--    +++ OK, passed 100 tests.
```

1.2. Examen 2 (4 de Diciembre de 2015)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (4 de diciembre de 2015)
-- ---------------------------------------------------------------------

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

import Data.Numbers.Primes
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    listasDecrecientesDesde :: Int -> [[Int]]
-- tal que (listasDecrecientesDesde n) es la lista de las sucesiones
-- estrictamente decrecientes cuyo primer elemento es n. Por ejemplo,
--    ghci> listasDecrecientesDesde 2
--    [[2],[2,1],[2,1,0],[2,0]]
--    ghci> listasDecrecientesDesde 3
--    [[3],[3,2],[3,2,1],[3,2,1,0],[3,2,0],[3,1],[3,1,0],[3,0]]
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución
listasDecrecientesDesde :: Int -> [[Int]]
listasDecrecientesDesde 0 = [[0]]
listasDecrecientesDesde n =
    [n] : [n:ys | m <- [n-1,n-2..0], ys <- listasDecrecientesDesde m]

-- 2ª solución
listasDecrecientesDesde2 :: Int -> [[Int]]
listasDecrecientesDesde2 n =
    [n : xs | xs <- subsequences [n-1,n-2..0]]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una propiedad del 2015 es que la suma de sus dígitos
-- coincide con el número de sus divisores; en efecto, la suma de sus
-- dígitos es 2+0+1+5=8 y tiene 8 divisores (1, 5, 13, 31, 65, 155, 403
-- y 2015).
--
-- Definir la sucesión
--    especiales :: [Int]
-- formada por los números n tales que la suma de los dígitos de n
-- coincide con el número de divisores de n. Por ejemplo,
--    take 12 especiales == [1,2,11,22,36,84,101,152,156,170,202,208]
--
-- Usando la sucesión, calcular cuál será el siguiente al 2015 que
-- cumplirá la propiedad.
-- ---------------------------------------------------------------------

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 del siguiente al 2015 que cumplirá la propiedad es
--    ghci> head (dropWhile (<=2015) especiales)
--    2101
```

```
-- ---------------------------------------------------------------------
-- 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. Un número x cumple la propiedad de Roldán si la suma de
-- x y el primo que sigue a x es un número primo. Por ejemplo, el número
-- 8 es un número de Roldán porque su siguiente primo es 11 y
-- 8+11=19 es primo. El 12 no es un número de Roldán porque su siguiente
-- primo es 13 y 12+13=25 no es primo.
--
-- Definir la sucesión
--     roldanes :: [Integer]
```

```haskell
-- cuyo elementos son los números de Roldán. Por ejemplo,
--    ghci> take 20 roldanes
--    [0,1,2,6,8,14,18,20,24,30,34,36,38,48,50,54,64,68,78,80]
--    ghci> roldanes3 !! 2015
--    18942
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

roldanes :: [Integer]
roldanes = 0: 1: [x | x <- [2,4..], primo (x + siguientePrimo x)]

primo :: Integer -> Bool
primo x = [y | y <- [1..x], x `rem` y == 0] == [1,x]

siguientePrimo :: Integer -> Integer
siguientePrimo x = head [y | y <- [x+1..], primo y]

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

roldanes2 :: [Integer]
roldanes2 = 0 : 1 : 2: aux [2,4..] primos where
    aux (x:xs) (y:ys)
        | y < x                = aux (x:xs) ys
        | (x+y) `pertenece` ys = x : aux xs (y:ys)
        | otherwise            = aux xs (y:ys)
    pertenece x ys = x == head (dropWhile (<x) ys)

primos :: [Integer]
primos = 2 : [x | x <- [3,5..], primo x]

-- 3ª definición (con la librería de primos)
-- =========================================

roldanes3 :: [Integer]
roldanes3 = 0: 1: [x | x <- [2,4..], isPrime (x + siguientePrimo3 x)]

siguientePrimo3 x = head [y | y <- [x+1..], isPrime y]
```

```haskell
-- 4ª definición (por recursión con la librería de primos)
-- =======================================================

roldanes4 :: [Integer]
roldanes4 = 0 : 1 : 2: aux [2,4..] primes where
    aux (x:xs) (y:ys)
        | y < x                = aux (x:xs) ys
        | (x+y) `pertenece` ys = x : aux xs (y:ys)
        | otherwise            = aux xs (y:ys)
    pertenece x ys = x == head (dropWhile (<x) ys)

-- 5ª definición
-- =============

roldanes5 :: [Integer]
roldanes5 = [a | q <- primes,
                 let p = siguientePrimo3 (q `div` 2),
                 let a = q-p,
                 siguientePrimo3 a == p]

-- 6ª definición
-- =============

roldanes6 :: [Integer]
roldanes6 = [x | (x,y) <- zip [0..] ps, isPrime (x+y)]
    where ps = 2:2:concat (zipWith f primes (tail primes))
          f p q = genericReplicate (q-p) q

-- 7ª definición
-- =============

roldanes7 :: [Integer]
roldanes7 = 0:1:(aux primes (tail primes) primes)
    where aux (x:xs) (y:ys) zs
              | null rs   = aux xs ys zs2
              | otherwise = [r-y | r <- rs] ++ (aux xs ys zs2)
              where a = x+y
                    b = 2*y-1
                    zs1 = takeWhile (<=b) zs
```

```
                    rs = [r | r <- [a..b], r `elem` zs1]
                    zs2 = dropWhile (<=b) zs

-- Comparación de eficiencia                                          --
--    ghci> :set +s
--
--    ghci> roldanes !! 700
--    5670
--    (12.72 secs, 1245938184 bytes)
--
--    ghci> roldanes2 !! 700
--    5670
--    (8.01 secs, 764775268 bytes)
--
--    ghci> roldanes3 !! 700
--    5670
--    (0.22 secs, 108982640 bytes)
--
--    ghci> roldanes4 !! 700
--    5670
--    (0.20 secs, 4707384 bytes)
--
--    ghci> roldanes5 !! 700
--    5670
--    (0.17 secs, 77283064 bytes)
--
--    ghci> roldanes6 !! 700
--    5670
--    (0.08 secs, 31684408 bytes)
--
--    ghci> roldanes7 !! 700
--    5670
--    (0.03 secs, 4651576 bytes)
--
--    ghci> roldanes3 !! 2015
--    18942
--    (1.78 secs, 1,065,913,952 bytes)
--
--    ghci> roldanes4 !! 2015
--    18942
```

```
--    (2.85 secs, 0 bytes)
--
--    ghci> roldanes5 !! 2015
--    18942
--    (1.17 secs, 694,293,520 bytes)
--
--    ghci> roldanes6 !! 2015
--    18942
--    (0.48 secs, 248,830,328 bytes)
--
--    ghci> roldanes7 !! 2015
--    18942
--    (0.11 secs, 0 bytes)
```

1.3. Examen 3 (25 de enero de 2016)

```
-- Informática: 3º examen de evaluación continua (25 de enero de 2016)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1. Los máximos y mínimos de una función son sus valores
-- óptimos respecto de las relaciones > y <, respectivamente. Por
-- ejemplo, para la lista xs = ["ab","c","de","f"], la función longitud
-- alcanza sus valores máximos (es decir, óptimos respecto >) en "ab" y
-- "de" (que son los elementos de xs de mayor longitud) y alcanza sus
-- valores mínimos (es decir, óptimos respecto <) en "c" y "f" (que son
-- los elementos de xs de menor longitud).
--
-- Definir la función
--    optimos :: Eq b => (b -> b -> Bool) -> (a -> b) -> [a] -> [a]
-- tal que (optimos r f xs) es la lista de los elementos de xs donde la
-- función f alcanza sus valores óptimos respecto de la relación r. Por
-- ejemplo,
--    optimos (>) length ["ab","c","de","f"]  ==  ["ab","de"]
--    optimos (<) length ["ab","c","de","f"]  ==  ["c","f"]
```

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

optimos :: Eq a => (b -> b -> Bool) -> (a -> b) -> [a] -> [a]
optimos r f xs =
    [x | x <- xs, null [y | y <- xs, x /= y, r (f y) (f x)]]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles se pueden representar mediante el siguiente
-- tipo de datos
--    data Arbol a = N a [Arbol a]
--                 deriving Show
-- Por ejemplo, los árboles
--      1              3
--     / \            /|\
--    2   3          / | \
--        |         5  4  7
--        4         |     /\
--                  6    2  1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]]
--
-- Definir la función
--    mayorProducto :: (Ord a, Num a) => Arbol a -> a
-- tal que (mayorProducto a) es el mayor producto de las ramas del árbol
-- a. Por ejemplo,
--    ghci> mayorProducto (N 1 [N 2 [], N  3 []])
--    3
--    ghci> mayorProducto (N 1 [N 8 [], N  4 [N 3 []]])
--    12
--    ghci> mayorProducto (N 1 [N 2 [],N 3 [N 4 []]])
--    12
--    ghci> mayorProducto (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
--    90
-- ---------------------------------------------------------------------

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

```haskell
-- 1º definición
mayorProducto1 :: (Ord a, Num a) => Arbol a -> a
mayorProducto1 (N x []) = x
mayorProducto1 (N x xs) = x * maximum [mayorProducto1 a | a <- xs]

-- Se puede usar map en lugar de comprensión:
mayorProducto1a :: (Ord a, Num a) => Arbol a -> a
mayorProducto1a (N x []) = x
mayorProducto1a (N x xs) = x * maximum (map mayorProducto1a xs)

-- 2ª definición
mayorProducto2 :: (Ord a, Num a) => Arbol a -> a
mayorProducto2 a = maximum [product xs | xs <- ramas a]

-- (ramas a) es la lista de las ramas del árbol a. Por ejemplo,
--    ghci> ramas (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
--    [[3,5,6],[3,4],[3,7,2],[3,7,1]]
ramas :: Arbol b -> [[b]]
ramas (N x []) = [[x]]
ramas (N x as) = [x : xs | a <- as, xs <- ramas a]

-- En la definición de mayorProducto2 se puede usar map en lugar de
-- comprensión.
mayorProducto2a :: (Ord a, Num a) => Arbol a -> a
mayorProducto2a a = maximum (map product (ramas a))

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la sucesión
--    sumasDeDosPrimos :: [Integer]
-- cuyos elementos son los números que se pueden escribir como suma de
-- dos números primos. Por ejemplo,
--    ghci> take 20 sumasDeDosPrimos
--    [4,5,6,7,8,9,10,12,13,14,15,16,18,19,20,21,22,24,25,26]
--    ghci> sumasDeDosPrimos !! 2016
--    3146
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============
```

```haskell
sumasDeDosPrimos1 :: [Integer]
sumasDeDosPrimos1 =
    [n | n <- [1..], not (null (sumaDeDosPrimos1 n))]

-- (sumasDeDosPrimos1 n) es la lista de pares de primos cuya suma es
-- n. Por ejemplo,
--    sumaDeDosPrimos  9  ==  [(2,7),(7,2)]
--    sumaDeDosPrimos 16  ==  [(3,13),(5,11),(11,5),(13,3)]
--    sumaDeDosPrimos 17  ==  []
sumaDeDosPrimos1 :: Integer -> [(Integer,Integer)]
sumaDeDosPrimos1 n =
    [(x,n-x) | x <- primosN, isPrime (n-x)]
    where primosN = takeWhile (< n) primes

-- 2ª definición
-- =============

sumasDeDosPrimos2 :: [Integer]
sumasDeDosPrimos2 =
    [n | n <- [1..], not (null (sumaDeDosPrimos2 n))]

-- (sumasDeDosPrimos2 n) es la lista de pares (x,y) de primos cuya suma
-- es n y tales que x <= y. Por ejemplo,
--    sumaDeDosPrimos2  9  ==  [(2,7)]
--    sumaDeDosPrimos2 16  ==  [(3,13),(5,11)]
--    sumaDeDosPrimos2 17  ==  []
sumaDeDosPrimos2 :: Integer -> [(Integer,Integer)]
sumaDeDosPrimos2 n =
    [(x,n-x) | x <- primosN, isPrime (n-x)]
    where primosN = takeWhile (<= (n `div` 2)) primes

-- 3ª definición
-- =============

sumasDeDosPrimos3 :: [Integer]
sumasDeDosPrimos3 = filter esSumaDeDosPrimos3 [4..]

-- (esSumaDeDosPrimos3 n) se verifica si n es suma de dos primos. Por
-- ejemplo,
--    esSumaDeDosPrimos3  9  ==  True
```

```
--    esSumaDeDosPrimos3 16  ==  True
--    esSumaDeDosPrimos3 17  ==  False
esSumaDeDosPrimos3 :: Integer -> Bool
esSumaDeDosPrimos3 n
    | odd n     = isPrime (n-2)
    | otherwise = any isPrime [n-x | x <- takeWhile (<= (n `div` 2)) primes]

-- 4ª definición
-- =============

-- Usando la conjetura de Goldbach que dice que "Todo número par mayor
-- que 2 puede escribirse como suma de dos números primos" .

sumasDeDosPrimos4 :: [Integer]
sumasDeDosPrimos4 = filter esSumaDeDosPrimos4 [4..]

-- (esSumaDeDosPrimos4 n) se verifica si n es suma de dos primos. Por
-- ejemplo,
--    esSumaDeDosPrimos4  9  ==  True
--    esSumaDeDosPrimos4 16  ==  True
--    esSumaDeDosPrimos4 17  ==  False
esSumaDeDosPrimos4 :: Integer -> Bool
esSumaDeDosPrimos4 n = even n || isPrime (n-2)

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

--    ghci> sumasDeDosPrimos1 !! 3000
--    4731
--    (6.66 secs, 3,278,830,304 bytes)
--    ghci> sumasDeDosPrimos2 !! 3000
--    4731
--    (3.69 secs, 1,873,984,088 bytes)
--    ghci> sumasDeDosPrimos3 !! 3000
--    4731
--    (0.35 secs, 175,974,016 bytes)
--    ghci> sumasDeDosPrimos4 !! 3000
--    4731
--    (0.07 secs, 18,396,432 bytes)
--
```

```
--    ghci> sumasDeDosPrimos3 !! 30000
--    49785
--    (6.65 secs, 3,785,736,416 bytes)
--    ghci> sumasDeDosPrimos4 !! 30000
--    49785
--    (1.06 secs, 590,767,736 bytes)

-- En lo que sigue usaremos la 1ª definición
sumasDeDosPrimos :: [Integer]
sumasDeDosPrimos = sumasDeDosPrimos1

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir el procedimiento
--    termino :: IO ()
-- que pregunta por una posición y escribe el término de la sucesión
-- sumasDeDosPrimos en dicha posición. Por ejemplo,
--    ghci> termino
--    Escribe la posicion: 5
--    El termino en la posicion 5 es 9
--    ghci> termino
--    Escribe la posicion: 19
--    El termino en la posicion 19 es 26
-- ---------------------------------------------------------------------

termino :: IO ()
termino = do
  putStr "Escribe la posicion: "
  xs <- getLine
  let n = read xs
  putStr "El termino en la posicion "
  putStr xs
  putStr " es "
  putStrLn (show (sumasDeDosPrimos !! n))

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Una función f entre dos conjuntos A e B se puede
-- representar mediante una lista de pares de AxB tales que para cada
-- elemento a de A existe un único elemento b de B tal que (a,b)
-- pertenece a f. Por ejemplo,
--    + [(1,2),(3,6)] es una función de [1,3] en [2,4,6];
```

```
--    + [(1,2)] no es una función de [1,3] en [2,4,6], porque no tiene
--      ningún par cuyo primer elemento sea igual a 3;
--    + [(1,2),(3,6),(1,4)] no es una función porque hay dos pares
--      distintos cuya primera coordenada es 1.
--
-- Definir la función
--    funciones :: [a] -> [a] -> [[(a,a)]]
-- tal que (funciones xs ys) es el conjunto de las funciones de xs en
-- ys. Por ejemplo,
--    ghci> funciones [] [2,4,6]
--    [[]]
--    ghci> funciones [3] [2,4,6]
--    [[(3,2)],[(3,4)],[(3,6)]]
--    ghci> funciones [1,3] [2,4,6]
--    [[(1,2),(3,2)], [(1,2),(3,4)], [(1,2),(3,6)], [(1,4),(3,2)], [(1,4),(3,4)],
--     [(1,4),(3,6)], [(1,6),(3,2)], [(1,6),(3,4)], [(1,6),(3,6)]]
-- ---------------------------------------------------------------------

funciones :: [a] -> [a] -> [[(a,a)]]
funciones [] _      = [[]]
funciones [x] ys    = [[(x,y)] | y <- ys]
funciones (x:xs) ys = [((x,y):f) | y <- ys, f <- fs]
    where fs = funciones xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que si xs es un conjunto con n
-- elementos e ys un conjunto con m elementos, entonces (funciones xs ys)
-- tiene m^n elementos.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_funciones
--    +++ OK, passed 100 tests.
-- ---------------------------------------------------------------------

prop_funciones :: [Int] -> [Int] -> Bool
prop_funciones xs ys =
    length (funciones xs ys) == (length ys)^(length xs)
```

1.4. Examen 4 (16 de marzo de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 4º examen de evaluación continua (16 de marzo de 2016)
-- ---------------------------------------------------------------------

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

import Data.Array
import Data.Char
import Data.Numbers.Primes
import qualified Data.Matrix as M

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    siguiente :: Eq a => a -> [a] -> Maybe a
-- tal que (siguiente x ys) es justo el elemento siguiente a la primera
-- ocurrencia de x en ys o Nothing si x no pertenece a ys. Por ejemplo,
--    siguiente 5 [3,5,2,5,7]                       ==  Just 2
--    siguiente 9 [3,5,2,5,7]                       ==  Nothing
--    siguiente 'd' "afdegdb"                       ==  Just 'e'
--    siguiente "todo" ["En","todo","la","medida"]  ==  Just "la"
--    siguiente "nada" ["En","todo","la","medida"]  ==  Nothing
--    siguiente 999999 [1..1000000]                 ==  Just 1000000
--    siguiente 1000000 [1..1000000]                ==  Nothing
-- ---------------------------------------------------------------------

-- 1ª solución (por recursión):
siguiente1 :: Eq a => a -> [a] -> Maybe a
siguiente1 x (y1:y2:ys) | x == y1   = Just y2
                        | otherwise = siguiente1 x (y2:ys)
siguiente1 x _ = Nothing

-- 2ª solución (por comprensión):
siguiente2 :: Eq a => a -> [a] -> Maybe a
siguiente2 x ys
    | null zs   = Nothing
    | otherwise = Just (snd (head zs))
```

```
    where zs = [(u,v) | (u,v) <- zip ys (tail ys), u == x]

-- 3ª solución (con dropWhile)
siguiente3 :: Eq a => a -> [a] -> Maybe a
siguiente3 x = aux . drop 1 . dropWhile (/=x)
    where aux []    = Nothing
          aux (y:_) = Just y

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

-- La comparación es
--    ghci> let n=10^6 in siguiente1 (n-1) [1..n]
--    Just 1000000
--    (1.34 secs, 277352616 bytes)
--
--    ghci> let n=10^6 in siguiente2 (n-1) [1..n]
--    Just 1000000
--    (1.45 secs, 340836576 bytes)
--
--    ghci> let n=10^6 in siguiente3 (n-1) [1..n]
--    Just 1000000
--    (0.26 secs, 84987544 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un número n es k-belga si la sucesión cuyo primer
-- elemento es k y cuyos elementos se obtienen sumando reiteradamente
-- los dígitos de n contiene a n. Por ejemplo,
-- + El 18 es 0-belga, porque a partir del 0 vamos a ir sumando
--   sucesivamente 1, 8, 1, 8, ... hasta llegar o sobrepasar el 18: 0, 1,
--   9, 10, 18, ... Como se alcanza el 18, resulta que el 18 es 0-belga.
-- + El 19 no es 1-belga, porque a partir del 1 vamos a ir sumando
--   sucesivamente 1, 9, 1, 9, ... hasta llegar o sobrepasar el 18: 1, 2,
--   11, 12, 21, 22, ... Como no se alcanza el 19, resulta que el 19 no es
--   1-belga.
--
-- Definir la función
--    esBelga :: Int -> Int -> Bool
-- tal que (esBelga k n)  se verifica si n es k-belga. Por ejemplo,
--    esBelga 0 18                                ==  True
```

```
--    esBelga 1 19                                  ==  False
--    esBelga 0 2016                                ==  True
--    [x | x <- [0..30], esBelga 7 x]               ==  [7,10,11,21,27,29]
--    [x | x <- [0..30], esBelga 10 x]              ==  [10,11,20,21,22,24,26]
--    length [n | n <- [1..9000], esBelga 0 n]      ==  2857
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

esBelga1 :: Int -> Int -> Bool
esBelga1 k n =
    n == head (dropWhile (<n) (scanl (+) k (cycle (digitos n))))

digitos :: Int -> [Int]
digitos n = map digitToInt (show n)

-- 2ª solución
-- ===========

esBelga2 :: Int -> Int -> Bool
esBelga2 k n =
    k <= n && n == head (dropWhile (<n) (scanl (+) (k + q * s) ds))
    where ds = digitos n
          s  = sum ds
          q  = (n - k) `div` s

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

--    ghci> length [n | n <- [1..9000], esBelga1 0 n]
--    2857
--    (2.95 secs, 1,115,026,728 bytes)
--    ghci> length [n | n <- [1..9000], esBelga2 0 n]
--    2857
--    (0.10 secs, 24,804,480 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios con datos en los nodos y hojas se
-- definen por
```

```
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Eq, 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))
--
-- Anotando cada elemento del árbol anterior con su profundidad, se
-- obtiene el árbol siguiente
--           3-0
--           / \
--          /   \
--         /     \
--       4-1     7-1
--       / \     / \
--     5-2 0-2 0-2 3-2
--     / \
--   2-3 0-3
--
-- Definir la función
--    anotado :: Arbol a -> Arbol (a,Int)
-- tal que (anotado x) es el árbol obtenido anotando los elementos de x
-- con su profundidad. Por ejemplo,
--    ghci> anotado ejArbol
--    N (3,0)
--      (N (4,1)
--         (N (5,2) (H (2,3)) (H (0,3)))
--         (H (0,2)))
--      (N (7,1) (H (0,2)) (H (3,2)))
-- ----------------------------------------------------------------------
```

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

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

-- 1ª solución
-- ===========

anotado1 :: Arbol a -> Arbol (a,Int)
anotado1 (H x)     = H (x,0)
anotado1 (N x i d) = aux (N x i d) 0
    where aux (H x)     n = H (x,n)
          aux (N x i d) n = N (x,n) (aux i (n+1)) (aux d (n+1))

-- 2ª solución
anotado2 :: Arbol a -> Arbol (a, Int)
anotado2 a = aux a [0..]
    where aux (H a)     (n:_ ) = H (a,n)
          aux (N a i d) (n:ns) = N (a,n) (aux i ns) (aux d ns)

-- ---------------------------------------------------------------------
-- Ejercicio 4. El pasado 11 de marzo se ha publicado el artículo
-- "Unexpected biases in the distribution of consecutive primes" en el
-- que muestra que los números primos repelen a otros primos que
-- terminan en el mismo dígito.
--
-- La lista de los últimos dígitos de los 30 primeros números es
--    [2,3,5,7,1,3,7,9,3,9,1,7,1,3,7,3,9,1,7,1,3,9,3,9,7,1,3,7,9,3]
-- Se observa que hay 6 números que su último dígito es un 1 y de sus
-- consecutivos 4 terminan en 3 y 2 terminan en 7.
--
-- Definir la función
--    distribucionUltimos :: Int -> M.Matrix Int
-- tal que (distribucionUltimos n) es la matriz cuyo elemento (i,j)
-- indica cuántos de los n primeros números primos terminan en i y su
-- siguiente número primo termina en j. Por ejemplo,
--    ghci> distribucionUltimos 30
--    ( 0 0 4 0 0 0 2 0 0 )
```

```
--    ( 0 0 1 0 0 0 0 0 0 )
--    ( 0 0 0 0 1 0 4 0 4 )
--    ( 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 )
--    ( 4 0 1 0 0 0 0 0 2 )
--    ( 0 0 0 0 0 0 0 0 0 )
--    ( 2 0 3 0 0 0 1 0 0 )
--
--    ghci> distribucionUltimos (10^5)
--    ( 4104    0 7961    0    0    0 8297    0 4605 )
--    (    0    0    1    0    0    0    0    0    0 )
--    ( 5596    0 3604    0    1    0 7419    0 8387 )
--    (    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 )
--    ( 6438    0 6928    0    0    0 3627    0 8022 )
--    (    0    0    0    0    0    0    0    0    0 )
--    ( 8830    0 6513    0    0    0 5671    0 3995 )
--
-- Nota: Se observa cómo se "repelen" ya que en las filas del 1, 3, 7 y
-- 9 el menor elemento es el de la diagonal.
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

distribucionUltimos1 :: Int -> M.Matrix Int
distribucionUltimos1 n =
    M.matrix 9 9
             (\(i,j) -> length (filter (==(i,j)) (take n ultimosConsecutivos)))

-- (ultimo n) es el último dígito de n.
ultimo :: Int -> Int
ultimo n = n `mod` 10

-- ultimos es la lista de el último dígito de los primos.
--    ghci> take 20 ultimos
--    [2,3,5,7,1,3,7,9,3,9,1,7,1,3,7,3,9,1,7,1]
ultimos :: [Int]
```

```haskell
ultimos = map ultimo primes

-- ultimosConsecutivos es la lista de los últimos dígitos de los primos
-- consecutivos.
--    ghci> take 10 ultimosConsecutivos
--    [(2,3),(3,5),(5,7),(7,1),(1,3),(3,7),(7,9),(9,3),(3,9),(9,1)]
ultimosConsecutivos :: [(Int,Int)]
ultimosConsecutivos = zip ultimos (tail ultimos)

-- 2ª solución
-- ===========

distribucionUltimos2 :: Int -> M.Matrix Int
distribucionUltimos2 n =
    M.fromList 9 9
               (elems (histograma ((1,1),(9,9)) (take n ultimosConsecutivos)))

-- (histograma r is) es el vector formado contando cuantas veces
-- aparecen los elementos del rango r en la lista de índices is. Por
-- ejemplo,
--    ghci> histograma (0,5) [3,1,4,1,5,4,2,7]
--    array (0,5) [(0,0),(1,2),(2,1),(3,1),(4,2),(5,1)]
histograma :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
histograma r is =
    accumArray (+) 0 r [(i,1) | i <- is, inRange r i]

-- 3ª definición
-- =============

distribucionUltimos3 :: Int -> M.Matrix Int
distribucionUltimos3 n
    | n < 4 = distribucionUltimos1 n
    | otherwise =  M.matrix 9 9 (\(i,j) -> f i j)
    where f i j | elem (i,j) [(2,3),(3,5),(5,7)] = 1
                | even i || even j = 0
                | otherwise = length (filter (==(i,j))
                                             (take n ultimosConsecutivos))

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

```
--    ghci> distribucionUltimos1 (10^5)
--    ( 4104    0 7961    0    0    0 8297    0 4605 )
--    (    0    0    1    0    0    0    0    0    0 )
--    ( 5596    0 3604    0    1    0 7419    0 8387 )
--    (    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 )
--    ( 6438    0 6928    0    0    0 3627    0 8022 )
--    (    0    0    0    0    0    0    0    0    0 )
--    ( 8830    0 6513    0    0    0 5671    0 3995 )
--
--    (3.51 secs, 941,474,520 bytes)
--    ghci> distribucionUltimos2 (10^5)
--    ( 4104    0 7961    0    0    0 8297    0 4605 )
--    (    0    0    1    0    0    0    0    0    0 )
--    ( 5596    0 3604    0    1    0 7419    0 8387 )
--    (    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 )
--    ( 6438    0 6928    0    0    0 3627    0 8022 )
--    (    0    0    0    0    0    0    0    0    0 )
--    ( 8830    0 6513    0    0    0 5671    0 3995 )
--
--    (1.75 secs, 560,891,792 bytes)
--    ghci> distribucionUltimos3 (10^5)
--    ( 4104    0 7961    0    0    0 8297    0 4605 )
--    (    0    0    1    0    0    0    0    0    0 )
--    ( 5596    0 3604    0    1    0 7419    0 8387 )
--    (    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 )
--    ( 6438    0 6928    0    0    0 3627    0 8022 )
--    (    0    0    0    0    0    0    0    0    0 )
--    ( 8830    0 6513    0    0    0 5671    0 3995 )
--
--    (1.70 secs, 623,371,360 bytes)
```

1.5. Examen 5 (4 de mayo de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 5º examen de evaluación continua (4 de mayo de 2016)
-- ---------------------------------------------------------------------

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

import Data.Numbers.Primes
import Data.Matrix
import Test.QuickCheck
import qualified Data.Set as S
import qualified Data.Map as M

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número entero positivo n es balanceado si n = 1 o
-- se puede escribir como un producto de un número par de factores
-- primos (no necesariamente distintos).
--
-- Definir la función
--    balanceado :: Int -> Bool
-- tal que (balanceado n) se verifica si n es balanceado. Por ejemplo,
--    balanceado 34  ==  True
--    balanceado 35  ==  True
--    balanceado 44  ==  False
-- ---------------------------------------------------------------------

balanceado :: Int -> Bool
balanceado 1 = True
balanceado n = n > 1 && even (length (primeFactors n))

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Un par (a,b) de enteros positivos es un balanceador
-- del número x si el producto (x+a)*(x+b) es balanceado.
--
-- Definir la función
--    balanceadores :: Int -> [(Int,Int)]
-- tal que (balanceadores x) es la lista de los balanceadores de x. Por
```

```
-- ejemplo,
--    take 5 (balanceadores 3)  ==  [(1,1),(1,3),(2,2),(3,1),(2,4)]
--    take 5 (balanceadores 5)  ==  [(1,1),(2,2),(1,4),(2,3),(3,2)]
-- ---------------------------------------------------------------------

balanceadores :: Int -> [(Int,Int)]
balanceadores x =
    [(a,b) | (a,b) <- enteros,
             balanceado ((x+a)*(x+b))]

--    ghci> take 10 enteros
--    [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1)]
enteros :: [(Int,Int)]
enteros = [(a,n-a) | n <- [1..],
                     a <- [1..n-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck si para cualquier entero
-- positivo x, la lista (balanceadores x) contiene a todos los pares
-- (a,a) con a mayor que 0.
-- ---------------------------------------------------------------------

prop_balanceadores :: Positive Int-> Int -> Property
prop_balanceadores (Positive x) a =
    a > 0 ==> balanceado ((x+a)*(x+a))

--    ghci> quickCheck prop_balanceadores
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un sistema de ecuaciones lineales Ax = b es triangular
-- inferior si todos los elementos de la matriz A que están por encima
-- de la diagonal principal son nulos; es decir, es de la forma
--    a(1,1)*x(1)                                                = b(1)
--    a(2,1)*x(1) + a(2,2)*x(2)                                  = b(2)
--    a(3,1)*x(1) + a(3,2)*x(2) + a(3,3)*x(3)                    = b(3)
--    ...
--    a(n,1)*x(1) + a(n,2)*x(2) + a(n,3)*x(3) +...+ a(x,x)*x(n) = b(n)
--
-- El sistema es compatible si, y sólo si, el producto de los elementos
```

```haskell
-- de la diagonal principal es distinto de cero. En este caso, la
-- solución se puede calcular mediante el algoritmo de bajada:
--    x(1) = b(1) / a(1,1)
--    x(2) = (b(2) - a(2,1)*x(1)) / a(2,2)
--    x(3) = (b(3) - a(3,1)*x(1) - a(3,2)*x(2)) / a(3,3)
--    ...
--    x(n) = (b(n) - a(n,1)*x(1) - a(n,2)*x(2) -...- a(n,n-1)*x(n-1)) / a(n,n)
--
-- Definir la función
--    bajada :: Matrix Double -> Matrix Double -> Matrix Double
-- tal que (bajada a b) es la solución, mediante el algoritmo de bajada,
-- del sistema compatible triangular superior ax = b. Por ejemplo,
--    ghci> let a = fromLists [[2,0,0],[3,1,0],[4,2,5.0]]
--    ghci> let b = fromLists [[3],[6.5],[10]]
--    ghci> bajada a b
--    ( 1.5 )
--    ( 2.0 )
--    ( 0.0 )
-- Es decir, la solución del sistema
--    2x            = 3
--    3x + y        = 6.5
--    4x + 2y + 5 z = 10
-- es x = 1.5, y = 2 y z = 0.
-- ---------------------------------------------------------------------

bajada :: Matrix Double -> Matrix Double -> Matrix Double
bajada a b = fromLists [[x i] | i <- [1..m]]
    where m = nrows a
          x k = (b!(k,1) - sum [a!(k,j) * x j | j <- [1..k-1]]) / a!(k,k)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    clasesEquivalencia :: Ord a =>
--                          S.Set a -> (a -> a -> Bool) -> S.Set (S.Set a)
-- tal que (clasesEquivalencia xs r) es la lista de las clases de
-- equivalencia de xs respecto de la relación de equivalencia r. Por
-- ejemplo,
--    ghci> let c = S.fromList [-3..3]
--    ghci> clasesEquivalencia c (\x y -> x `mod` 3 == y `mod` 3)
--    fromList [fromList [-3,0,3],fromList [-2,1],fromList [-1,2]]
```

```
--    ghci> clasesEquivalencia c (\x y -> (x - y) `mod` 2 == 0)
--    fromList [fromList [-3,-1,1,3],fromList [-2,0,2]]
-- ---------------------------------------------------------------------

clasesEquivalencia :: Ord a =>
                      S.Set a -> (a -> a -> Bool) -> S.Set (S.Set a)
clasesEquivalencia xs r
    | S.null xs =  S.empty
    | otherwise =  us `S.insert` clasesEquivalencia vs r
    where (y,ys)  = S.deleteFindMin xs
          (us,vs) = S.partition (r y) xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Los polinomios se pueden representar mediante
-- diccionarios con los exponentes como claves y los coeficientes como
-- valores. El tipo de los polinomios con coeficientes de tipo a se
-- define por
--    type Polinomio a = M.Map Int a
-- Dos ejemplos de polinomios (que usaremos en los ejemplos) son
--    3 + 7x - 5x^3
--    4 + 5x^3 + x^5
-- se definen por
--   ejPol1, ejPol2 :: Polinomio Int
--   ejPol1 = M.fromList [(0,3),(1,7),(3,-5)]
--   ejPol2 = M.fromList [(0,4),(3,5),(5,1)]
--
-- Definir la función
--    sumaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (sumaPol p q) es la suma de los polinomios p y q. Por ejemplo,
--   ghci> sumaPol ejPol1 ejPol2
--   fromList [(0,7),(1,7),(5,1)]
--   ghci> sumaPol ejPol1 ejPol1
--   fromList [(0,6),(1,14),(3,-10)]
-- ---------------------------------------------------------------------

type Polinomio a = M.Map Int a

ejPol1, ejPol2 :: Polinomio Int
ejPol1 = M.fromList [(0,3),(1,7),(3,-5)]
ejPol2 = M.fromList [(0,4),(3,5),(5,1)]
```

```haskell
sumaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
sumaPol p q =
    M.filter (/=0) (M.unionWith (+) p q)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--   multPorTerm :: Num a => (Int,a) -> Polinomio a -> Polinomio a
-- tal que (multPorTerm (n,a) p) es el producto del término ax^n por
-- p. Por ejemplo,
--   ghci> multPorTerm (2,3) (M.fromList [(0,4),(2,1)])
--   fromList [(2,12),(4,3)]
-- ---------------------------------------------------------------------

multPorTerm :: Num a => (Int,a) -> Polinomio a -> Polinomio a
multPorTerm (n,a) p =
    M.map (*a) (M.mapKeys (+n) p)

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--   multPol :: (Eq a, Num a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (multPol p q) es el producto de los polinomios p y q. Por
-- ejemplo,
--   ghci> multPol ejPol1 ejPol2
--   fromList [(0,12),(1,28),(3,-5),(4,35),(5,3),(6,-18),(8,-5)]
--   ghci> multPol ejPol1 ejPol1
--   fromList [(0,9),(1,42),(2,49),(3,-30),(4,-70),(6,25)]
--   ghci> multPol ejPol2 ejPol2
--   fromList [(0,16),(3,40),(5,8),(6,25),(8,10),(10,1)]
-- ---------------------------------------------------------------------

multPol :: (Eq a, Num a) => Polinomio a -> Polinomio a -> Polinomio a
multPol p q
    | M.null p  = M.empty
    | otherwise = sumaPol (multPorTerm t q) (multPol r q)
    where (t,r) = M.deleteFindMin p
```

1.6. Examen 6A (25 de mayo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 6º examen de evaluación continua (25 de mayo de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Matrix
import Data.Numbers.Primes
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    seccion :: Eq a => [a] -> [a]
-- tal que (seccion xs) es la mayor sección inicial de xs que no
-- contiene ningún elemento repetido. Por ejemplo:
--    seccion [1,2,3,2,4,5]                    == [1,2,3]
--    seccion "caceres"                        == "ca"
--    length (seccion ([1..7531] ++ [1..10^9]))  ==  7531
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

seccion1 :: Eq a => [a] -> [a]
seccion1 = last . filter (\ys -> nub ys == ys) . inits

-- 2ª solución
-- ===========

seccion2 :: Eq a => [a] -> [a]
seccion2 xs = aux xs []
    where aux [] ys = reverse ys
          aux (x:xs) ys | x `elem` ys = reverse ys
                        | otherwise   = aux xs (x:ys)
```

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

--    ghci> last (seccion1 [1..10^3])
--    1000
--    (6.19 secs, 59,174,640 bytes)
--    ghci> last (seccion2 [1..10^3])
--    1000
--    (0.04 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. 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 132  == False
-- ---------------------------------------------------------------------

esEspecial :: Integer -> Bool
esEspecial n =
    sort (show n) == sort (concatMap show (primeFactors n))

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. 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 2.3. Calcular los 5 primeros números especiales que no son
-- primos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> take 5 [n | n <- [2..], esEspecial n, not (isPrime n)]
--    [1255,12955,17482,25105,100255]

-- ---------------------------------------------------------------------
-- Ejercicio 3. 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
--         "B"
--         / \
--        /   \
--       /     \
--     "B"     "A"
--     / \     / \
--   "A" "B" "C" "C"
-- se puede definir por
--    ej1 :: Arbol String
--    ej1 = N "B" (N "B" (H "A") (H "B")) (N "A" (H "C") (H "C"))
--
-- Definir la función
--    enumeraArbol :: Arbol t -> Arbol Int
-- tal que (enumeraArbol a) es el árbol obtenido numerando las hojas y
-- los nodos de a desde la hoja izquierda hasta la raíz. Por ejemplo,
--    ghci> enumeraArbol ej1
--    N 6 (N 2 (H 0) (H 1)) (N 5 (H 3) (H 4))
-- Gráficamente,
--          6
--         / \
--        /   \
```

```haskell
--       /     \
--      2       5
--     / \     / \
--    0   1   3   4
-- ---------------------------------------------------------------------

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

ej1 :: Arbol String
ej1 = N "B" (N "B" (H "A") (H "B")) (N "A" (H "C") (H "C"))

enumeraArbol1 :: Arbol t -> Arbol Int
enumeraArbol1 a = fst (aux a 0)
    where aux :: Arbol a -> Int -> (Arbol Int,Int)
          aux (H _) n     = (H n, n+1)
          aux (N x i d) n = (N n2 i' d', 1+n2)
              where (i', n1) = aux i n
                    (d', n2) = aux d n1

-- ---------------------------------------------------------------------
-- Ejercicio 4. El buscaminas es un juego cuyo objetivo es despejar un
-- campo de minas sin detonar ninguna.
--
-- El campo de minas se representa mediante un cuadrado con NxN
-- casillas. Algunas casillas tienen un número, este número indica las
-- minas que hay en todas las casillas vecinas. Cada casilla tiene como
-- máximo 8 vecinas. Por ejemplo, el campo 4x4 de la izquierda
-- contiene dos minas, cada una representada por el número 9, y a la
-- derecha se muestra el campo obtenido anotando las minas vecinas de
-- cada casilla
--    9 0 0 0       9 1 0 0
--    0 0 0 0       2 2 1 0
--    0 9 0 0       1 9 1 0
--    0 0 0 0       1 1 1 0
-- de la misma forma, la anotación del siguiente a la izquierda es el de
-- la derecha
--    9 9 0 0 0     9 9 1 0 0
--    0 0 0 0 0     3 3 2 0 0
```

```
--    0 9 0 0 0     1 9 1 0 0
--
-- Utilizando la librería Data.Matrix, los campos de minas se
-- representan mediante matrices:
--    type Campo = Matrix Int
-- Por ejemplo, los anteriores campos de la izquierda se definen por
--    ejCampo1, ejCampo2 :: Campo
--    ejCampo1 = fromLists [[9,0,0,0],
--                          [0,0,0,0],
--                          [0,9,0,0],
--                          [0,0,0,0]]
--    ejCampo2 = fromLists [[9,9,0,0,0],
--                          [0,0,0,0,0],
--                          [0,9,0,0,0]]
--
-- Definir la función
--    buscaminas :: Campo -> Campo
-- tal que (buscaminas c) es el campo obtenido anotando las minas
-- vecinas de cada casilla. Por ejemplo,
--    ghci> buscaminas ejCampo1
--    ( 9 1 0 0 )
--    ( 2 2 1 0 )
--    ( 1 9 1 0 )
--    ( 1 1 1 0 )
--
--    ghci> buscaminas ejCampo2
--    ( 9 9 1 0 0 )
--    ( 3 3 2 0 0 )
--    ( 1 9 1 0 0 )
-- ---------------------------------------------------------------------

type Campo   = Matrix Int
type Casilla = (Int,Int)

ejCampo1, ejCampo2 :: Campo
ejCampo1 = fromLists [[9,0,0,0],
                      [0,0,0,0],
                      [0,9,0,0],
                      [0,0,0,0]]
ejCampo2 = fromLists [[9,9,0,0,0],
```

```
                     [0,0,0,0,0],
                     [0,9,0,0,0]]

-- 1ª solución
-- ===========

buscaminas1 :: Campo -> Campo
buscaminas1 c = matrix m n (\(i,j) -> minas c (i,j))
    where m = nrows c
          n = ncols c

-- (minas c (i,j)) es el número de minas en las casillas vecinas de la
-- (i,j) en el campo de mina c y es 9 si en (i,j) hay una mina. Por
-- ejemplo,
--    minas ejCampo (1,1)  ==  9
--    minas ejCampo (1,2)  ==  1
--    minas ejCampo (1,3)  ==  0
--    minas ejCampo (2,1)  ==  2
minas :: Campo -> Casilla -> Int
minas c (i,j)
    | c!(i,j) == 9 = 9
    | otherwise    = length (filter (==9)
                            [c!(x,y) | (x,y) <- vecinas m n (i,j)])
    where m = nrows c
          n = ncols c

-- (vecinas m n (i,j)) es la lista de las casillas vecinas de la (i,j) en
-- un campo de dimensiones mxn. Por ejemplo,
--    vecinas 4 (1,1)  ==  [(1,2),(2,1),(2,2)]
--    vecinas 4 (1,2)  ==  [(1,1),(1,3),(2,1),(2,2),(2,3)]
--    vecinas 4 (2,3)  ==  [(1,2),(1,3),(1,4),(2,2),(2,4),(3,2),(3,3),(3,4)]
vecinas :: Int -> Int -> Casilla -> [Casilla]
vecinas m n (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                             b <- [max 1 (j-1)..min n (j+1)],
                             (a,b) /= (i,j)]

-- 2ª solución
-- ===========

buscaminas2 :: Campo -> Campo
```

```haskell
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)]
```

1.7. Examen 6B (7 de junio de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (7 de junio de 2016)
-- ---------------------------------------------------------------------

-- Puntuación: Cada uno de los 5 ejercicios vale 2 puntos.

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

import Data.Numbers.Primes
import Data.List
import I1M.Grafo
import I1M.PolOperaciones
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    esSumaP :: Integer -> Integer -> Bool
-- tal que (esSumaP k n) se verifica si n es suma de k primos (no
-- necesariamente distintos). Por ejemplo,
--    esSumaP  3 10  ==  True
--    esSumaP  3  2  ==  False
--    esSumaP 10 21  ==  True
--    esSumaP 21 10  ==  False
```

```haskell
--    take 3 [n | n <- [1..], esSumaP 18 n]  == [36,37,38]
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

esSumaP1 :: Integer -> Integer -> Bool
esSumaP1 k n =
    n `elem` [sum xs | xs <- combinacionesR k (takeWhile (<=n) primes)]

-- (combinacionesR k xs) es la lista de las combinaciones orden k de los
-- elementos de xs con repeticiones. Por ejemplo,
--    ghci> combinacionesR 2 "abc"
--    ["aa","ab","ac","bb","bc","cc"]
--    ghci> combinacionesR 3 "bc"
--    ["bbb","bbc","bcc","ccc"]
--    ghci> combinacionesR 3 "abc"
--    ["aaa","aab","aac","abb","abc","acc","bbb","bbc","bcc","ccc"]
combinacionesR :: Integer -> [a] -> [[a]]
combinacionesR _ [] = []
combinacionesR 0 _  = [[]]
combinacionesR k (x:xs) =
    [x:ys | ys <- combinacionesR (k-1) (x:xs)] ++ combinacionesR k xs

-- 2ª definición
-- =============

esSumaP2 :: Integer -> Integer -> Bool
esSumaP2 k n = esSuma k n (takeWhile (<=n) primes)

esSuma :: Integer -> Integer -> [Integer] -> Bool
esSuma 0 x _  = x == 0
esSuma k 0 _  = k == 0
esSuma _ _ [] = False
esSuma k x (y:ys)
    | x < y     = False
    | otherwise = esSuma (k-1) (x-y) (y:ys) || esSuma k x ys

-- 3ª definición
-- =============
```

```haskell
esSumaP3 :: Integer -> Integer -> Bool
esSumaP3 1 n = isPrime n
esSumaP3 k n = any (esSumaP3 (k-1)) (map (n-) (takeWhile (<n) primes))

-- Equivalencia
-- ============

prop_equiv_esSumaP :: Positive Integer -> Bool
prop_equiv_esSumaP (Positive x) =
    esSumaP2 3 x == v
    && esSumaP2 3 x == v
    where v = esSumaP1 3 x

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

-- Eficiencia
-- ==========

--    ghci> [x | x <- [1..], esSumaP1 10 x] !! 700
--    720
--    (3.94 secs, 3,180,978,848 bytes)
--    ghci> [x | x <- [1..], esSumaP2 10 x] !! 700
--    720
--    (0.76 secs, 410,235,584 bytes)
--    ghci> [x | x <- [1..], esSumaP3 10 x] !! 700
--    720
--    (0.10 secs, 102,200,384 bytes)
--
--    ghci> take 3 [n | n <- [1..], esSumaP2 18 n]
--    [36,37,38]
--    (0.02 secs, 0 bytes)
--    ghci> take 3 [n | n <- [1..], esSumaP3 18 n]
--    [36,37,38]
--    (2.91 secs, 5,806,553,024 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que el menor número n para el
```

```haskell
-- que se cumple la condición (esSumaP k n) es 2*k.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_menor_esSuma
--    +++ OK, passed 100 tests.
-- ----------------------------------------------------------------------

-- La propiedad es
prop_menor_esSuma :: Integer -> Integer -> Property
prop_menor_esSuma k n =
    k > 0 && n > 0 ==>
      head [x | x <- [1..], esSumaP3 k x] == 2*k

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_menor_esSuma
--    +++ OK, passed 100 tests.
--    (0.03 secs, 0 bytes)

-- ----------------------------------------------------------------------
-- Ejercicio 2.1. Se consideran los árboles binarios representados
-- mediante el tipo Arbol definido por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                   deriving (Show,Eq)
-- Por ejemplo, el árbol
--         9
--        / \
--       /   \
--      3     8
--     / \
--    2   4
--       / \
--      1   5
-- se puede representar por
--    a1 :: Arbol Int
--    a1 = N 9 (N 3 (H 2) (N 4 (H 1) (H 5))) (H 8)
--
-- En el árbol a1 se cumple que todas sus ramas tiene un número par;
-- pero no se cumple que todas sus ramas tenga un número primo.
```

```haskell
--
-- Definir la función
--    propiedadAE :: (a -> Bool) -> Arbol a -> Bool
-- tal que (propiedadAE p a) se verifica si en todas las ramas de a hay
-- algún nodo que cumple la propiedad p. Por ejemplo,
--    propiedadAE even a1  == True
--    propiedadAE (<7) 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)

-- 1ª definición
propiedadAE :: (a -> Bool) -> Arbol a -> Bool
propiedadAE p (H x)     = p x
propiedadAE p (N x i d) = p x || (propiedadAE p i && propiedadAE p d)

-- 2ª definición
propiedadAE2 :: (a -> Bool) -> Arbol a -> Bool
propiedadAE2 p x = all (any p) (ramas x)

ramas :: Arbol a => [[a]]
ramas (H x)     = [[x]]
ramas (N x i d) = map (x:) (ramas i ++ ramas d)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    propiedadEA :: (a -> Bool) -> Arbol a -> Bool
-- tal que (propiedadEA p a) se verifica si el árbol a tiene alguna rama
-- en la que todos sus nodos cumplen la propiedad p. Por ejemplo,
--    propiedadEA (>0) a1  == True
--    propiedadEA even a1  == False
--    propiedadEA (>7) a1  == True
-- ---------------------------------------------------------------------

-- 1ª definición
```

```haskell
propiedadEA :: (a -> Bool) -> Arbol a -> Bool
propiedadEA p (H x)     = p x
propiedadEA p (N x i d) = p x && (propiedadEA p i || propiedadEA p d)

-- 2ª definición
propiedadEA2 :: (a -> Bool) -> Arbol a -> Bool
propiedadEA2 p x = any (all p) (ramas x)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un clique de un grafo no dirigido G es un conjunto de
-- vértices V tal que el subgrafo de G inducido por V es un grafo
-- completo. Por ejemplo, en el grafo,
--    6
--     \
--      4 ---- 5
--      |      | \
--      |      |  1
--      |      | /
--      3 ---- 2
-- el conjunto de vértices {1,2,5} es un clique y el conjunto {2,3,4,5}
-- no lo es.
--
-- En Haskell se puede representar el grafo anterior por
--    g1 :: Grafo Int Int
--    g1 = creaGrafo ND
--                   (1,6)
--                   [(1,2,0),(1,5,0),(2,3,0),(3,4,0),(5,2,0),(4,5,0),(4,6,0)]
--
-- Definir la función
--    esClique :: Grafo Int Int -> [Int] -> Bool
-- tal que (esClique g xs) se verifica si xs es un clique de g. Por
-- ejemplo,
--    esClique g1 [1,2,5]   == True
--    esClique g1 [2,3,4,5] == False
-- ---------------------------------------------------------------------

g1 :: Grafo Int Int
g1 = creaGrafo ND
               (1,6)
               [(1,2,0),(1,5,0),(2,3,0),(3,4,0),(5,2,0),(4,5,0),(4,6,0)]
```

```haskell
esClique :: Grafo Int Int -> [Int] -> Bool
esClique g xs = all (aristaEn g) [(x,y) | x <- ys, y <- ys, y < x]
    where ys = sort xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    polNumero :: Integer -> Polinomio Integer
-- tal que (polNumero n) es el polinomio cuyos coeficientes son los
-- dígitos de n. Por ejemplo, si n = 5703, el polinomio es
-- 5x^3 + 7x^2 + 3. En Haskell,
--    ghci> polNumero 5703
--    5*x^3 + 7*x^2 + 3
-- ---------------------------------------------------------------------

polNumero :: Integer -> Polinomio Integer
polNumero n = aux (zip [0..] (reverse (digitos n)))
    where aux []         = polCero
          aux ((m,a):ps) = consPol m a (aux ps)

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    zeta :: Integer -> Int
-- tal que (zeta n) es el número de enteros positivos k <= n, tales que
-- el polinomio (polNumero k) tiene alguna raíz entera. Por ejemplo,
--    zeta 100    == 33
--    zeta 100000 == 14696
-- ---------------------------------------------------------------------

zeta :: Integer -> Int
zeta n = length [k | k <- [1..n], tieneRaizEntera k]

tieneRaizEntera :: Integer -> Bool
tieneRaizEntera n = or [esRaiz c p | c <- ds]
    where p  = polNumero n
          t  = n `mod` 10
          ds = 0 : divisores t
```

```haskell
divisores :: Integer -> [Integer]
divisores n = concat [[x,-x] | x <- [1..n], rem n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck las siguientes propiedades:
-- + El valor de (polNumero n) en 0 es igual al último dígito de n
-- + El valor de (polNumero n) en 1 es igual a la suma de los dígitos de n.
-- + El valor de (polNumero n) en 10 es igual a n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_polNumero :: Integer -> Property
prop_polNumero n =
    n > 0 ==> valor (polNumero n) 0 == last ds &&
              valor (polNumero n) 1 == sum ds &&
              valor (polNumero n) 10 == n
    where ds = digitos n

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Para cada número n con k dígitos se define una sucesión
-- de tipo Fibonacci cuyos k primeros elementos son los dígitos de n y
-- los siguientes se obtienen sumando los k anteriores términos de la
-- sucesión. Por ejemplo, la sucesión definida por 197 es
--    1, 9, 7, 17, 33, 57, 107, 197, ...
--
-- Definir la función
--    fibGen :: Integer -> [Integer]
-- tal que (fibGen n) es la sucesión de tipo Fibonacci definida por
-- n. Por ejemplo,
--    ghci> take 10 (fibGen 197)
--    [1,9,7,17,33,57,107,197,361,665]
-- ---------------------------------------------------------------------

-- 1ª definición
fibGen1 :: Integer -> [Integer]
fibGen1 n = suc
    where ds     = digitos n
          k      = genericLength ds
          aux xs = sum (take k xs) : aux (tail xs)
```

```
          suc    = ds ++ aux suc

-- 2ª definición
fibGen2 :: Integer -> [Integer]
fibGen2 n = ds ++ map head (tail sucLista)
    where ds       = digitos n
          sig xs   = sum xs : init xs
          sucLista = iterate sig (reverse ds)

-- 3ª definición
fibGen3 :: Integer -> [Integer]
fibGen3 n = ds ++ map last (tail sucLista)
    where ds       = digitos n
          sig' xs  = tail xs ++ [sum xs]
          sucLista = iterate sig' ds

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Un número n > 9 es un número de Keith si n aparece en
-- la sucesión de tipo Fibonacci definida por n. Por ejemplo, 197 es un
-- número de Keith.
--
-- Definir la función
--    esKeith :: Integer -> Bool
-- tal que (esKeith n) se verifica si n es un número de Keith. Por
-- ejemplo,
--    esKeith 197   == True
--    esKeith 54798 == False
-- ---------------------------------------------------------------------

esKeith :: Integer -> Bool
esKeith n = n == head (dropWhile (<n) $ fibGen1 n)

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Calcular todos los número se Keith con 5 dígitos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> filter esKeith [10^4..10^5-1]
--    [31331,34285,34348,55604,62662,86935,93993]
```

1.8. Examen 7 (23 de junio de 2016)

```
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 1ª convocatoria (23 de junio de 2016)
-- ---------------------------------------------------------------------

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

import Data.Array
import Data.List
import Test.QuickCheck
import qualified Data.Set as S
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    particiones :: [a] -> Int -> [[[a]]]
-- tal que (particiones xs k) es la lista de las particiones de xs en k
-- subconjuntos disjuntos. Por ejemplo,
--    ghci> particiones [2,3,6] 2
--    [[[2],[3,6]],[[2,3],[6]],[[3],[2,6]]]
--    ghci> particiones [2,3,6] 3
--    [[[2],[3],[6]]]
--    ghci> particiones [4,2,3,6] 3
--    [[[4],[2],[3,6]],[[4],[2,3],[6]],[[4],[3],[2,6]],
--     [[4,2],[3],[6]],[[2],[4,3],[6]],[[2],[3],[4,6]]]
--    ghci> particiones [4,2,3,6] 1
--    [[[4,2,3,6]]]
--    ghci> particiones [4,2,3,6] 4
--    [[[4],[2],[3],[6]]]
-- ---------------------------------------------------------------------

particiones :: [a] -> Int -> [[[a]]]
particiones [] _     = []
particiones _  0     = []
particiones xs 1     = [[xs]]
particiones (x:xs) k = [[x]:ys | ys <- particiones xs (k-1)] ++
                       concat [inserciones x ys | ys <- particiones xs k]
```

```haskell
-- (inserciones x yss) es la lista obtenida insertando x en cada uno de los
-- conjuntos de yss. Por ejemplo,
--    inserciones 4 [[3],[2,5]]  ==  [[[4,3],[2,5]],[[3],[4,2,5]]]
inserciones :: a -> [[a]] -> [[[a]]]
inserciones x []       = []
inserciones x (ys:yss) = ((x:ys):yss) : [ys:zss | zss <- inserciones x yss]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Las expresiones aritméticas con números enteros, sumas
-- y restas se puede representar con el tipo de datos Expr definido por
--    data Expr = N Int
--              | S Expr Expr
--              | R Expr Expr
--      deriving (Eq, Show)
-- Por ejemplo, la expresión 3+(4-2) se representa por
-- (S (N 3) (R (N 4) (N 2)))
--
-- Definir la función
--    expresiones :: [Int] -> [Expr]
-- tal que (expresiones ns) es la lista de todas las expresiones
-- que se pueden construir intercalando las operaciones de suma o resta
-- entre los números de ns. Por ejemplo,
--   ghci> expresiones [2,3,5]
--   [S (N 2) (S (N 3) (N 5)),R (N 2) (S (N 3) (N 5)),S (N 2) (R (N 3) (N 5)),
--    R (N 2) (R (N 3) (N 5)),S (S (N 2) (N 3)) (N 5),R (S (N 2) (N 3)) (N 5),
--    S (R (N 2) (N 3)) (N 5),R (R (N 2) (N 3)) (N 5)]
--    > length (expresiones [2,3,5,9])
--    40
-- ---------------------------------------------------------------------

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

expresiones :: [Int] -> [Expr]
expresiones []  = []
expresiones [n] = [N n]
expresiones ns  = [e | (is,ds) <- divisiones ns
```

```
                    , i       <- expresiones is
                    , d       <- expresiones ds
                    , e       <- combina i d]

-- (divisiones xs) es la lista de las divisiones de xs en dos listas no
-- vacías. Por ejemplo,
--    divisiones "bcd"   ==  [("b","cd"),("bc","d")]
--    divisiones "abcd"  ==  [("a","bcd"),("ab","cd"),("abc","d")]
divisiones :: [a] -> [([a],[a])]
divisiones []     = []
divisiones [_]    = []
divisiones (x:xs) = ([x],xs) : [(x:is,ds) | (is,ds) <- divisiones xs]

-- (combina e1 e2) es la lista de las expresiones obtenidas combinando
-- las expresiones e1 y e2 con una operación. Por ejemplo,
--    combina (N 2) (N 3)  ==  [S (N 2) (N 3),R (N 2) (N 3)]
combina :: Expr -> Expr -> [Expr]
combina e1 e2 = [S e1 e2, R e1 e2]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    valores :: [Int] -> [Int]
-- tal que (valores n) es el conjunto de los valores de las expresiones
-- que se pueden construir intercalando las operaciones de suma o resta
-- entre los números de ns. Por ejemplo,
--    valores [2,3,5]    ==  [-6,0,4,10]
--    valores [2,3,5,9]  ==  [-15,-9,-5,1,3,9,13,19]
-- ---------------------------------------------------------------------

valores :: [Int] -> [Int]
valores ns = sort (nub (map valor (expresiones ns)))

-- (valor e) es el valor de la expresión e. Por ejemplo,
--    valor (S (R (N 2) (N 3)) (N 5))  ==  4
valor :: Expr -> Int
valor (N n)   = n
valor (S i d) = valor i + valor d
valor (R i d) = valor i - valor d

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

```
-- Ejercicio 3. Dada una matriz cuyos elementos son 0 ó 1, su relleno es
-- la matriz obtenida haciendo iguales a 1 los elementos de las filas y
-- de las columna que contienen algún uno. Por ejemplo, el relleno de la
-- matriz de la izquierda es la de la derecha:
--    0 0 0 0 0    1 0 0 1 0
--    0 0 0 0 0    1 0 0 1 0
--    0 0 0 1 0    1 1 1 1 1
--    1 0 0 0 0    1 1 1 1 1
--    0 0 0 0 0    1 0 0 1 0
--
-- Las matrices se pueden representar mediante tablas cuyos índices son
-- pares de enteros
--    type Matriz = Array (Int,Int) Int
-- por ejemplo, la matriz de la izquierda de la figura anterior se
-- define por
--    ej :: Matriz
--    ej = listArray ((1,1),(5,5)) [0, 0, 0, 0, 0,
--                                  0, 0, 0, 0, 0,
--                                  0, 0, 0, 1, 0,
--                                  1, 0, 0, 0, 0,
--                                  0, 0, 0, 0, 0]
-- Definir la función
--    relleno :: Matriz -> Matriz
-- tal que (relleno p) es el relleno de la matriz p. Por ejemplo,
--    ghci> elems (relleno ej)
--    [1,0,0,1,0,
--     1,0,0,1,0,
--     1,1,1,1,1,
--     1,1,1,1,1,
--     1,0,0,1,0]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

ej :: Matriz
ej = listArray ((1,1),(5,5)) [0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0,
                              0, 0, 0, 1, 0,
                              1, 0, 0, 0, 0,
                              0, 0, 0, 0, 0]
```

```haskell
-- 1ª solución
-- ===========

relleno1 :: Matriz -> Matriz
relleno1 p =
    array ((1,1),(m,n)) [((i,j),f i j) | i <- [1..m], j <- [1..n]]
    where (_,(m,n)) = bounds p
          f i j | 1 `elem` [p!(i,k) | k <- [1..n]] = 1
                | 1 `elem` [p!(k,j) | k <- [1..m]] = 1
                | otherwise                        = 0

-- 2ª solución
-- ===========

relleno2 :: Matriz -> Matriz
relleno2 p =
    array ((1,1),(m,n)) [((i,j),f i j) | i <- [1..m], j <- [1..n]]
    where (_,(m,n)) = bounds p
          filas     = filasConUno p
          columnas  = columnasConUno p
          f i j | i `elem` filas    = 1
                | j `elem` columnas = 1
                | otherwise         = 0

-- (filasConUno p) es la lista de las filas de p que tienen algún
-- uno. Por ejemplo,
--    filasConUno ej  ==  [3,4]
filasConUno :: Matriz -> [Int]
filasConUno p = [i | i <- [1..m], filaConUno p i]
    where (_,(m,n)) = bounds p

-- (filaConUno p i) se verifica si p tiene algún uno en la fila i. Por
-- ejemplo,
--    filaConUno ej 3  ==  True
--    filaConUno ej 2  ==  False
filaConUno :: Matriz -> Int -> Bool
filaConUno p i = any (==1) [p!(i,j) | j <- [1..n]]
    where (_,(_,n)) = bounds p
```

```haskell
-- (columnasConUno p) es la lista de las columnas de p que tienen algún
-- uno. Por ejemplo,
--    columnasConUno ej  ==  [1,4]
columnasConUno :: Matriz -> [Int]
columnasConUno p = [j | j <- [1..n], columnaConUno p j]
    where (_,(m,n)) = bounds p

-- (columnaConUno p i) se verifica si p tiene algún uno en la columna i. Por
-- ejemplo,
--    columnaConUno ej 1  ==  True
--    columnaConUno ej 2  ==  False
columnaConUno :: Matriz -> Int -> Bool
columnaConUno p j = any (==1) [p!(i,j) | i <- [1..m]]
    where (_,(m,_)) = bounds p

-- 3ª solución
-- ===========

relleno3 :: Matriz -> Matriz
relleno3 p = p // ([((i,j),1) | i <- filas,  j <- [1..n]] ++
                   [((i,j),1) | i <- [1..m], j <- columnas])
  where (_,(m,n)) = bounds p
        filas     = filasConUno p
        columnas  = columnasConUno p


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

--    ghci> let f i j = if i == j then 1 else 0
--    ghci> let q n = array ((1,1),(n,n)) [((i,j),f i j) | i <- [1..n], j <- [1..
--
--    ghci> sum (elems (relleno1 (q 200)))
--    40000
--    (6.90 secs, 1,877,369,544 bytes)
--
--    ghci> sum (elems (relleno2 (q 200)))
--    40000
--    (0.46 secs, 57,354,168 bytes)
--
```

```
--    ghci> sum (elems (relleno3 (q 200)))
--    40000
--    (0.34 secs, 80,465,144 bytes)
--
--    ghci> sum (elems (relleno2 (q 500)))
--    250000
--    (4.33 secs, 353,117,640 bytes)
--
--    ghci> sum (elems (relleno3 (q 500)))
--    250000
--    (2.40 secs, 489,630,048 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un número natural n es una potencia perfecta si existen
-- dos números naturales m > 1 y k > 1 tales que n = m^k. Las primeras
-- potencias perfectas son
--    4 = 2², 8 = 2³, 9 = 3², 16 = 2⁴, 25 = 5², 27 = 3³, 32 = 2⁵,
--    36 = 6², 49 = 7², 64 = 2⁶, ...
--
-- Definir la sucesión
--    potenciasPerfectas :: [Integer]
-- cuyos términos son las potencias perfectas. Por ejemplo,
--    take 10 potenciasPerfectas  ==  [4,8,9,16,25,27,32,36,49,64]
--    potenciasPerfectas !! 100   ==  6724
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

potenciasPerfectas1 :: [Integer]
potenciasPerfectas1 = filter esPotenciaPerfecta [4..]

-- (esPotenciaPerfecta x) se verifica si x es una potencia perfecta. Por
-- ejemplo,
--    esPotenciaPerfecta 36  ==  True
--    esPotenciaPerfecta 72  ==  False
esPotenciaPerfecta :: Integer -> Bool
esPotenciaPerfecta = not . null. potenciasPerfectasDe

-- (potenciasPerfectasDe x) es la lista de pares (a,b) tales que
```

```haskell
-- x = a^b. Por ejemplo,
--    potenciasPerfectasDe 64  ==  [(2,6),(4,3),(8,2)]
--    potenciasPerfectasDe 72  ==  []
potenciasPerfectasDe :: Integer -> [(Integer,Integer)]
potenciasPerfectasDe n =
    [(m,k) | m <- takeWhile (\x -> x*x <= n) [2..]
           , k <- takeWhile (\x -> m^x <= n) [2..]
           , m^k == n]

-- 2ª solución
-- ===========

potenciasPerfectas2 :: [Integer]
potenciasPerfectas2 = [x | x <- [4..], esPotenciaPerfecta2 x]

-- (esPotenciaPerfecta2 x) se verifica si x es una potencia perfecta. Por
-- ejemplo,
--    esPotenciaPerfecta2 36  ==  True
--    esPotenciaPerfecta2 72  ==  False
esPotenciaPerfecta2 :: Integer -> Bool
esPotenciaPerfecta2 x = mcd (exponentes x) > 1

-- (exponentes x) es la lista de los exponentes de l factorización prima
-- de x. Por ejemplos,
--    exponentes 36  ==  [2,2]
--    exponentes 72  ==  [3,2]
exponentes :: Integer -> [Int]
exponentes x = [length ys | ys <- group (primeFactors x)]

-- (mcd xs) es el máximo común divisor de la lista xs. Por ejemplo,
--    mcd [4,6,10]  ==  2
--    mcd [4,5,10]  ==  1
mcd :: [Int] -> Int
mcd = foldl1 gcd

-- 3ª definición
-- =============

potenciasPerfectas3 :: [Integer]
potenciasPerfectas3 = mezclaTodas potencias
```

```haskell
-- potencias es la lista las listas de potencias de todos los números
-- mayores que 1 con exponentes mayores que 1. Por ejemplo,
--    ghci> map (take 3) (take 4 potencias)
--    [[4,8,16],[9,27,81],[16,64,256],[25,125,625]]
potencias:: [[Integer]]
potencias = [[n^k | k <- [2..]] | n <- [2..]]

-- (mezclaTodas xss) es la mezcla ordenada sin repeticiones de las
-- listas ordenadas xss. Por ejemplo,
--    take 7 (mezclaTodas potencias)  ==  [4,8,9,16,25,27,32]
mezclaTodas :: Ord a => [[a]] -> [a]
mezclaTodas = foldr1 xmezcla
    where xmezcla (x:xs) ys = x : mezcla xs ys

-- (mezcla xs ys) es la mezcla ordenada sin repeticiones de las
-- listas ordenadas xs e ys. Por ejemplo,
--    take 7 (mezcla [2,5..] [4,6..])  ==  [2,4,5,6,8,10,11]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla (x:xs) (y:ys) | x < y  = x : mezcla xs (y:ys)
                     | x == y = x : mezcla xs ys
                     | x > y  = y : mezcla (x:xs) ys

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

--    ghci> potenciasPerfectas1 !! 100
--    6724
--    (3.39 secs, 692758212 bytes)
--    ghci> potenciasPerfectas2 !! 100
--    6724
--    (0.29 secs, 105,459,200 bytes)
--    ghci> potenciasPerfectas3 !! 100
--    6724
--    (0.01 secs, 1582436 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    minimales :: Ord a => S.Set (S.Set a) -> S.Set (S.Set a)
-- tal que (minimales xss) es el conjunto de los elementos de xss que no
```

```haskell
-- están contenidos en otros elementos de xss. Por ejemplo,
--    ghci> minimales (S.fromList (map S.fromList [[1,3],[2,3,1],[3,2,5]]))
--    fromList [fromList [1,3],fromList [2,3,5]]
--    ghci> minimales (S.fromList (map S.fromList [[1,3],[2,3,1],[3,1],[3,2,5]]))
--    fromList [fromList [1,3],fromList [2,3,5]]
--    ghci> minimales (S.fromList (map S.fromList [[1,3],[2,3,1],[3,1,3],[3,2,5]]
--    fromList [fromList [1,3],fromList [2,3,5]]
-- ---------------------------------------------------------------------

minimales :: Ord a => S.Set (S.Set a) -> S.Set (S.Set a)
minimales xss = S.filter esMinimal xss
    where esMinimal xs = S.null (S.filter (`S.isProperSubsetOf` xs) xss)
```

1.9. Examen 8 (1 de septiembre de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 2ª convocatoria (1 de septiembre de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Numbers.Primes
import Data.Matrix
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un primo cubano es un número primo que se puede escribir
-- como diferencia de dos cubos consecutivos. Por ejemplo, el 61 es un
-- primo cubano porque es primo y 61 = 5^3 - 4^3.
--
-- Definir la sucesión
--    cubanos :: [Integer]
-- tal que sus elementos son los números cubanos. Por ejemplo,
--    ghci> take 15 cubanos
--    [7,19,37,61,127,271,331,397,547,631,919,1657,1801,1951,2269]
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución
-- ===========

cubanos1 :: [Integer]
cubanos1 = filter isPrime (zipWith (-) (tail cubos) cubos)
  where cubos = map (^3) [1..]

-- 2ª solución
-- ===========

cubanos2 :: [Integer]
cubanos2 = filter isPrime [(x+1)^3 - x^3 | x <- [1..]]

-- 3ª solución
-- ===========

cubanos3 :: [Integer]
cubanos3 = filter isPrime [3*x^2 + 3*x + 1 | x <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    segmentos :: (Enum a, Eq a) => [a] -> [[a]]
-- tal que (segmentos xss) es la lista de los segmentos maximales (es
-- decir, de máxima longitud) de xss formados por elementos
-- consecutivos. Por ejemplo,
--    segmentos [1,2,5,6,4]     ==  [[1,2],[5,6],[4]]
--    segmentos [1,2,3,4,7,8,9] ==  [[1,2,3,4],[7,8,9]]
--    segmentos "abbccddeeebc"  ==  ["ab","bc","cd","de","e","e","bc"]
--
-- Nota: Se puede usar la función succ tal que (succ x) es el sucesor de
-- x. Por ejemplo,
--    succ 3    ==  4
--    succ 'c'  ==  'd'
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
segmentos1 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos1 []  = []
segmentos1 [x] = [[x]]
```

```haskell
segmentos1 (x:xs) | y == succ x = (x:y:ys):zs
                  | otherwise   = [x] : (y:ys):zs
    where ((y:ys):zs) = segmentos1 xs

-- 2ª definición.
segmentos2 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos2 []  = []
segmentos2 xs = ys : segmentos2 zs
  where ys = inicial xs
        n  = length ys
        zs = drop n xs

-- (inicial xs) es el segmento inicial de xs formado por elementos
-- consecutivos. Por ejemplo,
--    inicial [1,2,5,6,4]  ==  [1,2]
--    inicial "abccddeeebc"  ==  "abc"
inicial :: (Enum a, Eq a) => [a] -> [a]
inicial [] = []
inicial (x:xs) =
  [y | (y,z) <- takeWhile (\(u,v) -> u == v) (zip (x:xs) [x..])]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles se pueden representar mediante el siguiente
-- tipo de datos
--    data Arbol a = N a [Arbol a]
--                 deriving Show
-- Por ejemplo, los árboles
--      1              3
--     / \            /|\
--    2   3          / | \
--        |         5  4  7
--        4         |     /\
--                  6    2  1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]
--
-- Definir la función
--    ramasLargas :: Arbol b -> [[b]]
```

```haskell
-- tal que (ramasLargas a) es la lista de las ramas más largas del árbol
-- a. Por ejemplo,
--    ramas ej1  ==  [[1,3,4]]
--    ramas ej2  ==  [[3,5,6],[3,7,2],[3,7,1]]
-- ----------------------------------------------------------------------

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

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

ramasLargas :: Arbol b -> [[b]]
ramasLargas a = [xs | xs <- todasRamas,
                      length xs == m]
  where todasRamas = ramas a
        m = maximum (map length todasRamas)

-- (ramas a) es la lista de todas las ramas del árbol a. Por ejemplo,
--    ramas ej1  ==  [[1,2],[1,3,4]]
--    ramas ej2  ==  [[3,5,6],[3,4],[3,7,2],[3,7,1]]
ramas :: Arbol b -> [[b]]
ramas (N x []) = [[x]]
ramas (N x as) = [x : xs | a <- as, xs <- ramas a]

-- 2ª solución:
ramas2 :: Arbol b -> [[b]]
ramas2 (N x []) = [[x]]
ramas2 (N x as) = concatMap (map (x:)) (map ramas2 as)

-- ----------------------------------------------------------------------
-- Ejercicio 4.1. El problema de las N torres consiste en colocar N
-- torres en un  tablero con N filas y N columnas de forma que no haya
-- dos torres en la misma fila ni en la misma columna.
--
-- Cada solución del problema de puede representar mediante una matriz
-- con ceros y unos donde los unos representan las posiciones ocupadas
-- por las torres y los ceros las posiciones libres. Por ejemplo,
--    ( 0 1 0 )
```

```
--    ( 1 0 0 )
--    ( 0 0 1 )
-- representa una solución del problema de las 3 torres.
--
-- Definir la función
--    torres  :: Int -> [Matrix Int]
-- tal que (torres n) es la lista de las soluciones del problema de las
-- n torres. Por ejemplo,
--       ghci> torres 3
--       [( 1 0 0 )
--        ( 0 1 0 )
--        ( 0 0 1 )
--       ,( 1 0 0 )
--        ( 0 0 1 )
--        ( 0 1 0 )
--       ,( 0 1 0 )
--        ( 1 0 0 )
--        ( 0 0 1 )
--       ,( 0 1 0 )
--        ( 0 0 1 )
--        ( 1 0 0 )
--       ,( 0 0 1 )
--        ( 1 0 0 )
--        ( 0 1 0 )
--       ,( 0 0 1 )
--        ( 0 1 0 )
--        ( 1 0 0 )
--       ]
--   donde se ha indicado con 1 las posiciones ocupadas por las torres.
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

torres1 :: Int -> [Matrix Int]
torres1 n =
  [permutacionAmatriz n p | p <- sort (permutations [1..n])]

permutacionAmatriz :: Int -> [Int] -> Matrix Int
permutacionAmatriz n p =
```

```
  matrix n n f
  where f (i,j) | (i,j) `elem` posiciones = 1
                | otherwise               = 0
        posiciones = zip [1..n] p

-- 2ª definición
-- =============

torres2 :: Int -> [Matrix Int]
torres2 = map fromLists . permutations . toLists . identity

-- El cálculo con la definición anterior es:
--    ghci> identity 3
--    ( 1 0 0 )
--    ( 0 1 0 )
--    ( 0 0 1 )
--
--    ghci> toLists it
--    [[1,0,0],[0,1,0],[0,0,1]]
--
--    ghci> permutations it
--    [[[1,0,0],[0,1,0],[0,0,1]],
--     [[0,1,0],[1,0,0],[0,0,1]],
--     [[0,0,1],[0,1,0],[1,0,0]],
--     [[0,1,0],[0,0,1],[1,0,0]],
--     [[0,0,1],[1,0,0],[0,1,0]],
--     [[1,0,0],[0,0,1],[0,1,0]]]
--
--    ghci> map fromLists it
--    [( 1 0 0 )
--     ( 0 1 0 )
--     ( 0 0 1 )
--    ,( 0 1 0 )
--     ( 1 0 0 )
--     ( 0 0 1 )
--    ,( 0 0 1 )
--     ( 0 1 0 )
--     ( 1 0 0 )
--    ,( 0 1 0 )
--     ( 0 0 1 )
```

```
--     ( 1 0 0 )
--    ,( 0 0 1 )
--     ( 1 0 0 )
--     ( 0 1 0 )
--    ,( 1 0 0 )
--     ( 0 0 1 )
--     ( 0 1 0 )
--    ]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    nTorres :: Int -> Integer
-- tal que (nTorres n) es el número de soluciones del problema de las n
-- torres. Por ejemplo,
--       ghci> nTorres 3
--       6
--       ghci> length (show (nTorres (10^4)))
--       35660
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

nTorres1 :: Int -> Integer
nTorres1 = genericLength . torres1

-- 2ª definición de nTorres
-- ========================

nTorres2 :: Int -> Integer
nTorres2 n = product [1..fromIntegral n]

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

--    ghci> nTorres1 9
--    362880
--    (4.22 secs, 693,596,128 bytes)
--    ghci> nTorres2 9
--    362880
```

```
--    (0.00 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. El conjunto de todas las palabras se puede ordenar
-- como se muestra a continuación:
--      a,   b, ...,   z,   A,   B, ...,   Z,
--     aa,  ab, ...,  az,  aA,  aB, ...,  aZ,
--     ba,  bb, ...,  bz,  bA,  bB, ...,  bZ,
--    ...
--     za,  zb, ...,  zz,  zA,  zB, ...,  zZ,
--    aaa, aab, ..., aaz, aaA, aaB, ..., aaZ,
--    baa, bab, ..., baz, baA, baB, ..., baZ,
--    ...
--
-- Definir la función
--    palabras :: [String]
-- tal que palabras es la lista ordenada de todas las palabras. Por
-- ejemplo,
--      ghci> take 10 (drop 100 palabras)
--      ["aW","aX","aY","aZ","ba","bb","bc","bd","be","bf"]
--      ghci> take 10 (drop 2750 palabras)
--      ["ZU","ZV","ZW","ZX","ZY","ZZ","aaa","aab","aac","aad"]
--      ghci> palabras !! (10^6)
--      "gePO"
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

palabras1 :: [String]
palabras1 = concatMap palabrasDeLongitud [1..]

-- (palabrasDeLongitud n) es la lista ordenada de palabras longitud
-- n. Por ejemplo,
--    take 3 (palabrasDeLongitud 2)  ==  ["aa","ab","ac"]
--    take 3 (palabrasDeLongitud 3)  ==  ["aaa","aab","aac"]
palabrasDeLongitud :: Integer -> [String]
palabrasDeLongitud 1 = [[x]  | x <- letras]
palabrasDeLongitud n = [x:ys | x <- letras,
                               ys <- palabrasDeLongitud (n-1)]
```

```haskell
-- letras es la lista ordenada de las letras.
letras :: [Char]
letras = ['a'..'z'] ++ ['A'..'Z']

-- 2ª definición
-- =============

palabras2 :: [String]
palabras2 = concat (iterate f elementales)
  where f = concatMap (\x -> map (x++) elementales)

-- elementales es la lista de las palabras de una letra.
elementales :: [String]
elementales = map (:[]) letras

-- 3ª definición
-- =============

palabras3 :: [String]
palabras3 = tail $ map reverse aux
  where aux = "" : [c:cs | cs <- aux, c <- letras]

-- Comparación
-- ===========

--    ghci> palabras1 !! (10^6)
--    "gePO"
--    (0.87 secs, 480,927,648 bytes)
--    ghci> palabras2 !! (10^6)
--    "gePO"
--    (0.11 secs, 146,879,840 bytes)
--    ghci> palabras3 !! (10^6)
--    "gePO"
--    (0.25 secs, 203,584,608 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    posicion :: String -> Integer
-- tal que (posicion n) es la palabra que ocupa la posición n en la lista
```

```
-- ordenada de todas las palabras. Por ejemplo,
--    posicion "c"                    ==  2
--    posicion "ab"                   ==  53
--    posicion "ba"                   ==  104
--    posicion "eva"                  ==  14664
--    posicion "adan"                 ==  151489
--    posicion "HoyEsMartes"          ==  4957940944437977046
--    posicion "EnUnLugarDeLaMancha"  ==  241779893912461058861484239910864
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

posicion1 :: String -> Integer
posicion1 cs = genericLength (takeWhile (/=cs) palabras1)

-- 2ª definición
-- =============

posicion2 :: String -> Integer
posicion2 ""     = -1
posicion2 (c:cs) = (1 + posicionLetra c) * 52^(length cs) + posicion2 cs

posicionLetra :: Char -> Integer
posicionLetra c = genericLength (takeWhile (/=c) letras)

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Comprobar con QuickCheck que para todo entero positivo
-- n se verifica que
--    posicion (palabras `genericIndex` n) == n
-- ---------------------------------------------------------------------

-- La propiedad es
prop_palabras :: (Positive Integer) -> Bool
prop_palabras (Positive n) =
  posicion2 (palabras3 `genericIndex` n) == n

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

2

Exámenes del grupo 2

Antonia M. Chávez

2.1. Examen 1 (6 de Noviembre de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (6 de noviembre de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    tieneRepeticionesC :: Eq a => [a] -> Bool
-- tal que (tieneRepeticionesC xs) se verifica si xs tiene algún
-- elemento repetido. Por ejemplo,
--    tieneRepeticionesC [3,2,5,2,7]          ==  True
--    tieneRepeticionesC [3,2,5,4]            ==  False
--    tieneRepeticionesC (5:[1..2000000000])  ==  True
--    tieneRepeticionesC [1..20000]           ==  False
-- ---------------------------------------------------------------------

tieneRepeticionesC :: Eq a => [a] -> Bool
tieneRepeticionesC xs =
  or [ocurrencias x xs > 1 | x <- xs]
  where ocurrencias x xs = length [y | y <- xs, x == y]

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    tieneRepeticionesR :: Eq a => [a] -> Bool
-- tal que (tieneRepeticionesR xs) se verifica si xs tiene algún
```

```haskell
-- elemento repetido. Por ejemplo,
--    tieneRepeticionesR [3,2,5,2,7]          ==  True
--    tieneRepeticionesR [3,2,5,4]            ==  False
--    tieneRepeticionesR (5:[1..2000000000])  ==  True
--    tieneRepeticionesR [1..20000]           ==  False
-- ---------------------------------------------------------------------

tieneRepeticionesR :: Eq a => [a] -> Bool
tieneRepeticionesR []     = False
tieneRepeticionesR (x:xs) = elem x xs || tieneRepeticionesR xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión, la función
--    sinRepetidos :: Eq a => [a] -> [a]
-- tal que (sinRepetidos xs) es la lista xs sin repeticiones, da igual
-- el orden. Por ejemplo,
--    sinRepetidos [1,1,1,2]             == [1,2]
--    sinRepetidos [1,1,2,1,1]           == [2,1]
--    sinRepetidos [1,2,4,3,4,2,5,3,4,2] == [1,5,3,4,2]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión, la función
--    reparte :: [a] -> [Int] -> [[a]]
-- tal que (reparte xs ns) es la partición de xs donde las longitudes de
-- las partes las indican los elementos de ns. Por ejemplo,
--    reparte [1..10] [2,5,0,3]  ==  [[1,2],[3,4,5,6,7],[],[8,9,10]]
--    reparte [1..10] [1,4,2,3]  ==  [[1],[2,3,4,5],[6,7],[8,9,10]]
-- ---------------------------------------------------------------------

reparte :: [a] -> [Int] -> [[a]]
reparte [] _     = []
reparte _ []     = []
reparte xs (n:ns) = take n xs : reparte (drop n xs) ns
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    agrupa :: Eq a => (b -> a) -> [b] -> [(a, [b])]
-- tal que (agrupa f xs) es la lista de pares  obtenida agrupando los
-- elementos de xs según sus valores mediante la función f. Por ejemplo,
-- si queremos agrupar los elementos de la lista ["voy", "ayer", "ala",
-- "losa"] por longitudes, saldrá que hay dos palabras de longitud 3 y
-- otras dos de longitud 4
--    ghci> agrupa length ["voy", "ayer", "ala", "losa"]
--    [(3,["voy","ala"]),(4,["ayer","losa"])]
-- Si queremos agrupar las palabras de la lista ["claro", "ayer", "ana",
-- "cosa"] por su inicial, salen dos por la 'a' y otras dos por la 'c'.
--    ghci> agrupa head ["claro", "ayer", "ana", "cosa"]
--    [('a',["ayer","ana"]),('c',["claro","cosa"])]
-- ---------------------------------------------------------------------

agrupa :: Eq a => (b -> a) -> [b] -> [(a, [b])]
agrupa f xs =
  [(i,[y | y <- xs, f y == i]) | i <- yss]
  where yss = sinRepetidos [f y | y <- xs]
```

2.2. Examen 2 (4 de Diciembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (4 de diciembre de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    inserta :: [a] -> [[a]] -> [[a]]
-- tal que (inserta xs yss) es la lista obtenida insertando
-- + el primer elemento de xs como primero en la primera lista de yss,
-- + el segundo elemento de xs como segundo en la segunda lista de yss
--   (si la segunda lista de yss tiene al menos un elemento),
-- + el tercer elemento de xs como tercero en la tercera lista de yss
--   (si la tercera lista de yss tiene al menos dos elementos),
-- y así sucesivamente. Por ejemplo,
--    inserta [1,2,3] [[4,7],[6],[9,5,8]]  ==  [[1,4,7],[6,2],[9,5,3,8]]
--    inserta [1,2,3] [[4,7],[] ,[9,5,8]]  ==  [[1,4,7],[],   [9,5,3,8]]
```

```haskell
--    inserta [1,2]   [[4,7],[6],[9,5,8]]  ==  [[1,4,7],[6,2],[9,5,8]]
--    inserta [1,2,3] [[4,7],[6]]          ==  [[1,4,7],[6,2]]
--    inserta "tad"   ["odo","pra","naa"]  ==  ["todo","para","nada"]
-- ---------------------------------------------------------------------

inserta :: [a] -> [[a]] -> [[a]]
inserta xs yss = aux xs yss 0 where
    aux [] yss _ = yss
    aux xs []  _ = []
    aux (x:xs) (ys:yss) n
        | length us == n = (us ++ x : vs) : aux xs yss (n+1)
        | otherwise      = ys : aux xs yss (n+1)
        where (us,vs) = splitAt n ys

-- ---------------------------------------------------------------------
-- Ejercicio 2. 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 es un término si es un producto de variables. Por
-- ejemplo, x*(y*z) es un término pero x+(y*z) ni x*(y+z) lo son.
--
-- 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)

esTermino :: Expr -> Bool
esTermino (V _)   = True
esTermino (S _ _) = False
esTermino (P a b) = esTermino a && esTermino b

esNormal :: Expr -> Bool
esNormal (S a b) = esNormal a && esNormal b
esNormal a       = esTermino a

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 2.1. Un elemento de una lista es punta si ninguno de los
-- siguientes es mayor que él.
--
-- Definir la función
--    puntas :: [Int] -> [Int]
-- tal que (puntas xs) es la lista de los elementos puntas de xs. Por
-- ejemplo,
```

```haskell
--    puntas [80,1,7,8,4]  ==  [80,8,4]
-- ---------------------------------------------------------------------

-- 1ª definición:
puntas1 :: [Int] -> [Int]
puntas1 [] = []
puntas1 (x:xs) | x == maximum (x:xs) = x : puntas1 xs
               | otherwise           = puntas1 xs

-- 2ª definición (sin usar maximum):
puntas2 :: [Int] -> [Int]
puntas2 [] = []
puntas2 (x:xs) | all (<=x) xs = x : puntas2 xs
               | otherwise    = puntas2 xs

-- 3ª definición (por plegado):
puntas3 :: [Int] -> [Int]
puntas3 = foldr f []
    where f x ys | all (<= x) ys = x:ys
                 | otherwise     = ys

-- 4ª definición (por plegado y acumulador):
puntas4 :: [Int] -> [Int]
puntas4 xs = foldl f [] (reverse xs)
    where f ac x | all (<=x) ac = x:ac
                 | otherwise    = ac

-- Nota: Comparación de eficiencia
--    ghci> let xs = [1..4000] in last (puntas1 (xs ++ reverse xs))
--    1
--    (3.58 secs, 2,274,856,232 bytes)
--    ghci> let xs = [1..4000] in last (puntas2 (xs ++ reverse xs))
--    1
--    (2.27 secs, 513,654,880 bytes)
--    ghci> let xs = [1..4000] in last (puntas3 (xs ++ reverse xs))
--    1
--    (2.54 secs, 523,669,416 bytes)
--    ghci> let xs = [1..4000] in last (puntas4 (xs ++ reverse xs))
--    1
--    (2.48 secs, 512,952,728 bytes)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. Consideremos el tipo de los árboles binarios con enteros
-- en las hojas y nodos, definido por:
--    data Arbol1 = H1 Int
--                | N1 Int Arbol1 Arbol
--                deriving (Eq, Show)
-- y el tipo de árbol binario de hojas vacías y enteros en los nodos,
-- definido por
--    data Arbol2 = H2
--                | N2 Int Arbol2 Arbol2
--                deriving (Eq, Show)
--
-- Por ejemplo, los árboles
--         5                          10
--        / \                        / \
--       /   \                      /   \
--      3     7                    5     15
--     / \   / \                  /\     /\
--    1   4 6   9                .  .   .  .
-- se definen por
--    ejArbol1 :: Arbol1
--    ejArbol1 = N1 5 (N1 3 (H1 1) (H1 4)) (N1 7 (H1 6) (H1 9))
--    ejArbol2 :: Arbol2
--    ejArbol2 = N2 10 (N2 5 H2 H2) (N2 15 H2 H2)
--
-- Definir la función
--    comprime :: Arbol1 -> Arbol2
-- tal que (comprime a) es el árbol obtenido sustituyendo cada nodo por
-- la suma de sus hijos. Por ejemplo,
--    ghci> comprime ejArbol1
--    N2 10 (N2 5 H2 H2) (N2 15 H2 H2)
--    ghci> comprime ejArbol1 == ejArbol2
--    True
-- ---------------------------------------------------------------------

data Arbol1 = H1 Int
            | N1 Int Arbol1 Arbol1
            deriving (Eq, Show)
```

```haskell
data Arbol2 = H2
            | N2 Int Arbol2 Arbol2
            deriving (Eq, Show)

ejArbol1 :: Arbol1
ejArbol1 = N1 5 (N1 3 (H1 1) (H1 4)) (N1 7 (H1 6) (H1 9))

ejArbol2 :: Arbol2
ejArbol2 = N2 10 (N2 5 H2 H2) (N2 15 H2 H2)

comprime :: Arbol1 -> Arbol2
comprime (H1 x)     = H2
comprime (N1 x i d) = N2 (raiz i + raiz d) (comprime i) (comprime  d)

raiz :: Arbol1 -> Int
raiz (H1 x)     = x
raiz (N1 x _ _) = x
```

2.3. Examen 3 (25 de enero de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 3º examen de evaluación continua (25 de enero de 2016)
-- ---------------------------------------------------------------------

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

import Data.Char
import Data.List
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número n es autodescriptivo cuando para cada posición
-- k de n (empezando a contar las posiciones a partir de 0), el dígito
-- en la  posición k es igual al número de veces que ocurre k en n. Por
-- ejemplo, 1210 es autodescriptivo porque tiene 1 dígito igual a "0", 2
-- dígitos iguales a "1", 1 dígito igual a "2" y ningún dígito igual a
-- "3".
--
```

```haskell
-- Definir la función
--    autodescriptivo :: Integer -> Bool
-- tal que (autodescriptivo n) se verifica si n es autodescriptivo. Por
-- ejemplo,
--    autodescriptivo 1210                      == True
--    [x | x <- [1..100000], autodescriptivo x] == [1210,2020,21200]
--    autodescriptivo 9210000001000             == True
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

autodescriptivo1 :: Integer -> Bool
autodescriptivo1 n = autodescriptiva (digitos n)

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

autodescriptiva :: [Integer] -> Bool
autodescriptiva ns =
    and [x == ocurrencias k ns | (k,x) <- zip [0..] ns]

ocurrencias :: Integer -> [Integer] -> Integer
ocurrencias x ys = genericLength (filter (==x) ys)

-- 2ª solución
-- ===========

autodescriptivo2 :: Integer -> Bool
autodescriptivo2 n =
    and [length [y | y <- xs, read [y] == k] == read [x] |
         (x,k) <- zip xs [0..]]
    where xs = show n

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

--    ghci> [x | x <- [1..100000], autodescriptivo1 x]
--    [1210,2020,21200]
--    (6.94 secs, 3,380,620,264 bytes)
```

```
--    ghci> [x | x <- [1..100000], autodescriptivo2 x]
--    [1210,2020,21200]
--    (7.91 secs, 4,190,791,608 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir el procedimiento
--    autodescriptivos :: IO ()
-- que pregunta por un número n y escribe la lista de los n primeros
-- números autodescriptivos. Por ejemplo,
--    ghci> autodescriptivos
--    Escribe un numero: 2
--    Los 2 primeros numeros autodescriptivos son [1210,2020]
--    ghci> autodescriptivos
--    Escribe un numero: 3
--    Los 3 primeros numeros autodescriptivos son [1210,2020,21200]
-- ---------------------------------------------------------------------

autodescriptivos :: IO ()
autodescriptivos = do
  putStr "Escribe un numero: "
  xs <- getLine
  putStr ("Los " ++ xs ++ " primeros numeros autodescriptivos son ")
  putStrLn (show (take (read xs) [a | a <- [1..], autodescriptivo1 a]))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Dos enteros positivos a y b se dirán relacionados si
-- poseen, exactamente, un factor primo en común. Por ejemplo, 12 y 20
-- 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?
-- ---------------------------------------------------------------------

-- 1ª solución
```

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

paresRel1 :: [(Int,Int)]
paresRel1 = [(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)) paresRel1)
--    2016

-- 2ª solución
-- ===========

paresRel2 :: [(Int,Int)]
paresRel2 = [(x,y) | y <- [4..], x <- [2..y-2], rel x y]
    where rel x y = m /= 1 && all (== head ps) ps
            where m  = gcd x y
                  ps = primeFactors m

-- 3ª solución
-- ===========

paresRel3 :: [(Int,Int)]
paresRel3 =
    [(x,y) | y <- [2..], x <- [2..y-1], relacionados3 x y]

relacionados3 :: Int -> Int -> Bool
relacionados3 x y =
    length (group (primeFactors (gcd x y))) == 1

-- Comparación de eficiencia
--    ghci> paresRel1 !! 40000
--    (216,489)
--    (3.19 secs, 1,825,423,056 bytes)
--    ghci> paresRel2 !! 40000
--    (216,489)
--    (0.96 secs, 287,174,864 bytes)
```

```
--    ghci> paresRel3 !! 40000
--    (216,489)
--    (0.70 secs, 264,137,928 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    agrupa :: (a -> Bool) -> [a] -> [a]
-- tal que (agrupa p xs) es la lista obtenida separando los elementos
-- consecutivos de xs que verifican la propiedad p de los que no la
-- verifican. Por ejemplo,
--    agrupa odd  [1,2,0,4,9,6,4,5,7,2]  ==  [[1],[2,0,4],[9],[6,4],[5,7],[2]]
--    agrupa even [1,2,0,4,9,6,4,5,7,2]  ==  [[],[1],[2,0,4],[9],[6,4],[5,7],[2]]
--    agrupa (>4) [1,2,0,4,9,6,4,5,7,2]  ==  [[],[1,2,0,4],[9,6],[4],[5,7],[2]]
--    agrupa (<4) [1,2,0,4,9,6,4,5,7,2]  ==  [[1,2,0],[4,9,6,4,5,7],[2]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles binarios con datos en nodos y hojas se
-- definen por
--    data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
-- Por ejemplo, el árbol
--           3
--          / \
--         /   \
--        4     7
--       / \   / \
--      5   0 0   3
--     / \
--    2   0
-- se representa por
--    ejArbol :: Arbol Integer
--    ejArbol = N 3 (N 4 (N 5 (H 2)(H 0)) (H 0)) (N 7 (H 0) (H 3))
--
-- Definir la función
--    sucesores :: Arbol a -> [(a,[a])]
-- tal que (sucesores t) es la lista de los pares formados por los
```

```haskell
-- elementos del árbol t junto con sus sucesores. Por ejemplo,
--    ghci> sucesores ejArbol
--    [(3,[4,7]),(4,[5,0]),(5,[2,0]),(2,[]),(0,[]),(0,[]),
--     (7,[0,3]),(0,[]),(3,[])]
-- ---------------------------------------------------------------------

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))

sucesores :: Arbol a -> [(a,[a])]
sucesores (H x)     = [(x,[])]
sucesores (N x i d) = (x, [raiz i, raiz d]) : sucesores i ++ sucesores d

raiz :: Arbol a -> a
raiz (H x)      = x
raiz (N x _ _ ) = x
```

.

2.4. Examen 4 (11 de marzo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 4º examen de evaluación continua (11 de marzo de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import I1M.Cola
import Data.Array
import qualified Data.Matrix as M
import Data.Set (fromList, notMember)

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. En la siguiente figura, al rotar girando 90 grados en
-- el sentido del reloj la matriz de la izquierda, obtenemos 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
--    rota1 :: Array (Int,Int) Int  -> Array (Int,Int) Int
-- tal que (rota p) es la matriz obtenida girando en el sentido del
-- reloj la matriz cuadrada p. Por ejemplo,
--    ghci> elems (rota1 (listArray ((1,1),(3,3))[1..9]))
--    [7,4,1,8,5,2,9,6,3]
--    ghci> elems (rota1 (listArray ((1,1),(3,3))[7,4,1,8,5,2,9,6,3]))
--    [9,8,7,6,5,4,3,2,1]
-- ---------------------------------------------------------------------

rota1 :: Array (Int,Int) Int  -> Array (Int,Int) Int
rota1 p =
    array ((1,1),(n,m)) [((j,n+1-i),p!(i,j)) | i <- [1..m], j <- [1..n]]
    where (_,(m,n)) = bounds p

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    rota2 :: M.Matrix Int  -> M.Matrix Int
-- tal que (rota p) es la matriz obtenida girando en el sentido del
-- reloj la matriz cuadrada p. Por ejemplo,
--    ghci> rota2 (M.fromList 3 3 [1..9])
--    ( 7 4 1 )
--    ( 8 5 2 )
--    ( 9 6 3 )
--
--    ghci> rota2 (M.fromList 3 3 [7,4,1,8,5,2,9,6,3])
--    ( 9 8 7 )
--    ( 6 5 4 )
--    ( 3 2 1 )
-- ---------------------------------------------------------------------

rota2 :: M.Matrix Int  -> M.Matrix Int
rota2 p = M.matrix n m (\(i,j) -> p M.! (n+1-j,i))
    where m = M.nrows p
          n = M.ncols p
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Se dice que un número tiene una bajada cuando existe
-- un par de dígitos (a,b) tales que b está a la derecha de a y b es
-- menor que a. Por ejemplo, 4312 tiene 5 bajadas ((4,3), (4,1), (4,2),
-- (3,1) y (3,2)).
--
-- Definir la función
--    bajadas :: Int -> Int
-- tal que (bajadas n) es el número de bajadas de n. Por ejemplo,
--    bajadas 4312  == 5
--    bajadas 2134  == 1
-- ---------------------------------------------------------------------

bajadas :: Int -> Int
bajadas n = sum (map aux (tails (show n)))
    where aux []     = 0
          aux (x:xs) = length (filter (<x) xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Calcular cuántos números hay de 4 cifras con más de
-- dos bajadas.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    penultimo :: Cola a -> a
-- tal que (penultimo c) es el penúltimo elemento de la cola c. Si
-- la cola esta vacía o tiene un sólo elemento, dará el error
-- correspondiente, "cola vacia" o bien "cola unitaria". Por ejemplo,
--    ghci> penultimo vacia
--    *** Exception: cola vacia
--    ghci> penultimo (inserta 2 vacia)
--    *** Exception: cola unitaria
--    ghci> penultimo (inserta 3 (inserta 2 vacia))
--    2
```

```haskell
--    ghci> penultimo (inserta 5 (inserta 3 (inserta 2 vacia)))
--    3
-- ---------------------------------------------------------------------

penultimo :: Cola a -> a
penultimo c
    | esVacia c   = error "cola vacia"
    | esVacia rc  = error "cola unitaria"
    | esVacia rrc = primero c
    | otherwise   = penultimo rc
    where rc  = resto c
          rrc = resto rc

-- ---------------------------------------------------------------------
-- Ejercicio 4. Sea xs una lista y n su longitud. Se dice que xs es casi
-- completa si sus elementos son los numeros enteros entre 0 y n excepto
-- uno. Por ejemplo, la lista [3,0,1] es casi completa.
--
-- Definir la función
--    ausente :: [Integer] -> Integer
-- tal que (ausente xs) es el único entero (entre 0 y la longitud de xs)
-- que no pertenece a la lista casi completa xs. Por ejemplo,
--    ausente [3,0,1]               ==  2
--    ausente [1,2,0]               ==  3
--    ausente (1+10^7:[0..10^7-1])  ==  10000000
-- ---------------------------------------------------------------------

-- 1ª definición
ausente1 :: [Integer] -> Integer
ausente1 xs =
    head [n | n <- [0..], n `notElem` xs]

-- 2ª definición
ausente2 :: [Integer] -> Integer
ausente2 xs =
    head [n | n <- [0..], n `notMember` ys]
    where ys = fromList xs

-- 3ª definición (lineal)
ausente3 :: [Integer] -> Integer
```

```haskell
ausente3 xs =
    ((n * (n+1)) `div` 2) - sum xs
    where n = genericLength xs

-- 4ª definición
ausente4 :: [Integer] -> Integer
ausente4 xs =
    ((n * (n+1)) `div` 2) - foldl' (+) 0 xs
    where n = genericLength xs

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

--    ghci> let n = 10^5 in ausente1 (n+1:[0..n-1])
--    100000
--    (68.51 secs, 25,967,840 bytes)
--    ghci> let n = 10^5 in ausente2 (n+1:[0..n-1])
--    100000
--    (0.12 secs, 123,488,144 bytes)
--    ghci> let n = 10^5 in ausente3 (n+1:[0..n-1])
--    100000
--    (0.07 secs, 30,928,384 bytes)
--    ghci> let n = 10^5 in ausente4 (n+1:[0..n-1])
--    100000
--    (0.02 secs, 23,039,904 bytes)
--
--    ghci> let n = 10^7 in ausente2 (n+1:[0..n-1])
--    10000000
--    (14.32 secs, 15,358,509,280 bytes)
--    ghci> let n = 10^7 in ausente3 (n+1:[0..n-1])
--    10000000
--    (5.57 secs, 2,670,214,936 bytes)
--    ghci> let n = 10^7 in ausente4 (n+1:[0..n-1])
--    10000000
--    (3.36 secs, 2,074,919,184 bytes)
```

2.5. Examen 5 (6 de mayo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 5º examen de evaluación continua (6 de mayo de 2016)
```

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

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

import Data.List (nub, isPrefixOf)
import Data.Ix
import Test.QuickCheck
import Data.Matrix
import I1M.Grafo

-- ---------------------------------------------------------------------
-- Ejercicio 1. Los números ondulados son aquellos tales que sólo tienen
-- dos dígitos distintos como máximo que se repiten periódicamente. Por
-- ejemplo, 23232, 8989, 363, 38 son ondulados.
--
-- Definir la función
--    esOndulado :: Integer -> Bool
-- tal que (esOndulado n) se verifica si n es ondulado. Por ejemplo,
--    esOndulado 12     = True
--    esOndulado 312    = False
--    esOndulado 313    = True
--    esOndulado 313131 = True
--    esOndulado 54543  = False
--    esOndulado 3      = True
-- ---------------------------------------------------------------------

esOndulado :: Integer -> Bool
esOndulado n =
  length (nub xs) <= 2 &&
  xs `isPrefixOf` (cycle (take 2 xs))
  where xs = show n

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la funcion
--    onda :: Integer -> Matrix Integer
-- tal que (onda x) es la matriz cuadrada cuyas filas son los dígitos de
-- x. Por ejemplo,
--    ghci> onda 323
```

```
--    ( 3 2 3 )
--    ( 3 2 3 )
--    ( 3 2 3 )
--
--    ghci> onda 46464
--    ( 4 6 4 6 4 )
--    ( 4 6 4 6 4 )
--    ( 4 6 4 6 4 )
--    ( 4 6 4 6 4 )
--    ( 4 6 4 6 4 )
-- ---------------------------------------------------------------------

onda :: Integer -> Matrix Integer
onda x = fromLists (replicate n ds)
  where ds = digitos x
        n  = length ds

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la funcion
--    esOnda :: Matrix Integer -> Bool
-- tal que (esOnda p) se verifica si es existe un número ondulado x tal
-- que p es (onda x). Por ejemplo.
--    esOnda (fromLists [[3,2,3],[3,2,3],[3,2,3]])  ==  True
--    esOnda (fromLists [[3,2,2],[3,2,2],[3,2,2]])  ==  False
--    esOnda (fromLists [[3,2,3],[3,2,3]])          ==  False
-- ---------------------------------------------------------------------

esOnda :: Matrix Integer -> Bool
esOnda p = esOndulado x && p == onda x
  where x = digitosAnumero [p!(1,j) | j <- [1..ncols p]]

-- (digitosAnumero xs) es el número cuyos dígitos son los elementos de
-- xs. Por ejemplo,
--    digitosAnumero [3,2,5]  ==  325
digitosAnumero :: [Integer] -> Integer
digitosAnumero xs =
  read (concatMap show xs)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que toda matriz generada por
-- un número  onduladoa es una onda.
-- ---------------------------------------------------------------------

-- ondulado es un generador de números ondulados. Por ejemplo,
--    ghci> sample genOndulado
--    2
--    757
--    16161
--    6
--    86
--    686
--    4646
--    5959595
--    56565656565
--    2929292929292
--    8585858585
ondulado :: Gen Integer
ondulado = do
  x <- choose (1,9)
  y <- choose (0,9)
  n <- arbitrary
  return (digitosAnumero (take (1 + abs n) (cycle [x,y])))

-- La propiedad es
prop_ondas :: Property
prop_ondas =
  forAll ondulado (\x -> esOnda (onda x))

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles binarios con datos en los nodos y hojas se
-- definen por
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
```

```
--                    deriving (Eq, 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))
-- Si asociamos a cada elemento del árbol anterior su profundidad dentro
-- del mismo, se obtiene el árbol siguiente
--
--    profundidad 0:           (3,0)
--                              /  \
--                             /    \
--                            /      \
--    profundidad 1:       (4,1)    (7,1)
--                          / \      /  \
--    profundidad 2:     (5,2)(0,2)(0,2)(3,2)
--                        / \
--    profundidad 3: (2,3) (0,3)
--
-- Definir la función
--    profArbol :: Arbol a -> Arbol (a,Int)
-- tal que (profArbol x) es el árbol obtenido asociando los elementos de
-- x a su profundidad. Por ejemplo,
--   ghci> profArbol ejArbol
--   N (3,0)
--     (N (4,1)
--        (N (5,2) (H (2,3)) (H (0,3)))
--        (H (0,2)))
--     (N (7,1) (H (0,2)) (H (3,2)))
-- ----------------------------------------------------------------------

data Arbol a = H a
```

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

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

profArbol :: Arbol a -> Arbol (a,Int)
profArbol (H x)     = H (x,0)
profArbol (N x i d) = aux (N x i d) 0
  where aux (H x)     n = H (x,n)
        aux (N x i d) n = N (x,n) (aux i (n+1)) (aux d (n+1))

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. 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) [(1,2,3),(1,3,2),(2,3,5)]
--    g2 = creaGrafo ND (1,4) [(1,2,3),(1,3,2),(1,4,5),(2,4,7),(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) [(1,2,3),(1,3,2),(2,3,5)]
g2 = creaGrafo ND (1,4) [(1,2,3),(1,3,2),(1,4,5),(2,4,7),(3,4,0)]

camino :: (Ix a, Num t) => (Grafo a t) -> [a] -> Bool
camino g vs = all (aristaEn g) (zip vs (tail vs))

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    coste :: (Ix a, Num t) => (Grafo a t) -> [a] -> Maybe t
-- tal que (coste g vs) es la suma de los pesos de las aristas del
```

```haskell
-- camino vs en el grafo g o Nothing, si vs no es un camino en g. Por
-- ejemplo,
--    coste g1 [1,2,3]  ==  Just 8
--    coste g2 [1,2,3]  ==  Nothing
-- ---------------------------------------------------------------------

coste :: (Ix a, Num t) => (Grafo a t) -> [a] -> Maybe t
coste g vs
  | camino g vs = Just (sum [peso x y g | (x,y) <- zip vs (tail vs)])
  | otherwise   = Nothing
```

2.6. Examen 6 (7 de junio de 2016)

El examen es común con el del grupo 1 (ver página 51).

2.7. Examen 7 (23 de junio de 2016)

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

2.8. Examen 8 (01 de septiembre de 2016)

El examen es común con el del grupo 1 (ver página 70).

3

Exámenes del grupo 3

Francisco J. Martín

3.1. Examen 1 (27 de Octubre de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (27 de octubre de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir por comprensión la función
--    posiciones :: Int -> [Int] -> [Int]
-- tal que (posiciones x xs) es la lista de las posiciones (contando desde 0)
-- de las ocurrencias del elemento x en la lista xs. Por ejemplo,
--    posiciones 2 [1,2,3,4]                  ==  [1]
--    posiciones 2 [1,2,3,4,1,2,3,1]          ==  [1,5]
--    posiciones 2 [1,2,3,4,1,2,3,1,3,2,4,2]  ==  [1,5,9,11]
-- ---------------------------------------------------------------------

posiciones :: Int -> [Int] -> [Int]
posiciones x xs =
    [i | (y,i) <- zip xs [0..], x == y]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una forma de aproximar el número pi es usando la
-- siguiente igualdad:
--
--            pi         1     1*2     1*2*3     1*2*3*4
--            --- = 1 + --- + ----- + ------- + --------- + ....
```

```
--            2         3     3*5     3*5*7     3*5*7*9
--
-- Es decir, la serie cuyo término general n-ésimo es el cociente entre el
-- producto de los primeros n números y los primeros n números impares:
--
--                        Product i
--           s(n) =  -----------------
--                   Product (2*i+1)
--
-- Definir por comprensión la función:
--    aproximaPi :: Double -> Double
-- tal que (aproximaPi n) es la aproximación del número pi calculada con la
-- serie anterior hasta el término n-ésimo. Por ejemplo,
--    aproximaPi 10   ==  3.141106021601377
--    aproximaPi 30   ==  3.1415926533011587
--    aproximaPi 50   ==  3.1415926535897922
-- ---------------------------------------------------------------------

aproximaPi :: Double -> Double
aproximaPi n =
    2*(sum [product [1..m] / product [2*i+1 | i <- [1..m]] | m <- [0..n] ])

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir por recursión la función
--    subPar :: Integer -> Integer
-- tal que (subPar n) es el número formado por las cifras que ocupan una
-- posición par (contando desde 0 por las unidades) en n en el mismo
-- orden. Por ejemplo,
--    subPar 123        ==     13
--    subPar 123456     ==    246
--    subPar 123456789  ==  13579
-- ---------------------------------------------------------------------

-- 1ª solución
subPar :: Integer -> Integer
subPar 0 = 0
subPar n = (subPar (n `div` 100))*10 + (n `mod` 10)

-- 2ª solución
subPar2 :: Integer -> Integer
```

```haskell
subPar2 n = read (reverse (aux (reverse (show n))))
    where aux (x:y:zs) = x : aux zs
          aux xs       = xs

-- 3ª solución
subPar3 :: Integer -> Integer
subPar3 n = aux (digitos n)
    where aux []       = 0
          aux [x]      = x
          aux (x:y:zs) = x + 10 * aux zs

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir por recursión la función
--    subImpar :: Integer -> Integer
-- tal que (subImpar n) es el número formado por las cifras que ocupan una
-- posición impar (contando desde 0 por las unidades) en n en el mismo
-- orden. Por ejemplo,
--    subImpar 123        ==     2
--    subImpar 123456     ==   135
--    subImpar 123456789  ==  2468
-- ---------------------------------------------------------------------

-- 1ª solución
subImpar :: Integer -> Integer
subImpar 0 = 0
subImpar n = (subImpar (n `div` 100))*10 + ((n `mod` 100) `div` 10)

-- 2ª solución
subImpar2 :: Integer -> Integer
subImpar2 n = read (reverse (aux (reverse (show n))))
    where aux (x:y:zs) = y : aux zs
          aux xs       = []

-- 3ª solución
subImpar3 :: Integer -> Integer
subImpar3 n = aux (digitos n)
```

```
    where aux []       = 0
          aux [x]      = 0
          aux (x:y:zs) = y + 10 * aux zs

-- ---------------------------------------------------------------------
-- Ejercicio 4. Una forma de aproximar el número e es usando la
-- siguiente igualdad:
--
--                      1     1     1     1
--           e  = 1 + --- + --- + --- + --- + ....
--                      1!    2!    3!    4!
--
-- Es decir, la serie cuyo término general n-ésimo es el recíproco del
-- factorial de n:
--
--                    1
--           s(n) =  ---
--                    n!
--
-- Definir por recursión la función:
--    aproximaE :: Double -> Double
-- tal que (aproximaE n) es la aproximación del número e calculada con la
-- serie anterior hasta el término n-ésimo. Por ejemplo,
--    aproximaE 5   ==  2.7166666666666663
--    aproximaE 10  ==  2.7182818011463845
--    aproximaE 15  ==  2.718281828458995
--    aproximaE 20  ==  2.7182818284590455
-- ---------------------------------------------------------------------

aproximaE :: Double -> Double
aproximaE 0 = 1
aproximaE n = (1/(product [1..n])) + aproximaE (n-1)
```

3.2. Examen 2 (2 de Diciembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (2 de diciembre de 2015)
-- ---------------------------------------------------------------------

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

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Una lista hermanada es una lista de números
-- estrictamente positivos en la que cada elemento tiene algún factor
-- primo en común con el siguiente, en caso de que exista, o alguno de
-- los dos es un 1. Por ejemplo,
--    + [2,6,3,9,1,5] es una lista hermanada pues 2 y 6 tienen un factor
--      en común (2); 6 y 3 tienen un factor en común (3); 3 y 9 tienen un
--      factor en común (3); de 9 y 1 uno es el número 1; y de 1 y 5 uno
--      es el número 1.
--    + [2,3,5] no es una lista hermanada pues 2 y 3 no tienen ningún
--      factor primo en común.
--
-- Definir, por comprensión, la función
--    hermanadaC :: [Int] -> Bool
-- tal que (hermanada xs) se verifica si la lista xs es hermanada según la
-- definición anterior. Por ejemplo,
--    hermanadaC [2,6,3,9,1,5]  ==  True
--    hermanadaC [2,3,5]        ==  False
-- ---------------------------------------------------------------------

hermanadaC :: [Int] -> Bool
hermanadaC xs = and [hermanos p | p <- zip xs (tail xs)]

-- (hermanos (x,y)) se verifica si x e y son hermanos; es decir, alguno es
-- igual a 1 o tienen algún factor primo en común
hermanos :: (Int, Int) -> Bool
hermanos (x,y) = x == 1 || y == 1 || gcd x y /= 1

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, usando funciones de orden superior, la función
--    hermanadaS :: [Int] -> Bool
-- tal que (hermanada xs) se verifica si la lista xs es hermanada según la
-- definición anterior. Por ejemplo,
--    hermanadaS [2,6,3,9,1,5]  ==  True
```

```haskell
--    hermanadaS [2,3,5]        ==  False
-- ---------------------------------------------------------------------

hermanadaS :: [Int] -> Bool
hermanadaS xs = all hermanos (zip xs (tail xs))

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por recursión, la función
--    hermanadaR :: [Int] -> Bool
-- tal que (hermanada xs) se verifica si la lista xs es hermanada según la
-- definición anterior. Por ejemplo,
--    hermanadaR [2,6,3,9,1,5]  ==  True
--    hermanadaR [2,3,5]        ==  False
-- ---------------------------------------------------------------------

hermanadaR :: [Int] -> Bool
hermanadaR (x1:x:xs) = hermanos (x1,x) && hermanadaR (x:xs)
hermanadaR _         = True

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir, por plegado, la función
--    hermanadaP :: [Int] -> Bool
-- tal que (hermanada xs) se verifica si la lista xs es hermanada según la
-- definición anterior. Por ejemplo,
--    hermanadaP [2,6,3,9,1,5]  ==  True
--    hermanadaP [2,3,5]        ==  False
-- ---------------------------------------------------------------------

hermanadaP :: [Int] -> Bool
hermanadaP xs =
    foldl (\ws p -> hermanos p && ws) True (zip xs (tail xs))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión con acumulador, la función
--    sumaEnPosicion :: [Int] -> [Int] -> Int
-- tal que (sumaEnPosicion xs ys) es la suma de todos los elementos de xs
-- cuyas posiciones se indican en ys. Si alguna posición no existe en xs
-- entonces el valor se considera nulo. Por ejemplo,
--    sumaEnPosicion [1,2,3] [0,2]  ==  4
--    sumaEnPosicion [4,6,2] [1,3]  ==  6
```

```
--    sumaEnPosicion [3,5,1] [0,1]  ==  8
-- ---------------------------------------------------------------------

sumaEnPosicion :: [Int] -> [Int] -> Int
sumaEnPosicion xs ys = aux xs [y | y <- ys, 0 <= y, y < n] 0
    where n = length xs
          aux _  []     r = r
          aux xs (y:ys) r = aux xs ys (r + xs!!y)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    productoInfinito :: [Int] -> [Int]
-- tal que (productoInfinito xs) es la lista infinita que en la posición
-- N tiene el producto de los N primeros elementos de la lista infinita
-- xs. Por ejemplo,
--    take 5 (productoInfinito [1..])    ==  [1,2,6,24,120]
--    take 5 (productoInfinito [2,4..])  ==  [2,8,48,384,3840]
--    take 5 (productoInfinito [1,3..])  ==  [1,3,15,105,945]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
productoInfinito1 :: [Integer] -> [Integer]
productoInfinito1 xs = [product (take n xs) | n <- [1..]]

-- 2ª definición (por recursión)
productoInfinito2 :: [Integer] -> [Integer]
productoInfinito2 (x:y:zs) = x : productoInfinito2 (x*y:zs)

-- 2ª definición (por recursión y map)
productoInfinito3 :: [Integer] -> [Integer]
productoInfinito3 []     = [1]
productoInfinito3 (x:xs) = map (x*) (1 : productoInfinito3 xs)

-- 4ª definición (con scanl1)
productoInfinito4 :: [Integer] -> [Integer]
productoInfinito4 = scanl1 (*)

-- Comparación de eficiencia
--    ghci> take 20 (show (productoInfinito1 [2,4..] !! 10000))
--    "11358071114466915693"
```

```
--    (0.35 secs, 98,287,328 bytes)
--    ghci> take 20 (show (productoInfinito2 [2,4..] !! 10000))
--    "11358071114466915693"
--    (0.35 secs, 98,840,440 bytes)
--    ghci> take 20 (show (productoInfinito3 [2,4..] !! 10000))
--    "11358071114466915693"
--    (7.36 secs, 6,006,360,472 bytes)
--    ghci> take 20 (show (productoInfinito4 [2,4..] !! 10000))
--    "11358071114466915693"
--    (0.34 secs, 96,367,000 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    siembra :: [Int] -> [Int]
-- tal que (siembra xs) es la lista ys obtenida al repartir cada
-- elemento x de la lista xs poniendo un 1 en las x siguientes
-- posiciones de la lista ys. Por ejemplo,
--    siembra [4]      ==  [0,1,1,1,1]
--    siembra [0,2]    ==  [0,0,1,1]
--    siembra [4,2]    ==  [0,1,2,2,1]
-- El tercer ejemplo se obtiene sumando la siembra de 4 en la posición 0
-- (como el ejemplo 1) y el 2 en la posición 1 (como el ejemplo 2).
-- Otros ejemplos son
--    siembra [0,4,2]  ==  [0,0,1,2,2,1]
--    siembra [3]      ==  [0,1,1,1]
--    siembra [3,4,2]  ==  [0,1,2,3,2,1]
--    siembra [3,2,1]  ==  [0,1,2,3]
--
-- Comprobar con QuickCheck que la suma de los elementos de (siembra xs)
-- es igual que la suma de los de xs.
--
-- Nota: Se supone que el argumento es una lista de números no negativos
-- y que se puede ampliar tanto como sea necesario para repartir los
-- elementos.
-- ---------------------------------------------------------------------

siembra :: [Int] -> [Int]
siembra [] = []
siembra (x:xs) = mezcla (siembraElemento x) (0 : siembra xs)
```

```haskell
siembraElemento :: Int -> [Int]
siembraElemento x = 0 : replicate x 1

mezcla :: [Int] -> [Int] -> [Int]
mezcla xs ys =
    take (max (length xs) (length ys))
         (zipWith (+) (xs ++ repeat 0) (ys ++ repeat 0))

-- La propiedad es
prop_siembra :: [Int] -> Bool
prop_siembra xs =
    sum (siembra ys) == sum ys
    where ys = map (\x -> 1 + abs x) xs

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

3.3. Examen 3 (25 de enero de 2016)

El examen es común con el del grupo 2 (ver página 83).

3.4. Examen 4 (15 de marzo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (15 de marzo de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Array
import qualified Data.Matrix as M

import Test.QuickCheck

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

```haskell
-- Ejercicio 1. Definir la función
--    sumaMaxima :: [Integer] -> Integer
-- tal que (sumaMaxima xs) es el valor máximo de la suma de elementos
-- consecutivos de la lista xs. Por ejemplo,
--    sumaMaxima []             ==  0
--    sumaMaxima [-1,-2,-3]     ==  -1
--    sumaMaxima [2,-2,3,-3,4]  ==  4
--    sumaMaxima [2,-1,3,-2,3]  ==  5
--    sumaMaxima [1,-1,3,-2,4]  ==  5
--    sumaMaxima [2,-1,3,-2,4]  ==  6
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

sumaMaxima1 :: [Integer] -> Integer
sumaMaxima1 [] = 0
sumaMaxima1 xs =
    maximum (map sum [sublista xs i j | i <- [0..length xs - 1],
                                        j <- [i..length xs - 1]])

sublista :: [Integer] -> Int -> Int -> [Integer]
sublista xs i j =
    [xs!!k | k <- [i..j]]

-- 2ª definición
-- =============

sumaMaxima2 :: [Integer] -> Integer
sumaMaxima2 [] = 0
sumaMaxima2 xs
    | m <= 0    = m
    | otherwise = sumaMaximaAux 0 0 xs
    where m = maximum xs

sumaMaximaAux :: Integer -> Integer -> [Integer] -> Integer
sumaMaximaAux m v [] =
    max m v
sumaMaximaAux m v (x:xs)
    | x >= 0    = sumaMaximaAux m (v+x) xs
```

```
    | v+x > 0   = sumaMaximaAux (max m v) (v+x) xs
    | otherwise = sumaMaximaAux (max m v) 0 xs

-- 3ª definición
-- =============

sumaMaxima3 :: [Integer] -> Integer
sumaMaxima3 [] = 0
sumaMaxima3 xs = maximum (map sum (segmentos xs))

-- (segmentos xs) es la lista de los segmentos de xs. Por ejemplo
--    segmentos "abc"  ==  ["a","ab","abc","b","bc","c"]
segmentos :: [a] -> [[a]]
segmentos xs =
    concat [tail (inits ys) | ys <- init (tails xs)]

-- 4ª definición
-- =============

sumaMaxima4 :: [Integer] -> Integer
sumaMaxima4 [] = 0
sumaMaxima4 xs =
    maximum (concat [scanl1 (+) ys | ys <- tails xs])

-- Comprobación
-- ============

-- La propiedad es
prop_sumaMaxima :: [Integer] -> Bool
prop_sumaMaxima xs =
    sumaMaxima2 xs == n &&
    sumaMaxima3 xs == n &&
    sumaMaxima4 xs == n
    where n = sumaMaxima1 xs

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

-- Comparación de eficiencia
```

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

--    ghci> let n = 10^2 in sumaMaxima1 [-n..n]
--    5050
--    (2.10 secs, 390,399,104 bytes)
--    ghci> let n = 10^2 in sumaMaxima2 [-n..n]
--    5050
--    (0.02 secs, 0 bytes)
--    ghci> let n = 10^2 in sumaMaxima3 [-n..n]
--    5050
--    (0.27 secs, 147,705,184 bytes)
--    ghci> let n = 10^2 in sumaMaxima4 [-n..n]
--    5050
--    (0.04 secs, 11,582,520 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Una matriz de tipo par-nula es una matriz en la que
-- todas las posiciones cuya suma de índices es par tienen un valor
-- 0. Por ejemplo,
--
--        ( 0 3 0 )
--        ( 5 0 2 )
--
--        ( 0 3 )
--        ( 5 0 )
--        ( 0 4 )
--
--        ( 0 3 0 )
--        ( 5 0 2 )
--        ( 0 4 0 )
--
-- Representaremos las matrices mediante tablas de dos dimensiones cuyos
-- índices son pares de números naturales comenzando desde 1.
--
-- Definir la función
--    matrizParNula :: (Eq a,Num a) => Matriz a -> Bool
-- tal que (matrizParNula m) se verifica si la matriz m es de tipo
-- par-nula. Por ejemplo,
--    matrizParNula (listArray ((1,1),(2,3)) [0,3,0,5,0,2])        ==  True
--    matrizParNula (listArray ((1,1),(2,3)) [0,3,5,0,0,4])        ==  False
```

```
--    matrizParNula (listArray ((1,1),(3,2)) [0,3,5,0,0,4])        ==  True
--    matrizParNula (listArray ((1,1),(3,2)) [0,3,0,5,0,2])        ==  False
--    matrizParNula (listArray ((1,1),(3,3)) [0,3,0,5,0,2,0,4,0])  ==  True
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

matrizParNula :: (Eq a,Num a) => Matriz a -> Bool
matrizParNula m =
    all (== 0) [m!(i,j) | i <- [1..p], j <- [1..q], even (i+j)]
    where (p,q) = snd (bounds m)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    matrizParNulaDistancias :: Int -> Int -> Matriz Int
-- tal que (matrizParNulaDistancias p q) es la matriz de tipo par-nula que en
-- las posiciones no nulas tiene la distancia (número de casillas contando la
-- inicial) más corta desde dicha posición hasta uno de los bordes de la propia
-- matriz. Por ejemplo,
--    matrizParNulaDistancias 3 3  =>
--       ( 0 1 0 )
--       ( 1 0 1 )
--       ( 0 1 0 )
--    matrizParNulaDistancias 3 5  =>
--       ( 0 1 0 1 0 )
--       ( 1 0 2 0 1 )
--       ( 0 1 0 1 0 )
--    matrizParNulaDistancias 5 5  =>
--       ( 0 1 0 1 0 )
--       ( 1 0 2 0 1 )
--       ( 0 2 0 2 0 )
--       ( 1 0 2 0 1 )
--       ( 0 1 0 1 0 )
-- ---------------------------------------------------------------------

matrizParNulaDistancias :: Int -> Int -> Matriz Int
matrizParNulaDistancias p q =
    array ((1,1),(p,q))
          [((i,j),if odd (i+j)
                  then minimum [i,j,p+1-i,q+1-j]
```

```
                  else 0) | i <- [1..p], j <- [1..q]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    permutacionesM :: (Eq a) => [a] -> [[a]]
-- tal que (permutacionesM xs) es la lista de las permutaciones de la
-- lista xs sin elementos consecutivos iguales. Por ejemplo,
--    permutacionesM "abab"   ==  ["abab","baba"]
--    permutacionesM "abc"    ==  ["abc","acb","bac","bca","cab","cba"]
--    permutacionesM "abcab"  ==  ["abcab","abcba","abacb","ababc","acbab",
--                                 "bacab","bacba","babca","babac","bcaba",
--                                 "cabab","cbaba"]
-- ---------------------------------------------------------------------

permutacionesM :: (Eq a) => [a] -> [[a]]
permutacionesM [] = [[]]
permutacionesM xs =
    concat [map (x:) (filter (\ ys -> null ys || head ys /= x)
                             (permutacionesM (borra x xs))) | x <- nub xs]

borra :: (Eq a) => a -> [a] -> [a]
borra x [] = []
borra x (y:xs)
    | x == y = xs
    | otherwise = y : borra x xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Un árbol de Fibonacci de grado N es un árbol binario
-- construido de la siguiente forma:
-- + El árbol de Fibonacci de grado 0 es una hoja con dicho valor:
--      (0)
-- + El árbol de Fibonacci de grado 1 es:
--         (1)
--        /   \
--      (0)   (0)
-- + El árbol de Fibonacci de grado N (> 1) es el árbol que en su raíz
--   tiene el valor N, su hijo izquierdo es el árbol de Fibonacci de
--   grado (N-2) y su hijo derecho es el árbol de Fibonacci de grado
--   (N-1).
-- De esta forma, el árbol de Fibonacci de grado 2 es:
```

```
--          (2)
--         /   \
--       (0)   (1)
--            /   \
--          (0)   (0)
-- el árbol de Fibonacci de grado 3 es:
--                (3)
--              __/   \__
--            /         \
--          (1)         (2)
--         /   \       /   \
--       (0)   (0)   (0)   (1)
--                         /   \
--                       (0)   (0)
-- y el árbol de Fibonacci de grado 4 es:
--                        (4)
--                 _______/   \_______
--               /                   \
--             (2)                   (3)
--            /   \                __/   \__
--          (0)   (1)            /         \
--                /   \        (1)         (2)
--              (0)   (0)     /   \       /   \
--                          (0)   (0)   (0)   (1)
--                                            /   \
--                                          (0)   (0)
--
-- Definir la función
--    esArbolFib :: Arbol -> Bool
-- tal que (esArbolFib a) comprueba si el árbol a es un árbol de Fibonacci.
-- Por ejemplo,
--    esArbolFib (H 0)                          ==  True
--    esArbolFib (H 1)                          ==  False
--    esArbolFib (N 1 (H 0) (H 0))              ==  True
--    esArbolFib (N 2 (H 0) (H 1))              ==  False
--    esArbolFib (N 2 (H 0) (N 1 (H 0) (H 0)))  ==  True
--    esArbolFib (N 3 (H 0) (N 1 (H 0) (H 0)))  ==  False
-- ----------------------------------------------------------------------

data Arbol = H Int
```

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

raiz :: Arbol -> Int
raiz (H n) = n
raiz (N n _ _) = n

esArbolFib :: Arbol -> Bool
esArbolFib (H 0) = True
esArbolFib (N 1 (H 0) (H 0)) = True
esArbolFib (N n ai ad) =
    (n > 1) && raiz ai == n-2 && raiz ad == n-1 &&
    esArbolFib ai && esArbolFib ad
esArbolFib _ = False

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la constante
--    arbolesFib :: [Arbol]
-- tal que (arbolesFib) es la lista infinita que en la posición N tiene
-- el árbol de Fibonacci de grado N. Por ejemplo,
--    arbolesFib!!0  ==  H 0
--    arbolesFib!!1  ==  N 1 (H 0) (H 0)
--    arbolesFib!!2  ==  N 2 (H 0) (N 1 (H 0) (H 0))
--    arbolesFib!!3  ==  N 3 (N 1 (H 0) (H 0)) (N 2 (H 0) (N 1 (H 0) (H 0)))
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

arbolFib1 :: Int -> Arbol
arbolFib1 0 = H 0
arbolFib1 1 = N 1 (H 0) (H 0)
arbolFib1 n = N n (arbolFib1 (n-2)) (arbolFib1 (n-1))

arbolesFib1 :: [Arbol]
arbolesFib1 =
    [arbolFib1 k | k <- [0..]]

-- 2ª solución
-- ===========
```

```haskell
arbolesFib2 :: [Arbol]
arbolesFib2 =
    H 0 :
    N 1 (H 0) (H 0) :
    zipWith3 N [2..] arbolesFib2 (tail arbolesFib2)
```

3.5. Examen 5 (3 de mayo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (3 de mayo de 2016)
-- ============================================================================

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

import Data.List
import Data.Numbers.Primes
import I1M.Cola
import I1M.Pol
import qualified Data.Matrix as M

-- ----------------------------------------------------------------------------
-- Ejercicio 3. Dados dos polinomios P y Q, su producto de grado, que
-- notaremos P (o) Q, es el polinomio que se construye de la siguiente
-- forma: para cada par de términos de P y Q con el mismo grado se
-- considera un término con dicho grado cuyo coeficiente es el producto
-- de los coeficientes de dichos términos. Se muestran algunos ejemplos
-- de producto de grado entre polinomios:
-- + Dos polinomios con un único término con el mismo grado
--      2*x^2 (o) 3*x^2 = 6*x^2
-- + Dos polinomios entre los que no hay términos con el mismo grado
--      2*x^2 + 1 (o) 3*x^3 - x = 0
-- + Dos polinomios entre los que hay dos pares de términos con el mismo grado
--      2*x^2 + x + 2 (o) 3*x^2 - 4*x = 6*x^2 - 4*x
--
-- Definir la función
--    productoDeGrado :: Polinomio Int -> Polinomio Int -> Polinomio Int
-- tal que (productoDeGrado p q) es el producto de grado de los polinomios
```

```haskell
-- p y q. Por ejemplo, dados los polinomios
--    pol1  =  4*x^4 + 5*x^3 + 1
--    pol2  =  6*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
--    pol3  =  -3*x^7 + 3*x^6 + -2*x^4 + 2*x^3 + -1*x + 1
--    pol4  =  -1*x^6 + 3*x^4 + -3*x^2 + 1
-- entonces
--    productoDeGrado pol1 pol1  =>  16*x^4 + 25*x^3 + 1
--    productoDeGrado pol1 pol2  =>  20*x^4 + 20*x^3 + 1
--    productoDeGrado pol1 pol3  =>  -8*x^4 + 10*x^3 + 1
--    productoDeGrado pol3 pol4  =>  -3*x^6 + -6*x^4 + 1
-- ---------------------------------------------------------------------

listaApol :: [Int] -> Polinomio Int
listaApol xs = foldr (\ (n,c) p -> consPol n c p)
                     polCero
                     (zip [0..] xs)

pol1, pol2, pol3, pol4 :: Polinomio Int
pol1 = listaApol [1,0,0,5,4,0]
pol2 = listaApol [1,2,3,4,5,6]
pol3 = listaApol [1,-1,0,2,-2,0,3,-3]
pol4 = listaApol [1,0,-3,0,3,0,-1]

productoDeGrado :: Polinomio Int -> Polinomio Int -> Polinomio Int
productoDeGrado p q
    | esPolCero p = polCero
    | esPolCero q = polCero
    | gp < gq     = productoDeGrado p rq
    | gq < gp     = productoDeGrado rp q
    | otherwise   = consPol gp (cp*cq) (productoDeGrado rp rq)
    where gp = grado p
          gq = grado q
          cp = coefLider p
          cq = coefLider q
          rp = restoPol p
          rq = restoPol q

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Una posición (i,j) de una matriz p es un cruce si
-- todos los elementos de p que están fuera de la fila i y fuera de la
```

```
-- columna j son nulos. Una matriz cruz es una matriz que tiene algún
-- cruce. Por ejemplo, las siguientes matrices cruces
--    ( 0 0 2 0 )      ( 0 0 1 0 0 )      ( 0 0 6 0 0 )
--    ( 3 3 4 2 )      ( 0 0 2 0 0 )      ( 0 0 2 0 0 )
--    ( 0 0 0 0 )      ( 3 3 1 2 0 )      ( 3 5 1 2 5 )
--    ( 0 0 9 0 )      ( 0 0 9 0 0 )
-- ya que la 1ª tiene un cruce en (2,3), la 2ª en (3,3) y la 3ª en
-- (3,3).
--
-- Utilizaremos la librería Matrix para desarrollar este ejercicio.
--
-- Definir la función
--    esMatrizCruz :: Matrix Int -> Bool
-- tal que (esMatrizCruz m) comprueba si la matriz m es una matriz cruz.
-- Por ejemplo, dadas las matrices
--    p1 = M.matrix 3 3 (\ (i,j) -> (if any even [i,j] then 1 else 0))
--    p2 = M.matrix 3 4 (\ (i,j) -> (i+j))
-- entonces
--    esMatrizCruz p1  ==  True
--    esMatrizCruz p2  ==  False
-- ---------------------------------------------------------------------

p1, p2 :: M.Matrix Int
p1 = M.matrix 3 3 (\ (i,j) -> (if any even [i,j] then 1 else 0))
p2 = M.matrix 3 4 (uncurry (+))

-- 1ª definición
-- =============

esMatrizCruz :: M.Matrix Int -> Bool
esMatrizCruz p =
    or [esCruce p m n (i,j) | i <- [1..m], j <- [1..n]]
    where m = M.nrows p
          n = M.ncols p

-- (esCruce p m n (i,j)) se verifica si (i,j) es un cruce de p (que
-- tiene m filas y n columnas)
esCruce :: M.Matrix Int -> Int -> Int -> (Int,Int) -> Bool
esCruce p m n (i,j) =
    all (== 0) [p M.! (x,y) | x <- [1..i-1]++[i+1..m],
```

```
                                y <- [1..j-1]++[j+1..n]]

-- 2ª definición de esCruce
esCruce2 :: M.Matrix Int -> Int -> Int -> (Int,Int) -> Bool
esCruce2 p m n (i,j) =
    M.minorMatrix i j p == M.zero (m-1) (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    matrizCruz :: Int -> Int -> Int -> Int -> M.Matrix Int
-- tal que (matrizCruz m n i j) es la matriz cruz de dimensión (m,n) con
-- respecto a la posición (i,j), en la que el valor de cada elemento no
-- nulo es la distancia en línea recta a la posición (i,j), contando
-- también esta última. Por ejemplo,
--    ghci> matrizCruz 3 3 (2,2)
--    ( 0 2 0 )
--    ( 2 1 2 )
--    ( 0 2 0 )
--
--    ghci> matrizCruz 4 5 (2,3)
--    ( 0 0 2 0 0 )
--    ( 3 2 1 2 3 )
--    ( 0 0 2 0 0 )
-


-    ( 0 0 3 0 0 )
--
--    ghci> matrizCruz 5 3 (2,3)
--    ( 0 0 2 )
--    ( 3 2 1 )
--    ( 0 0 2 )
--    ( 0 0 3 )
--    ( 0 0 4 )
-- ---------------------------------------------------------------------

matrizCruz :: Int -> Int -> (Int,Int) -> M.Matrix Int
matrizCruz m n (i,j) =
    M.matrix m n generador
    where generador (a,b)
```

```
  | a == i    = 1 + abs (j - b)
  | b == j    = 1 + abs (i - a)
  | otherwise = 0
```

3.6. Examen 6 (7 de junio de 2016)

El examen es común con el del grupo 1 (ver página 51).

3.7. Examen 7 (23 de junio de 2016)

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

3.8. Examen 8 (01 de septiembre de 2016)

El examen es común con el del grupo 1 (ver página 70).

4

Exámenes del grupo 4

María J. Hidalgo

4.1. Examen 1 (3 de Noviembre de 2015)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (3 de noviembre de 2015)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La suma de la serie
--    1/3 + 1/15 + 1/35 + 1/63 + ...+ 1/(4*x^2-1) + ...
-- es 1/2.
--
-- Definir la función
--    sumaS2:: Double -> Double
-- tal que (sumaS2 n) es la aproximación de 1/2 obtenida mediante n
-- términos de la serie. Por ejemplo,
--    sumaS2 10  == 0.4761904761904761
--    sumaS2 100 == 0.49751243781094495
-- ---------------------------------------------------------------------

sumaS2 :: Double -> Double
sumaS2 n = sum [1/(4*x^2-1) | x <- [1..n]]

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

```haskell
-- Ejercicio 1.2. Comprobar con QuickCheck que la sumas finitas de la
-- serie siempre son menores que 1/2.
-- ---------------------------------------------------------------------

-- La propiedad es
propSumaS2 :: Double -> Bool
propSumaS2 n = sumaS2 n < 0.5

-- La comprobación es
--    ghci> quickCheck propSumaS2
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    menorNError :: Double -> Double
-- tal que (menorNError x) es el menor número de términos de la serie
-- anterior necesarios para obtener 1/2 con un error menor que x. Por
-- ejemplo,
--    menorNError 0.01  == 25.0
--    menorNError 0.001 == 250.0
-- ---------------------------------------------------------------------

menorNError :: Double -> Double
menorNError x =
    head [n | n <- [1..], 0.5 - sumaS2 n < x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Decimos que un número n es "muy divisible por 3" si es
-- divisible por 3 y sigue siendo divisible por 3 si vamos quitando
-- dígitos por la derecha. Por ejemplo, 96060 es muy divisible por 3
-- porque 96060, 9606, 960, 96 y 9 son todos divisibles por 3.
--
-- Definir la función
--    muyDivPor3 :: Integer -> Bool
-- tal que (muyDivPor3 n) se verifica si n es muy divisible por 3. Por
-- ejemplo,
--    muyDivPor3 96060 == True
--    muyDivPor3 90616 == False
-- ---------------------------------------------------------------------
```

```haskell
muyDivPor3 :: Integer -> Bool
muyDivPor3 n
    | n < 10    = n `rem` 3 == 0
    | otherwise = n `rem` 3 == 0 && muyDivPor3 (n `div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    numeroMuyDivPor3Cifras :: Integer -> Integer
-- tal que (numeroMuyDivPor3CifrasC k) es la cantidad de números de k
-- cifras muy divisibles por 3. Por ejemplo,
--    numeroMuyDivPor3Cifras 5 == 768
--    numeroMuyDivPor3Cifras 7 == 12288
--    numeroMuyDivPor3Cifras 9 == 196608
-- ---------------------------------------------------------------------

-- 1ª definición
numeroMuyDivPor3Cifras :: Integer -> Integer
numeroMuyDivPor3Cifras k =
    genericLength [x | x <- [10^(k-1)..10^k-1], muyDivPor3 x]

-- 2ª definición
numeroMuyDivPor3Cifras2 :: Integer -> Integer
numeroMuyDivPor3Cifras2 k =
    genericLength [x | x <- [n,n+3..10^k-1], muyDivPor3 x]
    where n = k*10^(k-1)

-- 3ª definición
-- =============

numeroMuyDivPor3Cifras3 :: Integer -> Integer
numeroMuyDivPor3Cifras3 k = genericLength (numeroMuyDivPor3Cifras3' k)

numeroMuyDivPor3Cifras3' :: Integer -> [Integer]
numeroMuyDivPor3Cifras3' 1 = [3,6,9]
numeroMuyDivPor3Cifras3' k =
    [10*x+y | x <- numeroMuyDivPor3Cifras3' (k-1),
              y <- [0,3..9]]

-- 4ª definición
-- =============
```

```haskell
numeroMuyDivPor3Cifras4 :: Integer -> Integer
numeroMuyDivPor3Cifras4 1 = 3
numeroMuyDivPor3Cifras4 k = 4 * numeroMuyDivPor3Cifras4 (k-1)

-- 5ª definición
numeroMuyDivPor3Cifras5 :: Integer -> Integer
numeroMuyDivPor3Cifras5 k = 3 * 4^(k-1)

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

--    ghci> numeroMuyDivPor3Cifras 6
--    3072
--    (3.47 secs, 534,789,608 bytes)
--    ghci> numeroMuyDivPor3Cifras2 6
--    2048
--    (0.88 secs, 107,883,432 bytes)
--    ghci> numeroMuyDivPor3Cifras3 6
--    3072
--    (0.01 secs, 0 bytes)
--
--    ghci> numeroMuyDivPor3Cifras2 7
--    0
--    (2.57 secs, 375,999,336 bytes)
--    ghci> numeroMuyDivPor3Cifras3 7
--    12288
--    (0.02 secs, 0 bytes)
--    ghci> numeroMuyDivPor3Cifras4 7
--    12288
--    (0.00 secs, 0 bytes)
--    ghci> numeroMuyDivPor3Cifras5 7
--    12288
--    (0.01 secs, 0 bytes)
--
--    ghci> numeroMuyDivPor3Cifras4 (10^5) `rem` 100000
--    32032
--    (5.74 secs, 1,408,600,592 bytes)
--    ghci> numeroMuyDivPor3Cifras5 (10^5) `rem` 100000
--    32032
--    (0.02 secs, 0 bytes)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir una función
--    intercala:: [a] -> [a] -> [a]
-- tal que (intercala xs ys) es la lista que resulta de intercalar los
-- elementos de xs con los de ys. Por ejemplo,
--    intercala [1..7] [9,2,0,3] == [1,9,2,2,3,0,4,3,5,6,7]
--    intercala [9,2,0,3] [1..7] == [9,1,2,2,0,3,3,4,5,6,7]
--    intercala "hola" "adios"   == "haodliaos"
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. La diferencia simétrica de dos conjuntos es el conjunto
-- cuyos elementos son aquellos que pertenecen a alguno de los conjuntos
-- iniciales, sin pertenecer a ambos a la vez. Por ejemplo, la
-- diferencia simétrica de {2,5,3} y {4,2,3,7} es {5,4,7}.

-- Definir la función
--    diferenciaSimetrica :: Eq a => [a] -> [a] -> [a]
-- tal que (diferenciaSimetrica xs ys) es la diferencia simétrica de xs
-- e ys. Por ejemplo,
--    diferenciaSimetrica [2,5,3] [4,2,3,7]    ==  [5,4,7]
--    diferenciaSimetrica [2,5,3] [5,2,3]      ==  []
--    diferenciaSimetrica [2,5,2] [4,2,3,7]    ==  [5,4,3,7]
--    diferenciaSimetrica [2,5,2] [4,2,4,7]    ==  [5,4,4,7]
--    diferenciaSimetrica [2,5,2,4] [4,2,4,7]  ==  [5,7]
-- ---------------------------------------------------------------------

diferenciaSimetrica :: Eq a => [a] -> [a] -> [a]
diferenciaSimetrica xs ys =
    [x | x <- xs, x `notElem` ys] ++ [y | y <- ys, y `notElem` xs]

-- ---------------------------------------------------------------------
-- Ejercicio 5. (Basado en el problema 4 del Proyecto Euler)
-- El número 9009 es capicúa y es producto de dos números de dos dígitos,
```

```haskell
-- pues 9009 = 91*99.
--
-- Definir la función
--    numerosC2Menores :: Int -> [Int]
-- tal que (numerosC2Menores n) es la lista de números capicúas menores
-- que n que son producto de 2 números de dos dígitos. Por ejemplo,
--    numerosC2Menores 100           == []
--    numerosC2Menores 400           == [121,242,252,272,323,363]
--    length (numerosC2Menores 1000) == 38
-- ---------------------------------------------------------------------

numerosC2Menores :: Int -> [Int]
numerosC2Menores n = [x | x <- productos n, esCapicua x]

-- (productos n) es la lista de números menores que n que son productos
-- de 2 números de dos dígitos.
productos :: Int -> [Int]
productos n =
    sort (nub [x*y | x <- [10..99], y <- [x..99], x*y < n])

-- 2ª definición de productos
productos2 :: Int -> [Int]
productos2 n =
    init (nub (sort [x*y | x <- [10..min 99 (n `div` 10)],
                           y <- [x..min 99 (n `div` x)]]))

-- (esCapicua x) se verifica si x es capicúa.
esCapicua :: Int -> Bool
esCapicua x = xs == reverse xs
    where xs = show x
```

4.2. Examen 2 (3 de Diciembre de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (3 de diciembre de 2015)
-- ---------------------------------------------------------------------

import Data.List
import Data.Numbers.Primes
import Test.QuickCheck
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir una función
--    sumaCuadradosDivisores1 :: Integer -> Integer
-- que calcule la suma de los cuadrados de los divisores de n. Por
-- ejemplo:
--    sumaCuadradosDivisores1 6  ==  50
-- ---------------------------------------------------------------------

sumaCuadradosDivisores1 :: Integer -> Integer
sumaCuadradosDivisores1 n = sum [x^2 | x <- divisores n]

sumaCuadradosDivisores1' :: Integer -> Integer
sumaCuadradosDivisores1' = sum . map (^2) . divisores

-- (divisores n) es la lista de los divisores de n.
divisores :: Integer -> [Integer]
divisores n = [x | x <- [1..n], n `rem` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. La suma de los cuadrados de los divisores de un número
-- se puede calcular a partir de su factorización prima. En efecto, si
-- la factorización prima de n es
--    a^x*b^y*...*c^z
-- entonces, la suma de los divisores de n es
--   (1+a+a^2+...+a^x) * (1+b+b^2+...+b^y) *...* (1+c+c^2+...+c^z)
-- es decir,
--   ((a^(x+1)-1)/(a-1)) * ((b^(y+1)-1)/(b-1)) *...* ((c^(z+1)-1)/(c-1))
-- Por tanto, la suma de sus cuadrados de los divisores de n
--   ((a^(2*(x+1))-1)/(a^2-1)) * ((b^(2*(y+1))-1)/(b^2-1)) *...*
--
-- Definir, a partir de la nota anterior, la función
--    sumaCuadradosDivisores2 :: Int -> Integer
-- tal que (sumaCuadradosDivisores2 n) es la suma de los cuadrados de
-- los divisores de n. Por ejemplo,
--    sumaCuadradosDivisores2 6  ==  50
-- ---------------------------------------------------------------------

sumaCuadradosDivisores2 :: Integer -> Integer
sumaCuadradosDivisores2 n =
```

```
    product [(a^(2*(m+1))-1) `div` (a^2-1) | (a,m) <- factorizacion n]

-- (factorizacion n) es la factorización prima de n.
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n =
    [(head xs, genericLength xs) | xs <- group (primeFactors n)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comparar las estadísticas del cálculo de las
-- siguientes expresiones
--    sumaCuadradosDivisores1 1000000
--    sumaCuadradosDivisores2 1000000
-- El cálculo es
--    ghci> sumaCuadradosDivisores1 1000000
--    1388804117611
--    (2.91 secs, 104321520 bytes)
--    ghci> sumaCuadradosDivisores2 1000000
--    1388804117611
--    (0.01 secs, 550672 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    cerosDelFactorial :: Integer -> Integer
-- tal que (cerosDelFactorial n) es el número de ceros en que termina el
-- factorial de n. Por ejemplo,
--    cerosDelFactorial 24                          ==  4
--    cerosDelFactorial 25                          ==  6
--    length (show (cerosDelFactorial (1234^5678)))  ==  17552
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

cerosDelFactorial1 :: Integer -> Integer
cerosDelFactorial1 n = ceros (factorial n)

-- (factorial n) es el factorial n. Por ejemplo,
--    factorial 3  ==  6
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```haskell
-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
ceros :: Integer -> Integer
ceros n | rem n 10 /= 0 = 0
        | otherwise     = 1 + ceros (div n 10)

-- 2ª definición
-- =============

cerosDelFactorial2 :: Integer -> Integer
cerosDelFactorial2 n = ceros2 (factorial n)

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
ceros2 :: Integer -> Integer
ceros2 n = genericLength (takeWhile (=='0') (reverse (show n)))

-- 3ª definición
-- =============

cerosDelFactorial3 :: Integer -> Integer
cerosDelFactorial3 n | n < 5     = 0
                     | otherwise = m + cerosDelFactorial3 m
                     where m = n `div` 5

-- Comparación de la eficiencia
--    ghci> cerosDelFactorial1 (3*10^4)
--    7498
--    (3.96 secs, 1,252,876,376 bytes)
--    ghci> cerosDelFactorial2 (3*10^4)
--    7498
--    (3.07 secs, 887,706,864 bytes)
--    ghci> cerosDelFactorial3 (3*10^4)
--    7498
--    (0.03 secs, 9,198,896 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. El Triángulo de Floyd, llamado así en honor a Robert
-- Floyd, es un triángulo rectángulo formado con números naturales. Para
-- crear un triángulo de Floyd, se comienza con un 1 en la esquina
-- superior izquierda, y se continúa escribiendo la secuencia de los
-- números naturales de manera que cada línea contenga un número más que
-- la anterior:
--    1
--    2    3
--    4    5       6
--    7    8       9       10
--    11   12      13      14      15
--
-- Definir la función
--    trianguloFloyd :: [[Int]]
-- tal que trianguloFloyd es la lista formada por todas las líneas del
-- triángulo. Por ejemplo,
--    ghci> take 10 trianguloFloyd
--    [[1],
--     [2,3],
--     [4,5,6],
--     [7,8,9,10],
--     [11,12,13,14,15],
--     [16,17,18,19,20,21],
--     [22,23,24,25,26,27,28],
--     [29,30,31,32,33,34,35,36],
--     [37,38,39,40,41,42,43,44,45],
--     [46,47,48,49,50,51,52,53,54,55]]
-- ---------------------------------------------------------------------

trianguloFloyd :: [[Integer]]
trianguloFloyd = iterate siguiente [1]

siguiente :: [Integer] -> [Integer]
siguiente xs = [a..a+n]
  where a = 1+last xs
        n = genericLength xs

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

```
--   filaTrianguloFloyd :: Int -> [Int]
-- tal que (filaTrianguloFloyd n) es la n-sima fila del triángulo de
-- Floyd. Por ejemplo,
--   filaTrianguloFloyd 6 == [16,17,18,19,20,21]
-- ---------------------------------------------------------------------

filaTrianguloFloyd :: Integer -> [Integer]
filaTrianguloFloyd n = trianguloFloyd `genericIndex` (n - 1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck la siguiente propiedad: la
-- suma de los números de la línea n es n(n^2 + 1)/2 .
-- ---------------------------------------------------------------------

-- La propiedad es
prop_trianguloFloyd n =
  n > 0 ==> sum (filaTrianguloFloyd n) == (n*(n^2+1)) `div` 2

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo, los árboles
--         5              8            5          5
--        / \            / \          / \        / \
--       /   \          /   \        /   \      /   \
--      9     7        9     3      9     2    4     7
--     / \   / \      / \   / \    / \             / \
--    1   4 6   8    1   4 6   2  1   4           6   2
-- se pueden representar por
--    ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
--    ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
--    ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
--    ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
```

```haskell
--    ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))
--
-- Definir la función
--    igualEstructura :: Arbol -> Arbol -> Bool
-- tal que (igualEstructura a1 a1) se verifica si los árboles a1 y a2
-- tienen la misma estructura. Por ejemplo,
--    igualEstructura ej3arbol1 ej3arbol2 == True
--    igualEstructura ej3arbol1 ej3arbol3 == False
--    igualEstructura ej3arbol1 ej3arbol4 == False
-- ---------------------------------------------------------------------

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

ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))

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

4.3. Examen 3 (25 de enero de 2016)

El examen es común con el del grupo 1 (ver página 23).

4.4. Examen 4 (3 de marzo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (3 de marzo de 2016)
-- ---------------------------------------------------------------------

import Data.List
import Data.Numbers.Primes
```

```haskell
import Data.Array
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [2.5 puntos] Un número es libre de cuadrados si no es
-- divisible por el cuadrado de ningún entero mayor que 1. Por ejemplo,
-- 70 es libre de cuadrado porque sólo es divisible por 1, 2, 5, 7 y 70;
-- en cambio, 40 no es libre de cuadrados porque es divisible por 2².
--
-- Definir la función
--    libreDeCuadrados :: Integer -> Bool
-- tal que (libreDeCuadrados x) se verifica si x es libre de
-- cuadrados. Por ejemplo,
--    libreDeCuadrados 70                 ==  True
--    libreDeCuadrados 40                 ==  False
--    libreDeCuadrados 510510             ==  True
--    libreDeCuadrados (((10^10)^10)^10)  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición
libreDeCuadrados :: Integer -> Bool
libreDeCuadrados n = all (==1) (map length $ group $ primeFactors n)

-- 2ª definición
libreDeCuadrados2 :: Integer -> Bool
libreDeCuadrados2 n = nub ps == ps
    where ps = primeFactors n

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Un número de Lucas-Carmichael es un entero positivo n
-- compuesto, impar, libre de cuadrados y  tal que si p es un factor
-- primo de n, entonces p + 1 es un factor de n + 1.
--
-- Definir la función
--    lucasCarmichael :: [Integer]
-- que calcula la sucesión de los números de Lucas-Carmichael. Por
-- ejemplo,
--    take 8 lucasCarmichael == [399,935,2015,2915,4991,5719,7055,8855]
-- ---------------------------------------------------------------------
```

```haskell
lucasCarmichael:: [Integer]
lucasCarmichael =
    [x | x <- [2..],
         let ps = primeFactors x,
         and [mod (x+1) (p+1) == 0 | p <- ps],
         nub ps == ps,
         not (isPrime x),
         odd x]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Calcular el primer número de Lucas-Carmichael con 5
-- factores primos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [x | x <- lucasCarmichael, length (primeFactors x) == 5]
--    588455

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] 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
-- 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))

ramasCon :: Eq a => Arbol a -> a -> [[a]]
```

```haskell
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]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Representamos las 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
```

```haskell
ampliada p =
    array ((1,1),(m+1,n)) [((i,j),f i j) | i <- [1..m+1], j <- [1..n]]
    where (_,(m,n)) = bounds p
          f i j | i <= m    = p!(i,j)
                | otherwise = sum [p!(i,j) | i <- [1..m]]

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Definir la función
--    acumulaSumas :: [[Int]] -> [Int]
-- tal que (acumulaSumas xss) calcula la suma de las listas de xss de
-- forma acumulada. Por ejemplo,
--   acumulaSumas [[1,2,5],[3,9], [1,7,6,2]] == [8,20,36]
-- ---------------------------------------------------------------------

-- 1ª definición
acumulaSumas :: [[Int]] -> [Int]
acumulaSumas xss = acumula $ map sum xss

acumula :: [Int] -> [Int]
acumula [] = []
acumula (x:xs) = x:(map (+x) $ acumula xs)

-- 2ª definición
acumulaSumas2 :: [[Int]] -> [Int]
acumulaSumas2 xss = scanl1 (+) $ map sum xss

-- 3ª definición
acumulaSumas3 :: [[Int]] -> [Int]
acumulaSumas3  = scanl1 (+) . map sum

-- ---------------------------------------------------------------------
-- Ejercicio 5 (en Maxima). [1.5 puntos]  Construir un programa que
-- calcule la suma de todos los números primos menores que n. Por
-- ejemplo,
--    sumaPrimosMenores (10)     = 17
--    sumaPrimosMenores (200000) = 1709600813
-- ---------------------------------------------------------------------
```

4.5. Examen 5 (5 de mayo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (5 de mayo de 2016)
-- ---------------------------------------------------------------------

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

import Data.Array
import Data.Maybe
import I1M.Pila
import I1M.PolOperaciones
import Test.QuickCheck
import qualified Data.Matrix as M
import qualified I1M.Cola as C

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Los números felices se caracterizan por lo siguiente:
-- dado n > 0, se reemplaza el número por la suma de los cuadrados de
-- sus dígitos, y se repite el proceso hasta que el número es igual a 1
-- o hasta que se entra en un bucle que no incluye el 1. Por ejemplo,
--   49 -> 4^2 + 9^2 = 97 -> 9^2 + 7^2 = 130 -> 1^3 + 3^2 = 10 -> 1^2 = 1
--
-- Los números que al finalizar el proceso terminan con 1, son conocidos
-- como números felices. Los que no, son conocidos como números
-- infelices (o tristes).
--
-- Definir la función
--     orbita:: Int -> [Int]
-- tal que (orbita n) es la sucesión de números obtenidos a partir de
-- n. Por ejemplo:
--     take 10 (orbita 49) == [49,97,130,10,1,1,1,1,1,1]
--     take 20 (orbita 48)
--     [48,80,64,52,29,85,89,145,42,20,4,16,37,58,89,145,42,20,4,16]
-- ---------------------------------------------------------------------

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

```haskell
sig :: Int -> Int
sig = sum . (map (^2)) . digitos

orbita :: Int -> [Int]
orbita n = iterate sig n

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Se observan las siguientes propiedades de la órbita de
-- un número natural n:
-- (*) O bien aparece un 1, en cuyo caso todos los siguientes son 1 y el
--     número es feliz.
-- (*) O bien, a partir de un término se repite periódicamente con un
--     periodo de longitud 8 (4,16,37,58,89,145,42,20)
--
-- Definir la constante
--    numerosFelices:: [Int]
-- que es la sucesión de números felices. Por ejemplo,
--    take 20 numerosFelices
--    [1,7,10,13,19,23,28,31,32,44,49,68,70,79,82,86,91,94,97,100]
--
-- Nota: Para determinar la felicidad de un número n, es suficiente recorrer
-- su órbita hasta encontrar un 1 o un 4.
-- ---------------------------------------------------------------------

-- 1ª solución
felicidad :: Int -> Maybe Bool
felicidad n = aux (orbita n)
    where aux (x:xs) | x == 1    = Just True
                     | x == 4    = Just False
                     | otherwise = aux xs

esFeliz :: Int -> Bool
esFeliz = fromJust . felicidad

numerosFelices :: [Int]
numerosFelices = filter esFeliz [1..]

-- 2ª solución
esFeliz2 :: Int -> Bool
```

```haskell
esFeliz2 n = head (until p tail (orbita n)) == 1
    where p (x:xs) = x == 1 || x == 4

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir las funciones
--    colaApila:: C.Cola a -> Pila a
--    pilaAcola:: Pila a -> C.Cola a
-- tales que (colaApila c) transforma la cola c en una pila y pilaAcola
-- realiza la transformación recíproca. Por ejemplo,
--   ghci> let p = foldr apila vacia [3,8,-1,0,9]
--   ghci> p
--   3|8|-1|0|9|-
--   ghci> let c = pilaAcola p
--   ghci> c
--   C [3,8,-1,0,9]
--   ghci> colaApila c
--   3|8|-1|0|9|-
--   ghci> colaApila (pilaAcola p) == p
--   True
--   ghci> pilaAcola (colaApila c) == c
--   True
-- ---------------------------------------------------------------------

colaApila:: C.Cola a -> Pila a
colaApila c | C.esVacia c = vacia
            | otherwise   = apila x (colaApila q)
    where x = C.primero c
          q = C.resto c

pilaAcola:: Pila a -> C.Cola a
pilaAcola p = aux p C.vacia
    where aux q c | esVacia q = c
                  | otherwise = aux (desapila q) (C.inserta (cima q) c)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    polDiagonal :: M.Matrix Int -> Polinomio Int
-- tal que (polDiagonal p) es el polinomio cuyas raíces son los
-- elementos de la diagonal de la matriz cuadrada p. Por ejemplo,
--    ghci> polDiagonal (M.fromLists[[1,2,3],[4,5,6],[7,8,9]])
```

```haskell
--    x^3 + -15*x^2 + 59*x + -45
-- ---------------------------------------------------------------------

polDiagonal :: M.Matrix Int -> Polinomio Int
polDiagonal m = multListaPol (map f (diagonal m))
    where f a = consPol 1 1 (consPol 0 (-a) polCero)

-- (diagonal p) es la lista de los elementos de la diagonal de la matriz
-- p. Por ejemplo,
--    diagonal (M.fromLists[[1,2,3],[4,5,6],[7,8,9]])  ==  [1,5,9]
diagonal :: Num a => M.Matrix a -> [a]
diagonal p = [p M.! (i,i) | i <- [1..min m n]]
    where m = M.nrows p
          n = M.ncols p

-- (multListaPol ps) es el producto de los polinomios de la lista ps.
multListaPol :: [Polinomio Int] -> Polinomio Int
multListaPol []     = polUnidad
multListaPol (p:ps) = multPol p (multListaPol ps)

-- 2ª definición de multListaPol
multListaPol2 :: [Polinomio Int] -> Polinomio Int
multListaPol2 = foldr multPol polUnidad

-- ---------------------------------------------------------------------
-- Ejercicio 4. El algoritmo de Damm (http://bit.ly/1SyWhFZ) se usa en
-- la detección de errores en códigos numéricos. Es un procedimiento
-- para obtener un dígito de control, usando la siguiente matriz, como
-- describimos en los ejemplos
--
--    |  0   1   2   3   4   5   6   7   8   9
-- --+----------------------------------------
-- 0 |  0   3   1   7   5   9   8   6   4   2
-- 1 |  7   0   9   2   1   5   4   8   6   3
-- 2 |  4   2   0   6   8   7   1   3   5   9
-- 3 |  1   7   5   0   9   8   3   4   2   6
-- 4 |  6   1   2   3   0   4   5   9   7   8
-- 5 |  3   6   7   4   2   0   9   5   8   1
-- 6 |  5   8   6   9   7   2   0   1   3   4
-- 7 |  8   9   4   5   3   6   2   0   1   7
```

```
-- 8 |  9   4   3   8   6   1   7   2   0   9
-- 9 |  2   5   8   1   4   3   6   7   9   0
--
-- Ejemplo 1: cálculo del dígito de control de 572
--    + se comienza con la fila 0 y columna 5 de la matriz -> 9
--    + a continuación, la fila 9 y columna 7 de la matriz -> 7
--    + a continuación, la fila 7 y columna 2 de la matriz -> 4
-- con lo que se llega al final del proceso. Entonces, el dígito de
-- control de 572 es 4.
--
-- Ejemplo 2: cálculo del dígito de control de 57260
--    + se comienza con la fila 0 y columna 5 de la matriz -> 9
--    + a continuación, la fila 9 y columna 7 de la matriz -> 7
--    + a continuación, la fila 9 y columna 2 de la matriz -> 4
--    + a continuación, la fila 6 y columna 4 de la matriz -> 5
--    + a continuación, la fila 5 y columna 0 de la matriz -> 3
-- con lo que se llega al final del proceso. Entonces, el dígito de
-- control de 57260 es 3.
--
-- Representamos las matrices como tablas cuyos índices son pares de
-- números naturales.
--    type Matriz a = Array (Int,Int) a
--
-- Definimos la matriz:
--    mDamm :: Matriz Int
--    mDamm = listArray ((0,0),(9,9)) [0,3,1,7,5,9,8,6,4,2,
--                                     7,0,9,2,1,5,4,8,6,3,
--                                     4,2,0,6,8,7,1,3,5,9,
--                                     1,7,5,0,9,8,3,4,2,6,
--                                     6,1,2,3,0,4,5,9,7,8,
--                                     3,6,7,4,2,0,9,5,8,1,
--                                     5,8,6,9,7,2,0,1,3,4,
--                                     8,9,4,5,3,6,2,0,1,7,
--                                     9,4,3,8,6,1,7,2,0,9,
--                                     2,5,8,1,4,3,6,7,9,0]
--
-- Definir la función
--    digitoControl :: Int -> Int
-- tal que (digitoControl n) es el dígito de control de n. Por ejemplo:
--    digitoControl 572          == 4
```

```
--    digitoControl 57260        == 3
--    digitoControl 12345689012  == 6
--    digitoControl 5724         == 0
--    digitoControl 572603       == 0
--    digitoControl 123456890126 == 0
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

mDamm :: Matriz Int
mDamm = listArray ((0,0),(9,9)) [0,3,1,7,5,9,8,6,4,2,
                                 7,0,9,2,1,5,4,8,6,3,
                                 4,2,0,6,8,7,1,3,5,9,
                                 1,7,5,0,9,8,3,4,2,6,
                                 6,1,2,3,0,4,5,9,7,8,
                                 3,6,7,4,2,0,9,5,8,1,
                                 5,8,6,9,7,2,0,1,3,4,
                                 8,9,4,5,3,6,2,0,1,7,
                                 9,4,3,8,6,1,7,2,0,9,
                                 2,5,8,1,4,3,6,7,9,0]

-- 1ª solución
digitoControl :: Int -> Int
digitoControl n = aux (digitos n) 0
    where aux [] d = d
          aux (x:xs) d = aux xs (mDamm ! (d,x))

-- 2ª solución:
digitoControl2 :: Int -> Int
digitoControl2 n = last (scanl f 0 (digitos n))
    where f d x = mDamm ! (d,x)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que si añadimos al final de
-- un número n su dígito de control, el dígito de control del número que
-- resulta siempre es 0.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_DC :: Int -> Property
```

```haskell
prop_DC n = n >= 0 ==> digitoControl m == 0
    where m = read (show n ++ show (digitoControl n))

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

-- 2ª expresión de la propiedad
prop_DC2 :: Int -> Bool
prop_DC2 n = digitoControl m == 0
    where m = read (show (abs n) ++ show (digitoControl (abs n)))

-- La comprobación es
--    ghci> quickCheck prop_DC2
--    +++ OK, passed 100 tests.

-- 3ª expresión de la propiedad
prop_DC3 :: Int -> Bool
prop_DC3 n = digitoControl (10 * n1 + digitoControl n1) == 0
    where n1 = abs n

-- La comprobación es
--    ghci> quickCheck prop_DC3
--    +++ OK, passed 100 tests.

-- 4ª expresión de la propiedad
prop_DC4 :: (Positive Int) -> Bool
prop_DC4 (Positive n) =
    digitoControl2 (10 * n + digitoControl2 n) == 0

-- La comprobación es
--    ghci > quickCheck prop_DC4
--    +++ OK, passed 100 tests.
```

4.6. Examen 6 (7 de junio de 2016)

El examen es común con el del grupo 1 (ver página 51).

4.7. Examen 7 (23 de junio de 2016)

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

4.8. Examen 8 (01 de septiembre de 2016)

El examen es común con el del grupo 1 (ver página 70).

5

Exámenes del grupo 5

Andrés Cordón

5.1. Examen 1 (12 de Noviembre de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (3 de noviembre de 2015)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    selecC :: Int -> Int -> [Int] -> [Int]
-- tal que (selecC a b xs) es la lista de los elementos de que son
-- múltiplos de a pero no de b. Por ejemplo,
--   selecC 3 2 [8,9,2,3,4,5,6,10] == [9,3]
--   selecC 2 3 [8,9,2,3,4,5,6,10] == [8,2,4,10]
-- ---------------------------------------------------------------------

selecC :: Int -> Int -> [Int] -> [Int]
selecC a b xs = [x | x <- xs, rem x a == 0, rem x b /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    selecR :: Int -> Int -> [Int] -> [Int]
-- tal que (selecR a b xs) es la lista de los elementos de que son
-- múltiplos de a pero no de b. Por ejemplo,
--   selecR 3 2 [8,9,2,3,4,5,6,10] == [9,3]
```

```haskell
--   selecR 2 3 [8,9,2,3,4,5,6,10] == [8,2,4,10]
-- ---------------------------------------------------------------------

selecR :: Int -> Int -> [Int] -> [Int]
selecR _ _ [] = []
selecR a b (x:xs)
    | rem x a == 0 && rem x b /= 0 = x : selecR a b xs
    | otherwise                    = selecR a b xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    mismaLongitud :: [[a]] -> Bool
-- tal que (mismaLongitud xss) se verifica si todas las listas de la
-- lista de listas xss tienen la misma longitud. Por ejemplo,
--    mismaLongitud [[1,2],[6,4],[0,0],[7,4]] == True
--    mismaLongitud [[1,2],[6,4,5],[0,0]]     == False
-- ---------------------------------------------------------------------

-- 1ª solución:
mismaLongitud1 :: [[a]] -> Bool
mismaLongitud1 []       = True
mismaLongitud1 (xs:xss) = and [length ys == n | ys <- xss]
    where n = length xs

-- 2ª solución:
mismaLongitud2 :: [[a]] -> Bool
mismaLongitud2 xss =
    and [length xs == length ys | (xs,ys) <- zip xss (tail xss)]

-- 3ª solución:
mismaLongitud3 :: [[a]] -> Bool
mismaLongitud3 (xs:ys:xss) =
    length xs == length ys && mismaLongitud3 (ys:xss)
mismaLongitud3 _ = True

-- 4ª solución:
mismaLongitud4 :: [[a]] -> Bool
mismaLongitud4 [] = True
mismaLongitud4 (xs:xss) =
    all (\ys -> length ys == n) xss
```

```
    where n = length xs

-- Comparación de eficiencia
--    ghci> mismaLongitud1 (replicate 20000 (replicate 20000 5))
--    True
--    (5.05 secs, 0 bytes)
--    ghci> mismaLongitud2 (replicate 20000 (replicate 20000 5))
--    True
--    (9.98 secs, 0 bytes)
--    ghci> mismaLongitud3 (replicate 20000 (replicate 20000 5))
--    True
--    (10.17 secs, 0 bytes)
--    ghci> mismaLongitud4 (replicate 20000 (replicate 20000 5))
--    True
--    (5.18 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los resultados de las votaciones a delegado en un grupo
-- de clase se recogen mediante listas de asociación. Por ejemplo,
--    votos :: [(String,Int)]
--    votos = [("Ana Perez",10),("Juan Lopez",7), ("Julia Rus", 27),
--             ("Pedro Val",1), ("Pedro Ruiz",27),("Berta Gomez",11)]
--
-- Definir la función
--    ganadores :: [(String,Int)] -> [String]
-- tal que (ganadores xs) es la lista de los estudiantes con mayor
-- número de votos en xs. Por ejemplo,
--     ganadores votos == ["Julia Rus","Pedro Ruiz"]
-- ---------------------------------------------------------------------

votos :: [(String,Int)]
votos = [("Ana Perez",10),("Juan Lopez",7), ("Julia Rus", 27),
         ("Pedro Val",1), ("Pedro Ruiz",27),("Berta Gomez",11)]

ganadores :: [(String,Int)] -> [String]
ganadores xs = [c | (c,x) <- xs, x == maxVotos]
    where maxVotos = maximum [j | (i,j) <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Un entero positivo x se dirá muy par si tanto x como
```

```haskell
-- x^2 sólo contienen cifras pares. Por ejemplo, 200 es muy par porque
-- todas las cifras de 200 y 200² = 40000 son pares; pero 26 no lo es
-- porque 26² = 767 tiene cifras impares.
--
-- Definir  la función
--    muyPar :: Integer -> Bool
-- tal que (muyPar x) se verifica si x es muy par. por ejemplo,
--    muyPar 200           == True
--    muyPar 26            == False
--    muyPar 828628040080  == True
-- ---------------------------------------------------------------------

muyPar :: Integer -> Bool
muyPar n = all even (digitos n) && all even (digitos (n*n))

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    siguienteMuyPar :: Integer -> Integer
-- tal que (siguienteMuyPar x) es el primer número mayor que x que es
-- muy par. Por ejemplo,
--    siguienteMuyPar 300           ==  668
--    siguienteMuyPar 668           ==  680
--    siguienteMuyPar 828268400000  ==  828268460602
-- ---------------------------------------------------------------------

siguienteMuyPar :: Integer -> Integer
siguienteMuyPar x =
    head [n | n <- [y,y+2..], muyPar n]
    where y = siguientePar x

-- (siguientePar x) es el primer número mayor que x que es par. Por
-- ejemplo,
--    siguientePar 3  ==  4
--    siguientePar 4  ==  6
siguientePar :: Integer -> Integer
siguientePar x | odd x     = x+1
               | otherwise = x+2
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir la función
--    transformada :: [a] -> [a]
-- tal que (transformada xs) es la lista obtenida repitiendo cada
-- elemento tantas veces como indica su posición en la lista. Por
-- ejemplo,
--    transformada [7,2,5] == [7,2,2,5,5,5]
--    transformada "eco"   == "eccooo"
-- ---------------------------------------------------------------------

transformada :: [a] -> [a]
transformada xs = concat [replicate n x | (n,x) <- zip [1..] xs]

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck si la transformada de una
-- lista de n números enteros, con n >= 2, tiene menos de n^3 elementos.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_transformada :: [Int] -> Property
prop_transformada xs = n >= 2 ==> length (transformada xs) < n^3
    where n = length xs

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

5.2. Examen 2 (17 de Diciembre de 2015)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (7 de diciembre de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
--    sumaDivC :: Int -> [Int] -> Int
-- tal que (SumaDivC x xs) es la suma de los cuadrados de los
-- elementos de xs que son divisibles por x. Por ejemplo,
--    sumaDivC 3 [1..7] == 45
```

```haskell
--    sumaDivC 2 [1..7] == 56
-- ---------------------------------------------------------------------

sumaDivC :: Int -> [Int] -> Int
sumaDivC x xs = sum [y^2 | y <- xs, rem y x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, usando funciones de orden superior, la
-- función
--    sumaDivS :: Int -> [Int] -> Int
-- tal que (SumaDivS x xs) es la suma de los cuadrados de los
-- elementos de xs que son divisibles por x. Por ejemplo,
--    sumaDivS 3 [1..7] == 45
--    sumaDivS 2 [1..7] == 56
-- ---------------------------------------------------------------------

sumaDivS :: Int -> [Int] -> Int
sumaDivS x =   sum . map (^2) . filter (\ y -> rem y x == 0)

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por recursión, la función
--    sumaDivR :: Int -> [Int] -> Int
-- tal que (SumaDivR x xs) es la suma de los cuadrados de los
-- elementos de xs que son divisibles por x. Por ejemplo,
--    sumaDivR 3 [1..7] == 45
--    sumaDivR 2 [1..7] == 56
-- ---------------------------------------------------------------------

sumaDivR :: Int -> [Int] -> Int
sumaDivR _ [] = 0
sumaDivR x (y:xs) | rem y x == 0 = y^2+sumaDivR x xs
                  | otherwise = sumaDivR x xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir, por plegado, la función
--    sumaDivP :: Int -> [Int] -> Int
-- tal que (SumaDivP x xs) es la suma de los cuadrados de los
-- elementos de xs que son divisibles por x. Por ejemplo,
--    sumaDivP 3 [1..7] == 45
--    sumaDivP 2 [1..7] == 56
```

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

sumaDivP :: Int -> [Int] -> Int
sumaDivP x = foldr (\y z -> if rem y x == 0 then y^2+z else z)  0

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una lista de enteros positivos se dirá encadenada si el
-- último dígito de cada elemento coincide con el primer dígito del
-- siguiente; y el último dígito del último número coincide con el
-- primer dígito del primer número.
--
-- Definir la función
--    encadenada :: [Int] -> Bool
-- tal que (encadenada xs) se verifica si xs está encadenada. Por
-- ejemplo,
--    encadenada [92,205,77,72]  == False
--    encadenada [153,32,207,71] == True
--    encadenada [153,32,207,75] == False
--    encadenada [123]           == False
--    encadenada [121]           == True
-- ---------------------------------------------------------------------

encadenada :: [Int] -> Bool
encadenada (x:xs) =
    and [last (show a) == head (show b) | (a,b) <- zip (x:xs) (xs++[x])]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Un entero positivo se dirá especial si la suma de sus
-- dígitos coincide con el número de sus divisores. Por ejemplo, 2015 es
-- especial, puesto que la suma de sus dígitos es 2+0+1+5=8 y tiene 8
-- divisores (1,5,13,31,65,155,403 y 2015).
--
-- Definir la sucesión infinita
--    especiales :: [Int]
-- formada por todos los números especiales. Por ejemplo:
--    take 12 especiales == [1,2,11,22,36,84,101,152,156,170,202,208]
-- ---------------------------------------------------------------------

especiales :: [Int]
especiales = [x | x <- [1..], sum (digitos x) == length (divisores x)]
```

```haskell
divisores :: Int -> [Int]
divisores x =[y | y <- [1..x], rem x y == 0]

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    siguienteEspecial :: Int -> Int
-- tal que (siguienteEspecial x) es el primer entero mayor que x que es
-- especial. Por ejemplo,
--    siguienteEspecial 2015 == 2101
-- ---------------------------------------------------------------------

siguienteEspecial :: Int -> Int
siguienteEspecial x =  head (dropWhile (<=x) especiales)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    palabras :: String -> [String]
-- tal que (palabras xs) es la lista de las palabras que aparecen en xs,
-- eliminando los espacios en blanco. Por ejemplo,
--    ghci> palabras " el lagarto   esta    llorando "
--    ["el","lagarto","esta","llorando"]
-- ---------------------------------------------------------------------

palabras :: String -> [String]
palabras [] = []
palabras (x:xs)
   | x ==' '   = palabras (dropWhile (==' ') xs)
   | otherwise = (x:takeWhile (/=' ') xs):palabras (dropWhile (/=' ') xs)

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Los árboles binarios se representan mediante el tipo de
-- datos
--    data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
--
-- Definir la función
--    maxHojas :: Ord a => Arbol a -> a
```

```haskell
-- tal que (maxHojas t) es el mayor valor que aparece en una hoja del
-- árbol t. Por ejemplo,
--    ghci> maxHojas (N 14 (N 2 (H 7) (H 10)) (H 3))
--    10
-- ---------------------------------------------------------------------

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

maxHojas :: Ord a => Arbol a -> a
maxHojas (H x)     = x
maxHojas (N _ i d) = max (maxHojas i) (maxHojas d)

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    sumaValor :: Num a => a -> Arbol a -> Arbol a
-- tal que (sumaValor v t) es el árbol obtenido a partir de t al sumar v
-- a todos sus nodos. Por ejemplo,
--    ghci> sumaValor 7 (N 14 (N 2 (H 7) (H 10)) (H 3))
--    N 21 (N 9 (H 14) (H 17)) (H 10)
-- ---------------------------------------------------------------------

sumaValor :: Num a => a -> Arbol a -> Arbol a
sumaValor v (H x)     = H (v+x)
sumaValor v (N x i d) = N (v+x) (sumaValor v i) (sumaValor v d)
```

5.3. Examen 3 (25 de enero de 2016)

El examen es común con el del grupo 2 (ver página 83).

5.4. Examen 4 (10 de marzo de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 4º examen de evaluación continua (10 de marzo de 2016)
-- ---------------------------------------------------------------------

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

```haskell
import Data.List
import Data.Numbers.Primes
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la lista infinita
--    paresRel :: [(Int,Int)]
-- tal que paresRel enumera todos los pares de enteros positivos (a,b),
-- con 1 <= a < b, tales que a y b tienes los mismos divisores primos.
-- Por ejemplo,
--    ghci> take 10 paresRel
--    [(2,4),(2,8),(4,8),(3,9),(6,12),(2,16),(4,16),(8,16),(6,18),(12,18)]
-- ---------------------------------------------------------------------

paresRel :: [(Int,Int)]
paresRel = [(a,b) | b <- [1..], a <- [1..b-1], p a b]
    where p a b = nub (primeFactors a) == nub (primeFactors b)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. ¿Qué lugar ocupa el par (750,1080) en la lista
-- infinita paresRel?
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> 1 + length (takeWhile (/=(750,1080)) paresRel)
--    1492

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    segmento :: Eq a => [a] -> [a]
-- tal que (segmento xs) es el mayor segmento inicial de xs que no
-- contiene ningún elemento repetido. Por ejemplo:
--    segmento [1,2,3,2,4,5] == [1,2,3]
--    segmento "caceres"     == "ca"
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

segmento1 :: Eq a => [a] -> [a]
```

```haskell
segmento1 = last . filter (\ys -> nub ys == ys) . inits

-- 2ª solución
-- ===========

segmento2 :: Eq a => [a] -> [a]
segmento2 xs = aux xs []
    where aux [] ys = reverse ys
          aux (x:xs) ys | x `elem` ys = reverse ys
                        | otherwise   = aux xs (x:ys)

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

--    ghci> last (segmento1 [1..10^3])
--    1000
--    (6.19 secs, 59,174,640 bytes)
--    ghci> last (segmento2 [1..10^3])
--    1000
--    (0.04 secs, 0 bytes)

-- Solución
-- ========

-- En lo que sigue usaremos la 2ª definición
segmento :: Eq a => [a] -> [a]
segmento = segmento2

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Se observa que 10! = 3628800 comienza con 4 dígitos
-- distintos (después se repite el dígito 8).
--
-- Calcular el menor número natural cuyo factorial comienza con 9
-- dígitos distintos.
-- ---------------------------------------------------------------------

factorial :: Integer -> Integer
factorial x = product [1..x]

-- El cálculo es
```

```haskell
--    ghci> head [x | x <- [0..], length (segmento (show (factorial x))) == 9]
--    314

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Las expresiones aritméticas se pueden definir mediante
-- el siguiente tipo de dato
--    data Expr  = N Int | V Char | Sum Expr Expr | Mul Expr Expr
--               deriving Show
-- Por ejemplo, (x+3)+(7*y) se representa por
--    ejExpr :: Expr
--    ejExpr = Sum (Sum (V 'x') (N 3))(Mul (N 7) (V 'y'))
--
-- Definir el predicado
--    numerico :: Expr -> Bool
-- tal que (numerico e) se verifica si la expresión e no contiene ninguna
-- variable. Por ejemplo,
--   numerico ejExpr            == False
--   numerico (Sum (N 7) (N 9)) == True
-- ---------------------------------------------------------------------

data Expr  = N Int | V Char | Sum Expr Expr | Mul Expr Expr
           deriving Show

ejExpr :: Expr
ejExpr = Sum (Sum (V 'x') (N 3))(Mul (N 7) (V 'y'))

numerico :: Expr -> Bool
numerico (N _)       = True
numerico (V _)       = False
numerico (Sum e1 e2) = numerico e1 && numerico e2
numerico (Mul e1 e2) = numerico e1 && numerico e2

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    evalua :: Expr -> Maybe Int
-- tal que (evalua e) devuelve 'Just v' si la expresión e es numérica y
-- v es su valor, o bien 'Nothing' si e no es numérica. Por ejemplo:
--    evalua ejExpr            == Nothing
--    evalua (Sum (N 7) (N 9)) == Just 16
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución
evalua1 :: Expr -> Maybe Int
evalua1 e | null (aux e) = Nothing
          | otherwise    = Just (head (aux e))
    where aux (N x)       = [x]
          aux (V _)       = []
          aux (Sum e1 e2) = [x+y| x <- aux e1, y <- aux e2]
          aux (Mul e1 e2) = [x*y| x <- aux e1, y <- aux e2]

-- 2ª solución
evalua2 :: Expr -> Maybe Int
evalua2 e | numerico e = Just (valor e)
          | otherwise  = Nothing
    where valor (N x)       = x
          valor (Sum e1 e2) = valor e1 + valor e2
          valor (Mul e1 e2) = valor e1 * valor e2

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Los vectores y matrices se definen usando tablas como
-- sigue:
--    type Vector a = Array Int a
--    type Matriz a = Array (Int,Int) a
--
-- Un elemento de un vector es un máximo local si no tiene ningún
-- elemento adyacente mayor o igual que él.
--
-- Definir la función
--    posMaxVec :: Ord a => Vector a -> [Int]
-- tal que (posMaxVec p) devuelve las posiciones del vector p en las que
-- p tiene un máximo local. Por ejemplo:
--    posMaxVec (listArray (1,6) [3,2,6,7,5,3]) == [1,4]
-- ---------------------------------------------------------------------

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

-- 1ª definición
posMaxVec :: Ord a => Vector a -> [Int]
posMaxVec p =
```

```
    (if p!1 > p!2 then [1] else []) ++
    [i | i <- [2..n-1], p!(i-1) < p!i && p!(i+1) < p!i] ++
    (if p!(n-1) < p!n then [n] else [])
    where (_,n) = bounds p

-- 2ª definición
posMaxVec2 :: Ord a => Vector a -> [Int]
posMaxVec2 p =
    [1 | p ! 1 > p ! 2] ++
    [i | i <- [2..n-1], p!(i-1) < p!i && p!(i+1) < p!i] ++
    [n | p ! (n - 1) < p ! n]
    where (_,n) = bounds p

-- 3ª definición
posMaxVec3 :: Ord a => Vector a -> [Int]
posMaxVec3 p =
    [i | i <- [1..n],
         all (<p!i) [p!j | j <- vecinos i]]
    where (_,n) = bounds p
          vecinos 1 = [2]
          vecinos j | j == n    = [n-1]
                    | otherwise = [j-1,j+1]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2.  Un elemento de una matriz es un máximo local si no
-- tiene ningún vecino mayor o igual que él.
--
-- Definir la función
--    posMaxMat :: Ord a => Matriz a -> [(Int,Int)]
-- tal que (posMaxMat p) es la lista de las posiciones de la matriz p en
-- las que p tiene un máximo local. Por ejemplo,
--    ghci> posMaxMat (listArray ((1,1),(3,3)) [1,20,15,4,5,6,9,8,7])
--    [(1,2),(3,1)]
-- ---------------------------------------------------------------------

posMaxMat :: Ord a => Matriz a -> [(Int,Int)]
posMaxMat p = [(x,y) | (x,y) <- indices p, all (<(p!(x,y))) (vs x y)]
   where (_,(n,m)) = bounds p
         vs x y = [p!(x+i,y+j) | i <- [-1..1], j <- [-1..1],
                                 (i,j) /= (0,0),
```

```
                                  (x+i) `elem` [1..n],
                                  (y+j) `elem` [1..m]]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Escribir una función Haskell
--    fun :: (Int -> Int) -> IO ()
-- que actúe como la siguente función programada en Maxima:
--    fun(f) := block([a,b], a:0, b:0,
--                    x:read("Escribe un natural" ),
--                    while b <= x do (a:a+f(b),b:b+1),
--                    print(a))$
-- ---------------------------------------------------------------------

fun :: (Int -> Int) -> IO ()
fun f = do putStr "Escribe un natural: "
           xs <- getLine
           let a = sum [f i | i <- [0..read xs]]
           print a

-- Por ejemplo,
--    ghci> fun (^2)
--    Escribe un natural: 5
--    55
```

5.5. Examen 5 (5 de mayo de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 5º examen de evaluación continua (5 de mayo de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Numbers.Primes
import Data.Matrix
import I1M.Pila
import I1M.Pol
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Un entero positivo se dirá muy primo si tanto él como
-- todos sus segmentos iniciales son números primos. Por ejemplo, 7193
-- es muy primo pues 7,71,719 y 7193 son todos primos.
--
-- Definir la lista
--    muyPrimos :: [Integer]
-- de todos los números muy primos. Por ejemplo,
--    ghci> take 20 muyPrimos
--    [2,3,5,7,23,29,31,37,53,59,71,73,79,233,239,293,311,313,317,373]
-- ---------------------------------------------------------------------

muyPrimos :: [Integer]
muyPrimos = filter muyPrimo [1..]
  where muyPrimo = all isPrime . takeWhile (/=0) . iterate (`div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un bosque es una lista de árboles binarios.
--
-- Representaremos los árboles y los bosques mediante los siguientes
-- tipo de dato
--    data Arbol  a = H a | N a (Arbol a) (Arbol a) deriving Show
--    data Bosque a = B [Arbol a] deriving Show

-- Define la función
--   cuentaBosque :: Eq a => a - > Bosque a -> Int
-- tal que (cuentaBosque x b) es el número de veces que aparece el
-- elemento  x en el bosque b. Por ejemplo:
--    ghci> let t1 = N 7 (N 9 (H 1) (N 5 (H 7) (H 6))) (H 5)
--    ghci> let t2 = N 8 (H 7) (H 7)
--    ghci> let t3 = N 0 (H 4) (N 6 (H 8) (H 9))
--    ghci> cuentaBosque 7 (B [t1,t2,t3])
--    4
--    ghci> cuentaBosque 4 (B [t2,t2,t3,t3])
--    2
-- ---------------------------------------------------------------------

data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
data Bosque a = B [Arbol a] deriving Show
```

```haskell
cuentaBosque :: Eq a => a -> Bosque a -> Int
cuentaBosque x (B ts) = sum (map (cuentaArbol x) ts)

cuentaArbol :: Eq a => a -> Arbol a -> Int
cuentaArbol x (H z)
  | x == z    = 1
  | otherwise = 0
cuentaArbol x (N z i d)
  | x == z    = 1 + cuentaArbol x i + cuentaArbol x d
  | otherwise = cuentaArbol x i + cuentaArbol x d

-- ---------------------------------------------------------------------
-- Ejercicio 3. Representamos las pilas mediante el TAD de las pilas
-- (I1M.Pila).
--
-- Definir la función
--    alternaPila :: (a -> b) -> (a -> b) -> Pila a -> Pila b
-- tal que (alternaPila f g p) devuelve la pila obtenida al aplicar las
-- funciones f y g, alternativamente y empezando por f, a los elementos
-- de la pila p. Por ejemplo:
--   ghci> alternaPila (*2) (+1) (foldr apila vacia [1,2,3,4,5,6])
--   2|3|6|5|10|7|-
-- ---------------------------------------------------------------------

alternaPila :: (a -> b) -> (a -> b) -> Pila a -> Pila b
alternaPila f g p
  | esVacia p = vacia
  | otherwise = apila (f (cima p)) (alternaPila g f (desapila p))

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Representaremos las matrices mediante la librería de
-- Haskell Data.Matrix.
--
-- Definir la función
--     acumula :: Num a => Matrix a -> Matrix a
-- tal que (acumula p) devuelve la matriz obtenida al sustituir cada
-- elemento x de la matriz p por la suma del resto de elementos que
-- aparecen en la fila y en la columna de x. Por ejemplo,
--            ( 1  2  2  2 )        (  8  4 10 12 )
--    acumula ( 1  0  1  0 )  ==>   (  3  3  7 11 )
```

```
--            ( 1 -1  4  7 )        ( 12 14 10  6 )
-- En Haskell,
--    ghci> acumula (fromLists [[1,2,2,2],[1,0,1,0],[1,-1,4,7]])
--    (  8  4 10 12 )
--    (  3  3  7 11 )
--    ( 12 14 10  6 )
-- ---------------------------------------------------------------------

acumula :: Num a => Matrix a -> Matrix a
acumula p = matrix n m f where
    n = nrows p
    m = ncols p
    f (i,j) = sum [p!(k,j) | k <- [1..n], k /= i] +
              sum [p!(i,k) | k <- [1..m], k /= j]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    filasEnComun :: Eq a => Matrix a -> Matrix a -> Matrix a
-- tal que (filasEnComun p q) devuelve la matriz formada por las filas
-- comunes a p y q, ordenadas según aparecen en la matriz p (suponemos
-- que ambas matrices tienen el mismo número de columnas y que tienen
-- alguna fila en común). Por ejemplo,
--
--                 ( 1 2 1  2 )    ( 9 3 7 -1 )      ( 1 2 1  2 )
--   filasEnComun  ( 0 1 1  0 )    ( 5 7 6  0 ) ==>  ( 9 3 7 -1 )
--                 ( 9 3 7 -1 )    ( 1 2 1  2 )
--                 ( 5 7 0  0 )
--
-- En Haskell,
--    ghci> let m1 = fromLists [[1,2,1,2],[0,1,1,0],[9,3,7,-1],[5,7,0,0]]
--    ghci> let m2 = fromLists [[9,3,7,-1],[5,7,6,0],[1,2,1,2]]
--    ghci> filasEnComun m1 m2
--    (  1  2  1  2 )
--    (  9  3  7 -1 )
-- ---------------------------------------------------------------------


filasEnComun :: Eq a => Matrix a -> Matrix a ->Matrix a
filasEnComun p q =
  fromLists (toLists q `intersect` toLists p)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Representamos los polinomios mediante el TAD de los
-- Polinomios (I1M.Pol).
--
-- Un polinomio se dirá completo si los exponentes de su variable van
-- sucesivamente desde su grado hasta cero, sin faltar ninguno. Por
-- ejemplo, 5*x^3+x^2-7*x+1 es completo.
--
-- Definir la función
--    completo :: (Num a,Eq a) => Polinomio a -> Bool
-- tal que (completo p) se verifica si p es completo. Por ejemplo,
--    completo (consPol 2 5 (consPol 1 7 polCero))  ==  True
--    completo (consPol 3 5 (consPol 1 7 polCero))  ==  False
-- ---------------------------------------------------------------------

completo :: (Num a,Eq a) => Polinomio a -> Bool
completo p
  | esPolCero p  = True
  | grado p == 0 = True
  | otherwise    = grado rp == grado p - 1 && completo rp
  where rp = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    interPol :: (Num a,Eq a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (interPol p q) es el polinomio formado por términos comunes a
-- los polinomios p y q. Por ejemplo, si p1 es 2*x^6-x^4-8*x^2+9 y p2
-- es 2*x^6+x^4+9, entonces (interPol p1 p2) es 2*x^6+9. En Haskell,
--    ghci> let p1 = consPol 6 2 (consPol 4 1 (consPol 2 8 (consPol 0 9 polCero))
--    ghci> let p2 = consPol 6 2 (consPol 4 1 (consPol 0 9 polCero))
--    ghci> interPol p1 p2
--    2*x^6 + 9
-- ---------------------------------------------------------------------

interPol :: (Num a,Eq a) => Polinomio a -> Polinomio a -> Polinomio a
interPol p q
  | esPolCero p = polCero
  | esPolCero q = polCero
  | gp > gq     = interPol rp q
```

```
  | gp < gq     = interPol p rq
  | cp == cq    = consPol gp cp (interPol rp rq)
  | otherwise   = interPol rp rq
  where gp = grado p
        gq = grado q
        cp = coefLider p
        cq = coefLider q
        rp = restoPol p
        rq = restoPol q
```

5.6. Examen 6 (7 de junio de 2016)

El examen es común con el del grupo 1 (ver página 51).

5.7. Examen 7 (23 de junio de 2016)

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

5.8. Examen 8 (01 de septiembre de 2016)

El examen es común con el del grupo 1 (ver página 70).

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.