Exámenes de "Programación funcional con Haskell"

Vol. 5 (Curso 2013–14)

José A. Alonso Jiménez

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

Sevilla, 20 de noviembre de 2014

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 2013–14) 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 2013–14) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2013–14) [2]
- Piensa en Haskell (Ejercicios de programación funcional con Haskell) [3]

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

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

Los códigos del libro están disponibles en GitHub [4]

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

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

[1] https://www.cs.us.es/~jalonso/cursos/i1m-13/temas/2013-14-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-13/ejercicios/ejercicios-I1M-2013.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol4
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

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

José A. Alonso
Sevilla, 20 de noviembre de 2014

[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

1

Exámenes del grupo 1

José A. Alonso y Luis Valencia

1.1. Examen 1 (5 de Noviembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 1º examen de evaluación continua (5 de noviembre de 2012)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Definir la función
--    divisoresPrimos :: Integer -> [Integer]
-- tal que (divisoresPrimos x) es la lista de los divisores primos de x.
-- Por ejemplo,
--    divisoresPrimos 40  ==  [2,5]
--    divisoresPrimos 70  ==  [2,5,7]
-- ---------------------------------------------------------------------

divisoresPrimos :: Integer -> [Integer]
divisoresPrimos x = [n | n <- divisores x, primo n]

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

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
```

```haskell
primo :: Integer -> Bool
primo n = divisores n == [1, n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] La multiplicidad de x en y es la mayor
-- potencia de x que divide a y. Por ejemplo, la multiplicidad de 2 en
-- 40 es 3 porque 40 es divisible por 2^3 y no lo es por 2^4. Además, la
-- multiplicidad de 1 en cualquier número se supone igual a 1.
--
-- Definir la función
--    multiplicidad :: Integer -> Integer -> Integer
-- tal que (multiplicidad x y) es la
-- multiplicidad de x en y. Por ejemplo,
--    multiplicidad 2 40  ==  3
--    multiplicidad 5 40  ==  1
--    multiplicidad 3 40  ==  0
--    multiplicidad 1 40  ==  1
-- ---------------------------------------------------------------------

multiplicidad :: Integer -> Integer -> Integer
multiplicidad 1 _ = 1
multiplicidad x y =
    head [n | n <- [0..], y `rem` (x^n) == 0, y `rem` (x^(n+1)) /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Un número es libre de cuadrados si no es
-- divisible el cuadrado de ningún entero mayor que 1. Por ejemplo, 70
-- es libre de cuadrado porque sólo es divisible por 1, 2, 5, 7 y 70; en
-- cambio, 40 no es libre de cuadrados porque es divisible por 2^2.
--
-- Definir la función
--    libreDeCuadrados :: Integer -> Bool
-- tal que (libreDeCuadrados x) se verifica si x es libre de cuadrados.
-- Por ejemplo,
--    libreDeCuadrados 70  ==  True
--    libreDeCuadrados 40  ==  False
-- Calcular los 10 primeros números libres de cuadrado de 3 cifras.
-- ---------------------------------------------------------------------

-- 1ª definición:
```

```haskell
libreDeCuadrados :: Integer -> Bool
libreDeCuadrados x = x == product (divisoresPrimos x)

-- NOTA: La función primo está definida en el ejercicio 1.

-- 2ª definición
libreDeCuadrados2 :: Integer -> Bool
libreDeCuadrados2 x =
    and [multiplicidad n x == 1 | n <- divisores x]

-- 3ª definición
libreDeCuadrados3 :: Integer -> Bool
libreDeCuadrados3 n =
    null [x | x <- [2..n], rem n (x^2) == 0]


-- El cálculo es
--    ghci> take 10 [n | n <- [100..], libreDeCuadrados n]
--    [101,102,103,105,106,107,109,110,111,113]

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] La distancia entre dos números es el valor
-- absoluto de su diferencia. Por ejemplo, la distancia entre 2 y 5 es
-- 3.
--
-- Definir la función
--    cercanos :: [Int] -> [Int] -> [(Int,Int)]
-- tal que (cercanos xs ys) es la lista de pares de elementos de xs e ys
-- cuya distancia es mínima. Por ejemplo,
--    cercanos [3,7,2,1] [5,11,9]  ==  [(3,5),(7,5),(7,9)]
-- ---------------------------------------------------------------------

cercanos :: [Int] -> [Int] -> [(Int,Int)]
cercanos xs ys =
    [(x,y) | (x,y) <- pares, abs (x-y) == m]
    where pares = [(x,y) | x <- xs, y <- ys]
          m = minimum [abs (x-y) | (x,y) <- pares]
```

1.2. Examen 2 (17 de Diciembre de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (17 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List (sort)

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Definir la función
--    expandida :: [Int] -> [Int]
-- tal que (expandida xs) es la lista obtenida duplicando cada uno de
-- los elementos pares de xs. Por ejemplo,
--    expandida [3,5,4,6,6,1,0]  ==  [3,5,4,4,6,6,6,6,1,0,0]
--    expandida [3,5,4,6,8,1,0]  ==  [3,5,4,4,6,6,8,8,1,0,0]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Comprobar con QuickCheck que el número de
-- elementos de (expandida xs) es el del número de elementos de xs más
-- el número de elementos pares de xs.
-- ---------------------------------------------------------------------

prop_expandida :: [Int] -> Bool
prop_expandida xs =
    length (expandida xs) == length xs + length (filter even xs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Definir la función
--    digitosOrdenados :: Integer -> Integer
```

```haskell
-- tal que (digitosOrdenados n) es el número obtenido ordenando los
-- dígitos de n de mayor a menor. Por ejemplo,
--    digitosOrdenados 325724237  ==  775433222
-- ---------------------------------------------------------------------

digitosOrdenados :: Integer -> Integer
digitosOrdenados n = read (ordenados (show n))

ordenados :: Ord a => [a] -> [a]
ordenados [] = []
ordenados (x:xs) =
    ordenados mayores ++ [x] ++ ordenados menores
    where mayores = [y | y <- xs, y > x]
          menores = [y | y <- xs, y <= x]

-- Nota: La función digitosOrdenados puede definirse por composición
digitosOrdenados2 :: Integer -> Integer
digitosOrdenados2 = read . ordenados . show

-- Nota: La función digitosOrdenados puede definirse por composición y
-- también usando sort en lugar de ordenados
digitosOrdenados3 :: Integer -> Integer
digitosOrdenados3 = read . reverse . sort . show

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] Sea f la siguiente función, aplicable a
-- cualquier número entero positivo:
-- * Si el número es par, se divide entre 2.
-- * Si el número es impar, se multiplica por 3 y se suma 1.
--
-- La carrera de Collatz consiste en, dada una lista de números ns,
-- sustituir cada número n de ns por f(n) hasta que alguno sea igual a
-- 1. Por ejemplo, la siguiente sucesión es una carrera de Collatz
--    [ 3, 6,20, 49, 73]
--    [10, 3,10,148,220]
--    [ 5,10, 5, 74,110]
--    [16, 5,16, 37, 55]
--    [ 8,16, 8,112,166]
--    [ 4, 8, 4, 56, 83]
--    [ 2, 4, 2, 28,250]
```

```
--    [ 1, 2, 1, 14,125]
-- En esta carrera, los ganadores son 3 y 20.
--
-- Definir la función
--    ganadores :: [Int] -> [Int]
--    ganadores [3,6,20,49,73]  ==  [3,20]
-- ---------------------------------------------------------------------

ganadores :: [Int] -> [Int]
ganadores xs = selecciona xs (final xs)

-- (final xs) es el estado final de la carrera de Collatz a partir de
-- xs. Por ejemplo,
--    final [3,6,20,49,73]  ==  [1,2,1,14,125]
final :: [Int] -> [Int]
final xs | elem 1 xs = xs
         | otherwise = final [siguiente x | x <- xs]

-- (siguiente x) es el siguiente de x en la carrera de Collatz. Por
-- ejemplo,
--    siguiente 3  ==  10
--    siguiente 6  ==  3
siguiente :: Int -> Int
siguiente x | even x    = x `div` 2
            | otherwise = 3*x+1

-- (selecciona xs ys) es la lista de los elementos de xs cuyos tales que
-- los elementos de ys en la misma posición son iguales a 1. Por ejemplo,
--    selecciona [3,6,20,49,73] [1,2,1,14,125]  ==  [3,20]
selecciona :: [Int] -> [Int] -> [Int]
selecciona xs ys =
    [x | (x,y) <- zip xs ys, y == 1]
```

1.3. Examen 3 (23 de Enero de 2014)

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

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

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

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

-- (ultimoNoNulo n) es el último dígito no nulo de n. Por ejemplo,
--    ultimoNoNulo 5040  ==  4
ultimoNoNulo :: Integer -> Integer
ultimoNoNulo n | m /= 0    = m
               | otherwise = ultimoNoNulo (n `div` 10)
               where m = n `rem` 10

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

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

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

```haskell
-- (es decir, es la lista xs tal que la comprimida de xs es ps). Por
-- ejemplo,
--    expandidaC [(2,1),(3,7),(2,5),(4,7)]  ==  [1,1,7,7,7,5,5,7,7,7,7]
-- ---------------------------------------------------------------------

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

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

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

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

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

-- 2ª definición (por comprensión)
angelini2 :: Integer -> Bool
```

```haskell
angelini2 n = not (null [x | x <- show n, x `elem` show (2*n)])

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

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

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

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

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

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

-- (esSumaDePrimos n) se verifica si n es una suma de primos menores que
```

```haskell
-- n. Por ejemplo,
--    esSumaDePrimos 37  ==  True
--    esSumaDePrimos 11  ==  False
esSumaDePrimos :: Integer -> Bool
esSumaDePrimos n = esSuma n [x | x <- 2:[3,5..n-1], primo x]

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

-- (esSuma n xs) s verifica si n es suma de elementos de xs. Por ejemplo,
--    esSuma 20 [4,2,7,9]  ==  True
--    esSuma 21 [4,2,7,9]  ==  False
esSuma :: Integer -> [Integer] -> Bool
esSuma 0 _                = True
esSuma n []               = False
esSuma n (x:xs) | n == x  = True
                | n > x   = esSuma n xs || esSuma (n-x) xs
                | otherwise = esSuma n xs

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

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

1.4. Examen 4 (21 de Marzo de 2014)

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

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

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

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

```haskell
--    siguienteHamming 15  ==  16
-- ---------------------------------------------------------------------

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

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

-- (mezcla3 xs ys zs) es la lista obtenida mezclando las listas
-- ordenadas xs, ys y zs y eliminando los elementos duplicados. Por
-- ejemplo,
--    ghci> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
--    [2,3,4,5,6,8,9,10,12]
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Las operaciones de suma, resta y
-- multiplicación se pueden representar mediante el siguiente tipo de
-- datos
--    data Op = S | R | M
-- La expresiones aritméticas con dichas operaciones se pueden
-- representar mediante el siguiente tipo de dato algebraico
--    data Expr = N Int | A Op Expr Expr
```

```haskell
-- Por ejemplo, la expresión
--    (7-3)+(2*5)
-- se representa por
--    A S (A R (N 7) (N 3)) (A M (N 2) (N 5))
--
-- Definir la función
--    valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--    valor (A S (A R (N 7) (N 3)) (A M (N 2) (N 5)))  ==  14
--    valor (A M (A R (N 7) (N 3)) (A S (N 2) (N 5)))  ==  28
-- ---------------------------------------------------------------------

data Op = S | R | M

data Expr = N Int | A Op Expr Expr

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

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

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

```haskell
-- tal que (transformaPol p) es el polinomio obtenido transformado el
-- polinomio p de la primera representación a la segunda. Por ejemplo,
--    transformaPol (C 2 5 (C 4 3 (C 2 1 O)))  ==  [5,5,3,3,3,3,1,1]
--    transformaPol (C 2 100 (C 3 1 O))        ==  [100,100,1,1,1]
-- ---------------------------------------------------------------------

data Polinomio = O | C Int Int Polinomio

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

1.5. Examen 5 (16 de Mayo de 2014)

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

import Data.Array

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

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

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

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

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


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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2 puntos] Definir la función
--    listaRectangular :: Int -> Int -> a -> [a] -> [[a]]
-- tal que (listaRectangular m n x xs) es una lista de m listas de
-- longitud n formadas con los elementos de xs completada con x, si no
-- xs no tiene suficientes elementos. Por ejemplo,
--    listaRectangular 2 4 7 [0,3,5,2,4]  ==  [[0,3,5,2],[4,7,7,7]]
--    listaRectangular 4 2 7 [0,3,5,2,4]  ==  [[0,3],[5,2],[4,7],[7,7]]
--    listaRectangular 2 3 7 [0..]        ==  [[0,1,2],[3,4,5]]
--    listaRectangular 3 2 7 [0..]        ==  [[0,1],[2,3],[4,5]]
--    listaRectangular 3 2 'p' "eva"      ==  ["ev","ap","pp"]
--    listaRectangular 3 2 'p' ['e'..]    ==  ["ef","gh","ij"]
-- ---------------------------------------------------------------------

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

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

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

```
                      ys <- [drop m xs ++ (replicate m x)]]

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4 [2 puntos] Las expresiones aritméticas se pueden definir
-- usando el siguiente tipo de datos
--    data Expr = N Int
--              | S Expr Expr
--              | P Expr Expr
--              deriving (Eq, Show)
-- Por ejemplo, la expresión
--    3*5 + 6*7
-- se puede definir por
--    S (P (N 3) (N 5)) (P (N 6) (N 7))
--
-- Definir la función
--    aplica :: (Int -> Int) -> Expr -> Expr
-- tal que (aplica f e) es la expresión obtenida aplicando la función f
-- a cada uno de los números de la expresión e. Por ejemplo,
--    ghci> aplica (+2) (S (P (N 3) (N 5)) (P (N 6) (N 7)))
--    S (P (N 5) (N 7)) (P (N 8) (N 9))
--    ghci> aplica (*2) (S (P (N 3) (N 5)) (P (N 6) (N 7)))
--    S (P (N 6) (N 10)) (P (N 12) (N 14))
-- ---------------------------------------------------------------------

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

aplica :: (Int -> Int) -> Expr -> Expr
```

```haskell
aplica f (N x)     = N (f x)
aplica f (S e1 e2) = S (aplica f e1) (aplica f e2)
aplica f (P e1 e2) = P (aplica f e1) (aplica f e2)

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] Las matrices enteras se pueden representar
-- mediante tablas con índices enteros:
--    type Matriz = Array (Int,Int) Int
-- Por ejemplo, la matriz
--    |4 1 3|
--    |1 2 8|
--    |6 5 7|
-- se puede definir por
--    listArray ((1,1),(3,3)) [4,1,3, 1,2,8, 6,5,7]
--
-- Definir la función
--    sumaColumnas :: Matriz -> Matriz
-- tal que (sumaColumnas p) es la matriz obtenida sumando a cada columna
-- la anterior salvo a la primera que le suma la última columna. Por
-- ejemplo,
--    ghci> sumaColumnas (listArray ((1,1),(3,3)) [4,1,3, 1,2,8, 6,5,7])
--    array ((1,1),(3,3)) [((1,1),7), ((1,2),5), ((1,3),4),
--                         ((2,1),9), ((2,2),3), ((2,3),10),
--                         ((3,1),13),((3,2),11),((3,3),12)]
-- es decir, el resultado es la matriz
--    | 7  5  4|
--    | 9  3 10|
--    |13 11 12|
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

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

1.6. Examen 6 (18 de Junio de 2014)

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1 [2 puntos]. Definir la función
--    divisiblesPorPrimero :: [Int] -> Bool
-- tal que (divisibles xs) se verifica si todos los elementos positivos
-- de xs son divisibles por el primero. Por ejemplo,
--    divisiblesPorPrimero [2,6,-3,0,18,-17,10]  ==  True
--    divisiblesPorPrimero [-13]                 ==  True
--    divisiblesPorPrimero [-3,6,1,-3,9,18]      ==  False
--    divisiblesPorPrimero [5,-2,-6,3]           ==  False
--    divisiblesPorPrimero []                    ==  False
--    divisiblesPorPrimero [0,2,4]               ==  False
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2 [2 puntos]. Definir la constante
```

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3 [2 puntos]. Un elemento x de un conjunto xs es minimal
-- respecto de una relación r si no existe ningún elemento y en xs tal
-- que (r y x). Por ejemplo,
--
```

```haskell
-- Definir la función
--    minimales :: Eq a => (a -> a -> Bool) -> [a] -> [a]
-- tal que (minimales xss) es la lista de los elementos minimales de
-- xs. Por ejemplo,
--    ghci> minimales (\x y -> y `rem` x == 0) [2,3,6,12,18]
--    [2,3]
--    ghci> minimales (\x y -> x `rem` y == 0) [2,3,6,12,18]
--    [12,18]
--    ghci> minimales (\x y -> maximum x < maximum y) ["ac","cd","aacb"]
--    ["ac","aacb"]
--    ghci> minimales (\xs ys -> all (`elem` ys) xs) ["ab","c","abc","d","dc"]
--    ["ab","c","d"]
-- ---------------------------------------------------------------------

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

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

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

```
--                                      0,  0, 0, 1]
--    ej2 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
--                                      0, -2, 0, 0,
--                                      1,  0, 0, 0,
--                                      0,  1, 0, 1]
-- Definir la función
--    esMonomial :: Matriz -> Bool
-- tal que (esMonomial p) se verifica si la matriz p es monomial. Por
-- ejemplo,
--    esMonomial ej1  ==  True
--    esMonomial ej2  ==  False
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

ej1, ej2 :: Matriz
ej1 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
                               0, -2, 0, 0,
                               1,  0, 0, 0,
                               0,  0, 0, 1]
ej2 = listArray ((1,1),(4,4)) [0,  0, 3, 0,
                               0, -2, 0, 0,
                               1,  0, 0, 0,
                               0,  1, 0, 1]

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5 [2 puntos]. Se consideran las expresiones vectoriales
-- formadas por un vector, la suma de dos expresiones vectoriales o el
-- producto de un entero por una expresión vectorial. El siguiente tipo
-- de dato define las expresiones vectoriales
--    data ExpV = Vec Int Int
--              | Sum ExpV ExpV
--              | Mul Int ExpV
--              deriving Show
--
-- Definir la función
--    valor :: ExpV -> (Int,Int)
-- tal que (valor e) es el valor de la expresión vectorial c. Por
-- ejemplo,
--    valor (Vec 1 2)                                  ==  (1,2)
--    valor (Sum (Vec 1 2 ) (Vec 3 4))                 ==  (4,6)
--    valor (Mul 2 (Vec 3 4))                          ==  (6,8)
--    valor (Mul 2 (Sum (Vec 1 2 ) (Vec 3 4)))         ==  (8,12)
--    valor (Sum (Mul 2 (Vec 1 2)) (Mul 2 (Vec 3 4)))  ==  (8,12)
-- ---------------------------------------------------------------------

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

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

```haskell
valor (Mul n e)   = (n*x,n*y) where (x,y) = valor e

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

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

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

1.7. Examen 7 (4 de Julio de 2014)

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

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

import Data.List
import Data.Array

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

```
--    esCompleta [6]        ==  True
--    esCompleta [6,7]      ==  True
--    esCompleta [6,8]      ==  False
--    esCompleta [6,7,9]    ==  True
--    esCompleta [8,7,5]    ==  True
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    unionG :: Ord a => [[a]] -> [a]
-- tal que (unionG xss) es la unión de xss cuyos elementos son listas
-- estrictamente crecientes (posiblemente infinitas). Por ejemplo,
--    ghci> take 10 (unionG [[2,4..],[3,6..],[5,10..]])
--    [2,3,4,5,6,8,9,10,12,14]
--    ghci> take 10 (unionG [[2,5..],[3,8..],[4,10..],[16..]])
--    [2,3,4,5,8,10,11,13,14,16]
--    ghci> unionG [[3,8],[4,10],[2,5],[16]]
--    [2,3,4,5,8,10,16]
-- ---------------------------------------------------------------------

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

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

-- 2ª definición (por plegado)
-- ===========================
```

```haskell
unionG2 :: Ord a => [[a]] -> [a]
unionG2 = foldr unionB []

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

-- 1ª solución
-- ===========
noEsSuma1 :: [Integer] -> Integer
noEsSuma1 xs = head [n | n <- [1..], n `notElem` sumas1 xs]

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

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

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

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

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

-- Las comparaciones son
--    ghci> noEsSuma1 ([1..10]++[12..20])
--    200
--    (8.28 secs, 946961604 bytes)
--    ghci> noEsSuma2 ([1..10]++[12..20])
--    200
--    (2.52 secs, 204156056 bytes)
--    ghci> noEsSuma3 ([1..10]++[12..20])
--    200
--    (0.01 secs, 520348 bytes)
--
--    ghci> noEsSuma2 (1:[2,4..30])
--    242
--    (4.97 secs, 399205788 bytes)
--    ghci> noEsSuma3 (1:[2,4..30])
--    242
--    (0.01 secs, 514340 bytes)
--
--    ghci> noEsSuma3 (1:[2,4..2014])
--    1015058
--    (0.01 secs, 1063600 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Los divisores medios de un número son los que
-- ocupan la posición media entre los divisores de n, ordenados de menor
-- a mayor. Por ejemplo, los divisores de 60 son
-- [1,2,3,4,5,6,10,12,15,20,30,60] y sus divisores medios son 6 y 10.
--
-- El árbol de factorización de un número compuesto n se construye de la
```

```
-- siguiente manera:
--    * la raíz es el número n,
--    * la rama izquierda es el árbol de factorización de su divisor
--      medio menor y
--    * la rama derecha es el árbol de factorización de su divisor
--      medio mayor
-- Si el número es primo, su árbol de factorización sólo tiene una hoja
-- con dicho número. Por ejemplo, el árbol de factorización de 60 es
--        60
--       /  \
--      6    10
--     / \   / \
--    2   3 2   5
--
-- Los árboles se representarán por
--    data Arbol = H Int
--               | N Int Arbol Arbol
--               deriving Show
--
-- Definir la función
--    arbolFactorizacion :: Int -> Arbol
-- tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
-- ejemplo,
--    ghci> arbolFactorizacion 60
--    N 60 (N 6 (H 2) (H 3)) (N 10 (H 2) (H 5))
--    ghci> arbolFactorizacion 45
--    N 45 (H 5) (N 9 (H 3) (H 3))
--    ghci> arbolFactorizacion 7
--    H 7
--    ghci> arbolFactorizacion 14
--    N 14 (H 2) (H 7)
--    ghci> arbolFactorizacion 28
--    N 28 (N 4 (H 2) (H 2)) (H 7)
--    ghci> arbolFactorizacion 84
--    N 84 (H 7) (N 12 (H 3) (N 4 (H 2) (H 2)))
-- ---------------------------------------------------------------------

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

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

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

-- (divisoresMedio n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio 30  ==  (5,6)
--    divisoresMedio  7  ==  (1,7)
divisoresMedio :: Int -> (Int,Int)
divisoresMedio n = (n `div` x,x)
    where xs = divisores n
          x  = xs !! (length xs `div` 2)

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

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

-- (divisoresMedio2 n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio2 30  ==  (5,6)
```

```haskell
--    divisoresMedio2  7  ==  (1,7)
divisoresMedio2 :: Int -> (Int,Int)
divisoresMedio2 n = (n `div` x,x)
    where m  = ceiling (sqrt (fromIntegral n))
          x = head [y | y <- [m..n], n `rem` y == 0]

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

```
--    array ((1,1),(5,5))
--          [((1,1),1),((1,2),0),((1,3),0),((1,4),0),((1,5),0),
--           ((2,1),1),((2,2),1),((2,3),0),((2,4),0),((2,5),0),
--           ((3,1),1),((3,2),2),((3,3),1),((3,4),0),((3,5),0),
--           ((4,1),1),((4,2),3),((4,3),3),((4,4),1),((4,5),0),
--           ((5,1),1),((5,2),4),((5,3),6),((5,4),4),((5,5),1)]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

-- 1ª solución
-- ===========

matrizPascal1 :: Int -> Matriz
matrizPascal1 1 = array ((1,1),(1,1)) [((1,1),1)]
matrizPascal1 n =
    array ((1,1),(n,n)) [((i,j), f i j) | i <- [1..n], j <- [1..n]]
    where f i j | i < n && j <  n  = p!(i,j)
                | i < n && j == n  = 0
                | j == 1 || j == n = 1
                | otherwise        = p!(i-1,j-1) + p!(i-1,j)
          p = matrizPascal2 (n-1)

-- 2ª solución
-- ===========

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

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


-- 2ª solución
-- ===========

matrizPascal3 :: Int -> Matriz
```

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

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

1.8. Examen 8 (10 de Septiembre de 2014)

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

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

import Data.List
import Data.Array

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

```haskell
-- En efecto,   3*8 = 24,  38+8 = 46  y   4*6 = 24
--            2*2*6 = 24, 226+8 = 234 y 2*3*4 = 24
--
-- Usando la función especiales, calcular la solución del problema.
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Las expresiones aritméticas pueden
-- representarse usando el siguiente tipo de datos
--    data Expr = N Int | S Expr Expr | P Expr Expr
--              deriving Show
-- Por ejemplo, la expresión 2*(3+7) se representa por
--    P (N 2) (S (N 3) (N 7))
-- La dual de una expresión es la expresión obtenida intercambiando las
-- sumas y los productos. Por ejemplo, la dual de 2*(3+7) es 2+(3*7).
--
-- Definir la función
--    dual :: Expr -> Expr
-- tal que (dual e) es la dual de la expresión e. Por ejemplo,
--    dual (P (N 2) (S (N 3) (N 7)))  ==  S (N 2) (P (N 3) (N 7))
-- ---------------------------------------------------------------------

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

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

```haskell
dual (P e1 e2) = S (dual e1) (dual e2)

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

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

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

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

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

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

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

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


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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] (Basado en el problema 196 del proyecto
-- Euler). Para cada número n la matriz completa de orden n es la matriz
-- cuadrada de orden n formada por los números enteros consecutivos. Por
-- ejemplo, la matriz completa de orden 3 es
--    |1 2 3|
--    |4 5 6|
--    |7 8 9|
-- las ternas primas de orden n son los listas formadas por un
-- elemento de la matriz junto con dos de sus vecinos de manera que los
-- tres son primos. Por ejemplo, en la matriz anterior una terna prima
-- es [2,3,5] (formada por el elemento 2, su vecino derecho 3 y su
-- vecino inferior 5), otra es [5,2,7] (formada por el elemento 5, su
-- vecino superior 2 y su vecino inferior-izquierda 7) y otra es [5,3,7]
-- (formada por el elemento 5, su vecino superior-derecha 3 y y su
-- vecino inferior-izquierda 7).
--
-- Definir la función
--    ternasPrimasOrden :: Int -> [[Int]]
-- tal que (ternasPrimasOrden n) es el conjunto de las ternas primas de
-- la matriz completa de orden n. Por ejemplo,
--    ghci> ternasPrimasOrden 3
--    [[2,3,5],[3,2,5],[5,2,3],[5,2,7],[5,3,7]]
--    ghci> ternasPrimasOrden 4
--    [[2,3,5],[2,3,7],[2,5,7],[3,2,7],[7,2,3],[7,2,11],[7,3,11]]
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

type Matriz = Array (Int,Int) Int
```

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

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

-- (ternas p) es la lista de las ternas de p formadas por un elemento de
-- p junto con dos vecinos. Por ejemplo,
--    ghci>  ternas (listArray ((1,1),(3,3)) [2,3,7,5,4,0,6,8,9])
--     [[2,3,5],[2,3,4],[2,5,4],[3,2,7],[3,2,5],[3,2,4],[3,2,0],[3,7,5],
--      [3,7,4],[3,7,0],[3,5,4],[3,5,0],[3,4,0],[7,3,4],[7,3,0],[7,4,0],
--      [5,2,3],[5,2,4],[5,2,6],[5,2,8],[5,3,4],[5,3,6],[5,3,8],[5,4,6],
--      [5,4,8],[5,6,8],[4,2,3],[4,2,7],[4,2,5],[4,2,0],[4,2,6],[4,2,8],
--      [4,2,9],[4,3,7],[4,3,5],[4,3,0],[4,3,6],[4,3,8],[4,3,9],[4,7,5],
--      [4,7,0],[4,7,6],[4,7,8],[4,7,9],[4,5,0],[4,5,6],[4,5,8],[4,5,9],
--      [4,0,6],[4,0,8],[4,0,9],[4,6,8],[4,6,9],[4,8,9],[0,3,7],[0,3,4],
--      [0,3,8],[0,3,9],[0,7,4],[0,7,8],[0,7,9],[0,4,8],[0,4,9],[0,8,9],
--      [6,5,4],[6,5,8],[6,4,8],[8,5,4],[8,5,0],[8,5,6],[8,5,9],[8,4,0],
--      [8,4,6],[8,4,9],[8,0,6],[8,0,9],[8,6,9],[9,4,0],[9,4,8],[9,0,8]]
ternas :: Matriz -> [[Int]]
ternas p =
    [[p!(i1,j1),p!(i2,j2),p!(i3,j3)] |
     (i1,j1) <- indices p,
     ((i2,j2):ps) <- tails (vecinos (i1,j1) n),
     (i3,j3) <- ps]
    where (_,(n,_)) = bounds p

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

```
                          (a,b) /= (i,j)]

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

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

-- 2ª definición
-- =============

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

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

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

```
--    (0.46 secs, 18091148 bytes)
```

1.9. Examen 9 (20 de Noviembre de 2014)

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

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

import Data.List
import Data.Array

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sinDobleCero :: Int -> [[Int]]
-- tal que (sinDobleCero n) es la lista de las listas de longitud n
-- formadas por el 0 y el 1 tales que no contiene dos ceros
-- consecutivos. Por ejemplo,
--    ghci> sinDobleCero 2
--    [[1,0],[1,1],[0,1]]
```

```
--    ghci> sinDobleCero 3
--    [[1,1,0],[1,1,1],[1,0,1],[0,1,0],[0,1,1]]
--    ghci> sinDobleCero 4
--    [[1,1,1,0],[1,1,1,1],[1,1,0,1],[1,0,1,0],[1,0,1,1],
--     [0,1,1,0],[0,1,1,1],[0,1,0,1]]
-- ---------------------------------------------------------------------

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

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

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

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

-- 2ª definición de digitosPrimos:
digitosPrimos2 :: Int -> Bool
```

```
digitosPrimos2 n = subconjunto (cifras n) [2,3,5,7]

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

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

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

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

```haskell
--    ghci> opMatriz (-) a b
--    array ((1,1),(2,3)) [((1,1),2),((1,2),0),((1,3),4),
--                         ((2,1),3),((2,2),5),((2,3),5)]
-- ---------------------------------------------------------------------

type Matriz = Array (Int,Int) Int

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

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

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

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

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

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

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

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

2

Exámenes del grupo 2

Antonia M. Chávez

2.1. Examen 1 (6 de Noviembre de 2013)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Se dice que dos números son hermanos si tienen el
-- mismo número de divisores propios. Por ejemplo, 6 y 22 son hermanos
-- porque ambos tienen tres divisores propios.
--
-- Definir la función hermanos tal que (hermanos x y) se verifica si x e
-- y son hermanos. Por ejemplo,
--    hermanos 6 10 == True
--    hermanos 3 4  == False
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

prop_hermanos2 x = hermanos x x

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

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

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

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

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

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

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

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

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

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

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

2.2. Examen 2 (4 de Diciembre de 2013)

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

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

import Test.QuickCheck

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función primerosYultimos tal que
-- (primerosYultimos xss) es el par formado  por la lista de los
-- primeros elementos de las listas no vacías de xss y la lista de los
```

```
-- últimos elementos de las listas no vacías de xs. Por ejemplo,
--    ghci> primerosYultimos [[1,2],[5,3,4],[],[0,8,7,6],[],[9]]
--    ([1,5,0,9],[2,4,6,9])
-- ---------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

2.3. Examen 3 (23 de Enero de 2014)

El examen es común con el del grupo 3 (ver página 16).

2.4. Examen 4 (24 de Marzo de 2014)

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

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

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

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

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

ej1 = Monomio 3 2 (Monomio 2 1 (Indep (-5)))
ej2 = Monomio 4 3 (Monomio (-2) 1 (Indep 0))

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

```haskell
    | otherwise  = sPol r1 r2

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios se pueden representar mediante el
-- siguiente tipo de datos
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
--
-- Definir la función
--    profundidades :: (Num t, Eq t) => Arbol t -> t -> [t]
-- tal que (profundidades a x) es la lista de las profundidades que ocupa x
-- en el árbol a. Por ejemplo,
--    profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 1  ==  [1,2,3]
--    profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 2  ==  [3]
--    profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 3  ==  [2]
--    profundidades (N 1 (H 1) (N 3 (H 1) (H 2))) 4  ==  []
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por comprensión, la función
--    phills :: [Integer]
-- tal que phills es la sucesión de Phill. Por ejemplo,
--    take 8 phills  ==  [2,7,12,17,22,27,32,37]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir, por recursión (evaluación perezosa) la
-- función
--    phills1 :: [Integer]
```

```haskell
-- tal que phills1 es la sucesion de Phill. Por ejemplo,
--    take 8 phills1  ==  [2,7,12,17,22,27,32,37]
-- Nota: Dar dos definiciones, una sin usar zipWith o otra usándola.
-- ---------------------------------------------------------------------

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir, usando unidades, la propiedad
--    propPhill :: Int -> Bool
-- tal que (propPhill n) se verifica si el término n-ésimo de la
-- sucesión de Phill termina en 2 o en 7 según n sea par o impar.
--
-- Comprobar la propiedad para los 500 primeros términos.
-- ---------------------------------------------------------------------

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

-- La comprobación es
```

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

2.5. Examen 5 (19 de Mayo de 2014)

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

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

import Data.List
import Data.Array

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

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

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

-- 2ª definición (por recursión)
```

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

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

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

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

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

primos :: [Integer]
```

```haskell
primos = [x | x <- [1 .. ], factores x == [1,x]]

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

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

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

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

cuenta :: Integer -> Integer -> Int -> [Int]
cuenta a b n =
    map (length . takeWhile (< b)) [sucesionN a m | m <- [1 .. n]]

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

```
--    simetricos "ala"              == "a"
--    simetricos [1..20]            == []
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. La matrices piramidales son las formadas por unos y ceros
-- de forma que los unos forman una pirámide. Por ejemplo,
--   |1|   |0 1 0|   |0 0 1 0 0|   |0 0 0 1 0 0 0|
--         |1 1 1|   |0 1 1 1 0|   |0 0 1 1 1 0 0|
--                   |1 1 1 1 1|   |0 1 1 1 1 1 0|
--                                 |1 1 1 1 1 1 1|
--
-- El tipo de las matrices se define por
--    type Matriz a = Array (Int,Int) a
-- Por ejemplo, las matrices anteriores se definen por
--    p1, p2, p3 :: Matriz Int
--    p1 = listArray ((1,1),(1,1)) [1]
--    p2 = listArray ((1,1),(2,3)) [0,1,0,
--                                  1,1,1]
--    p3 = listArray ((1,1),(3,5)) [0,0,1,0,0,
--                                  0,1,1,1,0,
--                                  1,1,1,1,1]
--
-- Definir la función
--    esPiramidal :: (Eq a, Num a) => Matriz a -> Bool
-- tal que (esPiramidal p) se verifica si la matriz p es piramidal. Por
-- ejemplo,
--    esPiramidal p3                                          ==  True
--    esPiramidal (listArray ((1,1),(2,3)) [0,1,0, 1,5,1])   ==  False
--    esPiramidal (listArray ((1,1),(2,3)) [0,1,1, 1,1,1])   ==  False
--    esPiramidal (listArray ((1,1),(2,3)) [0,1,0, 1,0,1])   ==  False
-- ---------------------------------------------------------------------

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

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

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

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

-- 2ª definición
-- =============

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

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

```
                | otherwise = 0

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

-- se representan por
--    ej1, ej2,ej3 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 3 [N 5 [N 6 []],
--               N 4 [],
--               N 7 [N 2 [], N 1 []]]
--    ej3 = N 3 [N 5 [N 6 []],
--               N 4 [N 1 [N 2 [],N 3 [N 4 []]]],
--               N 7 [N 2 [], N 1 []]]
--
-- Definir la función
--     ramifica :: Arbol a -> Arbol a -> (a -> Bool) -> Arbol a
-- tal que (ramifica a1 a2 p) el árbol que resulta de añadir una copia
-- del árbol a2 a los nodos de a1 que cumplen un predicado p. Por
-- ejemplo,
--    ghci> ramifica (N 3 [N 5 [N 6 []],N 4 [],N 7 [N 2 [],N 1 []]]) (N 8 []) (>5
--    N 3 [N 5 [N 6 [N 8 []]],N 4 [],N 7 [N 2 [],N 1 [],N 8 []]]
-- ---------------------------------------------------------------------

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

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

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

2.6. Examen 6 (18 de Junio de 2014)

El examen es común con el del grupo 3 (ver página 30).

2.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 38).

2.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 45).

2.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 49).

3

Exámenes del grupo 3

María J. Hidalgo

3.1. Examen 1 (7 de Noviembre de 2013)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1 [Del problema 21 del Proyecto Euler]. Sea d(n) la suma de
-- los divisores propios de n. Si d(a) = b y d(b) = a, siendo a ≠ b,
-- decimos que a y b son un par de números amigos. Por ejemplo, los
-- divisores propios de 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y
-- 110; por tanto, d(220) = 284. Los divisores propios de 284 son 1, 2,
-- 4, 71 y 142; por tanto,  d(284) = 220.  Luego, 220 y 284 son dos
-- números amigos.
--
-- Definir la función amigos tal que (amigos a b) se verifica si a y b
-- son números amigos. Por ejemplo,
--    amigos 6 6       == False
--    amigos 220 248   == False
--    amigos 220 284   == True
--    amigos 100 200   == False
--    amigos 1184 1210 == True
-- ---------------------------------------------------------------------

amigos a b = sumaDivisores a == b && sumaDivisores b == a
    where sumaDivisores n = sum [x | x<-[1..n-1], n `rem` x == 0]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Una representación de 20 en base 2 es [0,0,1,0,1] pues
-- 20 = 1*2^2 + 1*2^4. Y una representación de 46 en base 3 es [1,0,2,1]
-- pues 46 = 1*3^0 + 0*3^1 + 2*3^2 + 1*3^3.
--
-- Definir la función deBaseABase10 tal que (deBaseABase10 b xs) es el
-- número n tal que su representación en base b es xs. Por ejemplo,
--    deBaseABase10 2 [0,0,1,0,1]      == 20
--    deBaseABase10 2 [1,1,0,1]        == 11
--    deBaseABase10 3 [1,0,2,1]        == 46
--    deBaseABase10 5 [0,2,1,3,1,4,1]  == 29160
-- ---------------------------------------------------------------------

deBaseABase10 b xs = sum [y*b^n | (y,n) <- zip xs [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. [De la IMO-1996-S-21]. Una sucesión [a(0),a(1),...,a(n)]
-- se denomina cuadrática si para cada i ∈ {1, 2,..., n} se cumple que
--    |a(i) − a(i−1)| = i^2 .
-- Definir una función esCuadratica tal que (esCuadratica xs) se
-- verifica si xs cuadrática.  Por ejemplo,
--    esCuadratica [2,1,-3,6]                     == True
--    esCuadratica [2,1,3,5]                      == False
--    esCuadratica [3,4,8,17,33,58,94,45,-19,-100] == True
-- ---------------------------------------------------------------------

esCuadratica xs =
    and [abs (y-x) == i^2 | ((x,y),i) <- zip (adyacentes xs) [1..]]

adyacentes xs = zip xs (tail xs)

-- ---------------------------------------------------------------
-- Ejercicio 4.1. Sea t una lista de pares de la forma
--    (nombre, [(asig1, nota1),...,(asigk, notak)])
-- Por ejemplo,
--    t1 = [("Ana",[("Algebra",1),("Calculo",3),("Informatica",8),("Fisica",2)]),
--          ("Juan",[("Algebra",5),("Calculo",1),("Informatica",2),("Fisica",9)]),
--          ("Alba",[("Algebra",5),("Calculo",6),("Informatica",6),("Fisica",5)]),
--          ("Pedro",[("Algebra",9),("Calculo",5),("Informatica",3),("Fisica",1)]]
```

```
-- Definir la función calificaciones tal que (calificaciones t p) es la
-- lista de las calificaciones de la persona p en la lista t. Por
-- ejemplo,
--    ghci> calificaciones t1 "Pedro"
--    [("Algebra",9),("Calculo",5),("Informatica",3),("Fisica",1)]
-- ---------------------------------------------------------------------

t1 = [("Ana",[("Algebra",1),("Calculo",3),("Informatica",8),("Fisica",2)]),
      ("Juan",[("Algebra",5),("Calculo",1),("Informatica",2),("Fisica",9)]),
      ("Alba",[("Algebra",5),("Calculo",6),("Informatica",6),("Fisica",5)]),
      ("Pedro",[("Algebra",9),("Calculo",5),("Informatica",3),("Fisica",1)])]

calificaciones t p = head [xs | (x,xs) <-t, x == p]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función todasAprobadas tal que
-- (todasAprobadas t p) se cumple si en la lista t, p tiene todas las
-- asignaturas aprobadas. Por ejemplo,
--    todasAprobadas t1 "Alba"  ==  True
--    todasAprobadas t1 "Pedro" ==  False
-- ---------------------------------------------------------------------

todasAprobadas t p = numeroAprobados t p == numeroAsignaturas t p

numeroAprobados t p = length [n | (_,n) <- calificaciones t p, n >= 5]

numeroAsignaturas t p = length (calificaciones t p)

apruebanTodo t = [p | (p,_) <- t, todasAprobadas t p]
```

3.2. Examen 2 (19 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (19 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

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

```haskell
-- Ejercicio 1.1. Definir la función
--    bocata :: Eq a => a -> a -> [a] -> [a]
-- tal que (bocata x y zs) es la lista obtenida colocando y delante y
-- detrás de todos los elementos de zs que coinciden con x. Por ejemplo,
--    > bocata "chorizo" "pan" ["jamon", "chorizo", "queso", "chorizo"]
--    ["jamon","pan","chorizo","pan","queso","pan","chorizo","pan"]
--    > bocata "chorizo" "pan" ["jamon", "queso", "atun"]
--    ["jamon","queso","atun"]
-- ---------------------------------------------------------------------

bocata :: Eq a => a -> a -> [a] -> [a]
bocata _ _ []     = []
bocata x y (z:zs) | z == x    = y : z : y : bocata x y zs
                  | otherwise = z : bocata x y zs

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que el número de elementos de
-- (bocata a b xs) es el número de elementos de xs más el doble del
-- número de elementos de xs que coinciden con a.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_bocata :: String -> String -> [String] -> Bool
prop_bocata a b xs =
    length (bocata a b xs) == length xs + 2 * length (filter (==a) xs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    mezclaDigitos :: Integer -> Integer -> Integer
-- tal que (mezclaDigitos n m) es el número formado intercalando los
-- dígitos de n y m, empezando por los de n. Por ejemplo,
--    mezclaDigitos 12583 4519       == 142551893
--    mezclaDigitos 12583 4519091256 == 142551893091256
-- ---------------------------------------------------------------------

mezclaDigitos :: Integer -> Integer -> Integer
```

```haskell
mezclaDigitos n m =
    read (intercala (show n) (show m))

-- (intercala xs ys) es la lista obtenida intercalando los elementos de
-- xs e ys. Por ejemplo,
--    intercala [2,5,3] [4,7,9,6,0]  ==  [2,4,5,7,3,9,6,0]
--    intercala [4,7,9,6,0] [2,5,3]   ==  [4,2,7,5,9,3,6,0]
intercala :: [a] -> [a] -> [a]
intercala [] ys = ys
intercala xs [] = xs
intercala (x:xs) (y:ys) = x : y : intercala xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Problema 211 del proyecto Euler) Dado un entero
-- positivo n, consideremos la suma de los cuadrados de sus divisores,
-- Por ejemplo,
--    f(10) = 1 + 4 + 25 + 100 = 130
--    f(42) = 1 + 4 +  9 +  36 + 49 + 196 + 441 + 1764 = 2500
-- Decimos que n es especial si f(n) es un cuadrado perfecto. En los
-- ejemplos anteriores, 42 es especial y 10 no lo es.
--
-- Definir la función
--    especial:: Int -> Bool
-- tal que (especial x) se verifica si x es un número es especial. Por
-- ejemplo,
--    especial 42  ==  True
--    especial 10  ==  False
-- Calcular todos los números especiales de tres cifras.
-- ---------------------------------------------------------------------

especial:: Int -> Bool
especial n = esCuadrado (sum (map (^2) (divisores n)))

-- (esCuadrado n) se verifica si n es un cuadrado perfecto. Por ejemplo,
--    esCuadrado 36  ==  True
--    esCuadrado 40  ==  False
esCuadrado :: Int -> Bool
esCuadrado n = y^2 == n
    where y = floor (sqrt (fromIntegral n))
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    verificaMax :: (a -> Bool) -> [[a]] -> [a]
-- tal que (verificaMax p xss) es la lista de xss con mayor número de
-- elementos que verifican la propiedad p. Por ejemplo,
--    ghci> verificaMax even [[1..5], [2,4..20], [3,2,1,4,8]]
--    [2,4,6,8,10,12,14,16,18,20]
--    ghci> verificaMax even [[1,2,3], [6,8], [3,2,10], [3]]
--    [6,8]
-- Nota: En caso de que haya más de una lista, obtener la primera.
-- ---------------------------------------------------------------------

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

3.3. Examen 3 (23 de Enero de 2014)

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

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

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

import Data.List

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4. Examen 4 (20 de Marzo de 2014)

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

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

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

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

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

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

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

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

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

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

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

-- El cálculo es
--    ghci> suc' 100
--    3006843434908259388021180629494759675292054662578918108250552307032128680700

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. El método de la bisección para calcular un cero de una
-- función en el intervalo [a,b] se basa en el Teorema de Bolzano: "Si
-- f(x) es una función continua en el intervalo [a, b], y si, además, en
-- los  extremos del intervalo la función f(x) toma valores de signo
-- opuesto (f(a) * f(b) < 0), entonces existe al menos un valor c en (a,
```

```
-- b) para el que f(c) = 0".
--
-- La idea es tomar el punto medio del intervalo c = (a+b)/2 y
-- considerar los siguientes casos:
-- (*) Si f(c) ~= 0, hemos encontrado una aproximación del punto que
--     anula f en el intervalo con un error aceptable.
-- (*) Si f(c) tiene signo distinto de f(a), repetir el proceso en el
--     intervalo [a,c].
-- (*) Si no, repetir el proceso en el intervalo [c,b].
--
-- Definir la función
--    ceroBiseccionE :: (Double -> Double) ->
--                      Double -> Double -> Double -> Double
-- tal que (ceroBiseccionE f a b e) calcule una aproximación del punto
-- del intervalo [a,b] en el que se anula la función f, con un error
-- menor que e, aplicando el método de la bisección. Por ejemplo,
-- si f1 y f2 son las funciones definidas por
--    f1 x = 2 - x
--    f2 x = x^2 - 3
-- entonces
--    ceroBiseccionE f1 0 3 0.0001     == 2.00006103515625
--    ceroBiseccionE f2 0 2 0.0001     == 1.7320556640625
--    ceroBiseccionE f2 (-2) 2 0.00001 == -1.732048
--    ceroBiseccionE cos 0 2 0.0001    == -1.7320480346679688
-- ---------------------------------------------------------------------

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

ceroBiseccionE :: (Double -> Double) ->
                  Double -> Double -> Double -> Double
ceroBiseccionE f a b e = aux a b
    where aux c d | aceptable m     = m
                  | (f c)*(f m) < 0 = aux c m
                  | otherwise       = aux m d
              where m = (c+d)/2
                    aceptable x = abs (f x) < e

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Los polinomios de Fibonacci se definen como sigue
```

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

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

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

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

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

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

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

3.5. Examen 5 (15 de Mayo de 2014)

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

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

import Data.List
import Data.Array

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

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

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

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

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

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

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

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Las sucesiones así construidas tienen un elemento
-- común, a partir del cual los términos coinciden. Por ejemplo,
--    take 7 (sucesion 3)  ==  [3,6,    12,14,18,26,38]
--    take 7 (sucesion 5)  ==  [5,10,11,12,14,18,26]
-- se observa que las sucesiones que empiezan en 3 y 5, respectivamente,
-- coinciden a partir del término 12.
--
-- Definir la función
--     comun :: Integer -> Integer -> Integer
-- tal que (comun x y) es el primer elemento común de las sucesiones que
-- empiezan en x e y, respectivamente. Por ejemplo,
--     comun 3 5     == 12
--     comun 3 4     == 26
--     comun 3 8     == 26
--     comun 3 20    == 26
```

```haskell
--     comun 3 34    == 126
--     comun 234 567 == 1474
-- ---------------------------------------------------------------------

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

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

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


-- ---------------------------------------------------------------------
-- Ejercicio 4. Se consideran los árboles binarios definidos por
--    data Arbol a = Hoja | Nodo a (Arbol a) (Arbol a)
--                   deriving Show
--
-- Por ejemplo, el árbol
--            5
--           / \
--          /   \
--         4     7
--        / \   / \
--       1         8
```

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

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Los vectores son tablas cuyos índices son números
-- naturales.
--    type Vector a = Array Int a
-- Las matrices son tablas cuyos índices son pares de números naturales.
--    type Matriz a = Array (Int,Int) a
-- Por ejemplo,
--    c1, c2:: Matriz Double
--    c1 = listArray ((1,1),(4,4)) [1,3,0,0,
--                                  -1, 1,-1, 1,
--                                  1,-1, 1,-1,
--                                  1, 1,-1, 1]
--
--    c2 = listArray ((1,1),(2,2)) [1,1,1,-1]
```

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

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

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

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

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


-- (submatriz i j p) es la submatriz de p obtenida eliminado la fila i y
-- la columna j. Por ejemplo,
--    submatriz 2 3 (listArray ((1,1),(3,3)) [2,1,5,1,2,3,5,4,2])
--    array ((1,1),(2,2)) [((1,1),2),((1,2),1),((2,1),5),((2,2),4)]
--    submatriz 2 3 (listArray ((1,1),(3,3)) [1..9])
--    array ((1,1),(2,2)) [((1,1),1),((1,2),2),((2,1),7),((2,2),8)]
submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p = array ((1,1), (m-1,n -1))
                  [((k,l), p ! f k l) | k <- [1..m-1], l <- [1..n-1]]
                      where (_,(m,n)) = bounds p
                            f k l | k <  i && l <  j = (k,l)
```

```
                                    | k >= i && l <  j = (k+1,l)
                                    | k <  i && l >= j = (k,l+1)
                                    | otherwise = (k+1,l+1)
```

3.6. Examen 6 (18 de Junio de 2014)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

type Matriz a = Array (Int,Int) a

-- 1ª solución
-- ===========

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

-- 2ª solución
-- ===========

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

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

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

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

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

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

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


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

-- 2ª solución para esSumaCuadradosConsecutivos:
```

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

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

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

3.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 38).

3.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 45).

3.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 49).

4

Exámenes del grupo 4

Francisco J. Martín

4.1. Examen 1 (5 de Noviembre de 2013)

```
-- Informática (1º del Grado en Matemáticas y en Estadística)
-- 1º examen de evaluación continua (11 de noviembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función listaIgualParidad tal que al
-- evaluarla sobre una lista de números naturales devuelva la lista de
-- todos los elementos con la misma paridad que la posición que ocupan
-- (contada desde 0); es decir, todos los pares en una posición par y
-- todos los impares en una posición impar. Por ejemplo,
--    listaIgualParidad [1,3,5,7]  ==  [3,7]
--    listaIgualParidad [2,4,6,8]  ==  [2,6]
--    listaIgualParidad [1..10]    ==  []
--    listaIgualParidad [0..10]    ==  [0,1,2,3,4,5,6,7,8,9,10]
--    listaIgualParidad []         ==  []
-- ---------------------------------------------------------------------

listaIgualParidad xs = [x | (x,i) <- zip xs [0..], even x == even i]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Decimos que una lista está equilibrada si el número de
-- elementos de la lista que son menores que la media es igual al número
-- de elementos de la lista que son mayores.
--
```

```haskell
-- Definir la función listaEquilibrada que comprueba dicha propiedad
-- para una lista. Por ejemplo,
--    listaEquilibrada [1,7,1,6,2]  ==  False
--    listaEquilibrada [1,7,4,6,2]  ==  True
--    listaEquilibrada [8,7,4,6,2]  ==  False
--    listaEquilibrada []           ==  True
-- ---------------------------------------------------------------------

listaEquilibrada xs =
    length [y | y <- xs, y < media xs] ==
    length [y | y <- xs, y > media xs]

-- (media xs) es la media de xs. Por ejemplo,
--    media [6, 3, 9]  ==  6.0
media xs = sum xs / fromIntegral (length xs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. El trozo inicial de los elementos de una lista que
-- cumplen una propiedad es la secuencia de elementos de dicha lista
-- desde la posición 0 hasta el primer elemento que no cumple la
-- propiedad, sin incluirlo.
--
-- Definirla función trozoInicialPares que devuelve el trozo inicial de
-- los elementos de una lista que son pares. Por ejemplo,
--    trozoInicialPares []         ==  []
--    trozoInicialPares [1,2,3,4]  ==  []
--    trozoInicialPares [2,4,3,2]  ==  [2,4]
--    trozoInicialPares [2,4,6,8]  ==  [2,4,6,8]
-- ---------------------------------------------------------------------

trozoInicialPares xs = take (posicionPrimerImpar xs) xs

-- (posicionPrimerImpar xs) es la posición del primer elemento impar de
-- la lista xs o su longitud si no hay ninguno. Por ejemplo,
--    posicionPrimerImpar [2,4,3,2]  ==  2
--    posicionPrimerImpar [2,4,6,2]  ==  4
posicionPrimerImpar xs =
    head ([i | (x,i) <- zip xs [0..], odd x] ++ [length xs])

-- La función anterior se puede definir por recursión
```

```
posicionPrimerImpar2 [] = 0
posicionPrimerImpar2 (x:xs)
    | odd x     = 0
    | otherwise = 1 + posicionPrimerImpar2 xs

-- 2ª definición (por recursión).
trozoInicialPares2 [] = []
trozoInicialPares2 (x:xs) | odd x     = []
                          | otherwise = x : trozoInicialPares2 xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El registro de entradas vendidas de un cine se
-- almacena en una lista en la que cada elemento tiene el título de una
-- película, a continuación el número de entradas vendidas sin promoción
-- a 6 euros y por último el número de entradas vendidas con alguna de las
-- promociones del cine (menores de 4 años, mayores de 60, estudiantes
-- con carnet) a 4 euros. Por ejemplo,
--    entradas = [("Gravity",22,13),("Séptimo",18,6), ("Turbo",19,0),
--                ("Gravity",10,2), ("Séptimo",22,10),("Turbo",32,10),
--                ("Gravity",18,8), ("Séptimo",20,14),("Turbo",18,10)]
--
-- Definir la función ingresos tal que (ingresos bd) sea el total de
-- ingresos obtenidos según la información sobre entradas vendidas
-- almacenada en la lista 'bd'. Por ejemplo,
--    ingressos entradas  =  1366
-- ---------------------------------------------------------------------

entradas = [("Gravity",22,13),("Séptimo",18,6), ("Turbo",19,0),
            ("Gravity",10,2), ("Séptimo",22,10),("Turbo",32,10),
            ("Gravity",18,8), ("Séptimo",20,14),("Turbo",18,10)]

ingresos bd = sum [6*y+4*z | (_,y,z) <- bd]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función ingresosPelicula tal que
-- (ingresos bd p) sea el total de ingresos obtenidos en las distintas
-- sesiones de la película p según la información sobre entradas
-- vendidas almacenada en la lista bd. Por ejemplo,
--    ingresosPelicula entradas "Gravity"  ==  392
--    ingresosPelicula entradas "Séptimo"  ==  480
```

```
--    ingresosPelicula entradas "Turbo"    ==  494
-- ---------------------------------------------------------------------

ingresosPelicula bd p = sum [6*y+4*z | (x,y,z) <- bd, x == p]

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

4.2. Examen 2 (16 de Diciembre de 2013)

```
-- Informática (1º del Grado en Matemáticas y en Estadística)
-- 2º examen de evaluación continua (16 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Test.QuickCheck

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por comprensión, la función
-- numeroConsecutivosC tal que (numeroConsecutivosC xs) es la cantidad
-- de números
-- consecutivos que aparecen al comienzo de la lista xs. Por ejemplo,
--    numeroConsecutivosC [1,3,5,7,9]      ==  1
--    numeroConsecutivosC [1,2,3,4,5,7,9]  ==  5
--    numeroConsecutivosC []               ==  0
--    numeroConsecutivosC [4,5]            ==  2
--    numeroConsecutivosC [4,7]            ==  1
--    numeroConsecutivosC [4,5,0,4,5,6]    ==  2
-- ---------------------------------------------------------------------

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

-- 2ª solución (con zip3)
```

```
numeroConsecutivosC2 :: (Num a, Eq a) => [a] -> Int
numeroConsecutivosC2 [] = 0
numeroConsecutivosC2 [x] = 1
numeroConsecutivosC2 [x,y] | x+1 == y  = 2
                           | otherwise = 1
numeroConsecutivosC2 xs =
    head [k | (x,y,k) <- zip3 xs (tail xs) [1..], y /= x+1]

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función numeroConsecutivosR
-- tal que (numeroConsecutivosR xs) es la cantidad de números
-- consecutivos que aparecen al comienzo de la lista xs. Por ejemplo,
--    numeroConsecutivosC [1,3,5,7,9]      ==  1
--    numeroConsecutivosC [1,2,3,4,5,7,9]  ==  5
--    numeroConsecutivosC []               ==  0
--    numeroConsecutivosC [4,5]            ==  2
--    numeroConsecutivosC [4,7]            ==  1
--    numeroConsecutivosC [4,5,0,4,5,6]    ==  2
-- ---------------------------------------------------------------------

numeroConsecutivosR []  = 0
numeroConsecutivosR [x] = 1
numeroConsecutivosR (x:y:ys)
    | y == x+1  = 1 + numeroConsecutivosR (y:ys)
    | otherwise = 1

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Una sustitución es una lista de parejas
-- [(x1,y1),...,(xn,yn)] que se usa para indicar que hay que reemplazar
-- cualquier ocurrencia de cada uno de los xi, por el correspondiente
-- yi. Por ejemplo,
--    sustitucion = [('1','a'),('2','n'),('3','v'),('4','i'),('5','d')]
-- es la sustitución que reemplaza '1' por 'a', '2' por 'n', ...
--
-- Definir, por comprensión, la función sustitucionEltC tal que
```

```haskell
-- (sustitucionEltC xs z) es el resultado de aplicar la sustitución xs
-- al elemento z. Por ejemplo,
--    sustitucionEltC sustitucion '4'  ==  'i'
--    sustitucionEltC sustitucion '2'  ==  'n'
--    sustitucionEltC sustitucion '0'  ==  '0'
-- ----------------------------------------------------------------------

sustitucion = [('1','a'),('2','n'),('3','v'),('4','i'),('5','d')]

sustitucionEltC xs z = head [y | (x,y) <- xs, x == z] ++ [z]

-- ----------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función sustitucionEltR tal
-- que (sustitucionEltR xs z) es el resultado de aplicar la sustitución
-- xs al elemento z. Por ejemplo,
--    sustitucionEltR sustitucion '4'  ==  'i'
--    sustitucionEltR sustitucion '2'  ==  'n'
--    sustitucionEltR sustitucion '0'  ==  '0'
-- ----------------------------------------------------------------------

sustitucionEltR [] z = z
sustitucionEltR ((x,y):xs) z
    | x == z    = y
    | otherwise = sustitucionEltR xs z

-- ----------------------------------------------------------------------
-- Ejercicio 2.3, Definir, por comprensión, la función sustitucionLstC
-- tal que (sustitucionLstC xs zs) es el resultado de aplicar la
-- sustitución xs a los elementos de la lista zs. Por ejemplo,
--    sustitucionLstC sustitucion "2151"     ==  "nada"
--    sustitucionLstC sustitucion "3451"     ==  "vida"
--    sustitucionLstC sustitucion "2134515"  ==  "navidad"
-- ----------------------------------------------------------------------

sustitucionLstC xs zs = [sustitucionEltC xs z | z <- zs]

-- ----------------------------------------------------------------------
-- Ejercicio 2.4. Definir, por recursión, la función sustitucionLstR tal
-- que (sustitucionLstR xs zs) es el resultado de aplicar la sustitución
-- xs a los elementos de la lista zs. Por ejemplo,
```

```
--    sustitucionLstR sustitucion "2151"     ==  "nada"
--    sustitucionLstR sustitucion "3451"     ==  "vida"
--    sustitucionLstR sustitucion "2134515"  ==  "navidad"
-- ---------------------------------------------------------------------

sustitucionLstR xs []     = []
sustitucionLstR xs (z:zs) =
     sustitucionEltR xs z : sustitucionLstR xs zs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión, la función sublista tal que
-- (sublista xs ys) se verifica si todos los elementos de xs aparecen en
-- ys en el mismo orden aunque no necesariamente consecutivos. Por ejemplo,
--    sublista "meta"   "matematicas"  ==  True
--    sublista "temas"  "matematicas"  ==  True
--    sublista "mitica" "matematicas"  ==  False
-- ---------------------------------------------------------------------

sublista []     ys = True
sublista (x:xs) [] = False
sublista (x:xs) (y:ys)
    | x == y           = sublista xs ys
    | otherwise        = sublista (x:xs) ys

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por recursión, la función numeroDigitosPares
-- tal que (numeroDigitosPares n) es la cantidad de dígitos pares que
-- hay en el número natural n. Por ejemplo,
--    numeroDigitosPares      0  ==  1
--    numeroDigitosPares      1  ==  0
--    numeroDigitosPares    246  ==  3
--    numeroDigitosPares    135  ==  0
--    numeroDigitosPares 123456  ==  3
-- ---------------------------------------------------------------------


numeroDigitosPares2 n
    | n < 10    = aux n
    | otherwise = (aux n `rem` 10) + numeroDigitosPares2 (n `div` 10)
    where aux n | even n    = 1
```

```
                | otherwise = 0
```

4.3. Examen 3 (23 de Enero de 2014)

El examen es común con el del grupo 1 (ver página 79).

4.4. Examen 4 (20 de Marzo de 2014)

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

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

```
--      3     2
--     / \   / \
--    5   4 6   7
-- son: [1], [3,2] y [5,4,6,7]
--
-- Definir la función
--    capas :: Arbol -> [[Int]]
-- tal que (capas a) es la lista de las capas de dicho árbol ordenadas
-- según la profunidad. Por ejemplo,
--    capas ej1  ==  [[1],[3,2],[5,4,6,7]]
--    capas ej2  ==  [[3],[1,4]]
--    capas ej3  ==  [[2],[3,5]]
--    capas ej4  ==  [[1],[2,2],[3,3]]
--    capas ej5  ==  [[1],[2,2],[3,5]]
-- ---------------------------------------------------------------------

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

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

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

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Un árbol es subárbol de otro si se puede establecer una
-- correspondencia de los nodos del primero con otros mayores o iguales
-- en el segundo, de forma que se respeten las relaciones de
-- descendencia. Este concepto se resume en varias situaciones posibles:
-- * El primer árbol es subárbol del hijo izquierdo del segundo
--   árbol. De esta forma ej2 es subárbol de ej1.
-- * El primer árbol es subárbol del hijo derecho del segundo árbol. De
--   esta forma ej3 es subárbol de ej1.
-- * La raíz del primer árbol es menor o igual que la del segundo, el
--   hijo izquierdo del primer árbol es subárbol del hijo izquierdo del
--   segundo y el hijo derecho del primer árbol es subárbol del hijo
--   derecho del segundo. De esta forma ej4 es subárbol de ej1.
--
-- Definir la función
--    subarbol :: Arbol -> Arbol -> Bool
-- tal que (subarbol a1 a2) se verifica si a1 es subárbol de a2. Por
-- ejemplo,
--    subarbol ej2 ej1  ==  True
--    subarbol ej3 ej1  ==  True
--    subarbol ej4 ej1  ==  True
--    subarbol ej5 ej1  ==  False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1 (1.2 puntos): Definir la función
--    intercalaRep :: Eq a => a -> [a] -> [[a]]
-- tal que (intercalaRep x ys), es la lista de las listas obtenidas
```

```haskell
-- intercalando x entre los elementos de ys, hasta la primera ocurrencia
-- del elemento x en ys. Por ejemplo,
--    intercalaRep 1 []       ==  [[1]]
--    intercalaRep 1 [1]      ==  [[1,1]]
--    intercalaRep 1 [2]      ==  [[1,2],[2,1]]
--    intercalaRep 1 [1,1]    ==  [[1,1,1]]
--    intercalaRep 1 [1,2]    ==  [[1,1,2]]
--    intercalaRep 1 [2,1]    ==  [[1,2,1],[2,1,1]]
--    intercalaRep 1 [1,2,1]  ==  [[1,1,2,1]]
--    intercalaRep 1 [2,1,1]  ==  [[1,2,1,1],[2,1,1,1]]
--    intercalaRep 1 [1,1,2]  ==  [[1,1,1,2]]
--    intercalaRep 1 [1,2,2]  ==  [[1,1,2,2]]
--    intercalaRep 1 [2,1,2]  ==  [[1,2,1,2],[2,1,1,2]]
--    intercalaRep 1 [2,2,1]  ==  [[1,2,2,1],[2,1,2,1],[2,2,1,1]]
-- ------------------------------------------------------------------

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

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

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

```
--    permutacionesRep [1,2,1]    ==  [[1,2,1],[2,1,1],[1,1,2]]
--    permutacionesRep [1,1,2]    ==  [[1,1,2],[1,2,1],[2,1,1]]
--    permutacionesRep [2,1,1]    ==  [[2,1,1],[1,2,1],[1,1,2]]
--    permutacionesRep [1,1,1]    ==  [[1,1,1]]
--    permutacionesRep [1,1,2,2]  ==  [[1,1,2,2],[1,2,1,2],[2,1,1,2],
--                                     [1,2,2,1],[2,1,2,1],[2,2,1,1]]
-- ----------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

4.5. Examen 5 (22 de Mayo de 2014)

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

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

import Data.Array
import GrafoConListas
import Data.List

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
--    ( 0 0 9 1 2 0 )
--    ( 0 0 0 3 4 5 )
--    ( 0 0 0 0 6 7 )
--
-- Definir la función
--    creaTridiagonal :: Int -> Matriz Int
-- tal que (creaTridiagonal n) es la siguiente matriz tridiagonal
-- cuadrada con n filas y n columnas:
--    ( 1 1 0 0 0 0 ... 0  0  )
--    ( 1 2 2 0 0 0 ... 0  0  )
--    ( 0 2 3 3 0 0 ... 0  0  )
--    ( 0 0 3 4 4 0 ... 0  0  )
--    ( 0 0 0 4 5 5 ... 0  0  )
--    ( 0 0 0 0 5 6 ... 0  0  )
--    ( ..................... )
--    ( 0 0 0 0 0 0 ... n  n  )
--    ( 0 0 0 0 0 0 ... n n+1 )
-- Por ejemplo,
--    ghci> creaTridiagonal 4
--    array ((1,1),(4,4)) [((1,1),1),((1,2),1),((1,3),0),((1,4),0),
--                         ((2,1),1),((2,2),2),((2,3),2),((2,4),0),
--                         ((3,1),0),((3,2),2),((3,3),3),((3,4),3),
--                         ((4,1),0),((4,2),0),((4,3),3),((4,4),4)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

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

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

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

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

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

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

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

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

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

4.6. Examen 6 (18 de Junio de 2014)

El examen es común con el del grupo 1 (ver página 96).

4.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 38).

4.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 45).

4.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 49).

5

Exámenes del grupo 5

Andrés Cordón y Miguel A. Martínez

5.1. Examen 1 (5 de Noviembre de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas y en Física)
-- 1º examen de evaluación continua (4 de noviembre de 2013)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un año es bisiesto si, o bien es divisible por 4 pero no
-- por 100, o bien es divisible por 400. En cualquier otro caso, no lo
-- es.
--
-- Definir el predicado
--    bisiesto :: Int -> Bool
-- tal que (bisiesto a) se verifica si a es un año bisiesto. Por ejemplo:
--    bisiesto 2013 == False        bisiesto 2012 == True
--    bisiesto 1700 == False        bisiesto 1600 == True
-- ---------------------------------------------------------------------

-- 1ª definición:
bisiesto :: Int -> Bool
bisiesto x = (mod x 4 == 0 && mod x 100 /= 0) || mod x 400 == 0

-- 2ª definición (con guardas):
bisiesto2 :: Int -> Bool
bisiesto2 x | mod x 4 == 0 && mod x 100 /= 0 = True
            | mod x 400 == 0                 = True
```

```
            | otherwise                    = False

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    entre :: Int -> Int -> [Int]
-- tal que (entre a b) devuelve la lista de todos los bisiestos entre
-- los años a y b. Por ejemplo:
--    entre 2000 2019 == [2000,2004,2008,2012,2016]
-- ---------------------------------------------------------------------

entre :: Int -> Int -> [Int]
entre a b = [x | x <- [a..b], bisiesto x]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir el predicado
--    coprimos :: (Int,Int) -> Bool
-- tal que (coprimos (a,b)) se verifica si a y b son primos entre sí;
-- es decir, no tienen ningún factor primo en común. Por ejemplo,
--    coprimos (12,25) == True
--    coprimos (6,21)  == False
--    coprimos (1,5)   == True
-- Calcular todos los números de dos cifras coprimos con 30.
-- ---------------------------------------------------------------------

-- 1ª definición
coprimos :: (Int,Int) -> Bool
coprimos (a,b) = and [mod a x /= 0 | x <- factores b]
    where factores x = [z | z <- [2..x], mod x z == 0]

-- 2º definición
coprimos2 :: (Int,Int) -> Bool
coprimos2 (a,b) = gcd a b == 1

-- El cálculo es
--    ghci> [x | x <- [10..99], coprimos (x,30)]
--    [11,13,17,19,23,29,31,37,41,43,47,49,53,59,61,67,71,73,77,79,83,89,91,97]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    longCamino :: [(Float,Float)] -> Float
```

```haskell
-- tal que (longCamino xs) es la longitud del camino determinado por los
-- puntos del plano listados en xs. Por ejemplo,
--    longCamino [(0,0),(1,0),(2,1),(2,0)] == 3.4142137
-- ---------------------------------------------------------------------

longCamino :: [(Float,Float)] -> Float
longCamino xs =
    sum [sqrt ((a-c)^2+(b-d)^2)| ((a,b),(c,d)) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Se quiere poner en marcha un nuevo servicio de correo
-- electrónico. Se requieren las siguientes condiciones para las
-- contraseñas: deben contener un mínimo de 8 caracteres, al menos deben
-- contener dos números, y al menos deben contener una letra
-- mayúscula. Se asume que el resto de caracteres son letras del
-- abecedario sin tildes.
--
-- Definir la función
--    claveValida :: String -> Bool
-- tal que (claveValida xs) indica si la contraseña es válida. Por
-- ejemplo,
--    claveValida "EstoNoVale" == False
--    claveValida "Tampoco7"   == False
--    claveValida "SiVale23"   == True
-- ---------------------------------------------------------------------

claveValida :: String -> Bool
claveValida xs =
    length xs >= 8 &&
    length [x | x <- xs, x `elem` ['0'..'9']] > 1 &&
    [x | x <- xs, x `elem` ['A'..'Z']] /= []

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    media :: [String] -> Float
-- tal que (media xs) es la media de las longitudes de las contraseñas
-- válidas de xs. Por ejemplo,
--    media ["EstoNoVale","Tampoco7","SiVale23","grAnada1982"] == 9.5
-- Indicación: Usar fromIntegral.
-- ---------------------------------------------------------------------
```

```haskell
media :: [String] -> Float
media xss =
    fromIntegral (sum [length xs | xs <- validas]) / fromIntegral (length validas
    where validas = [xs | xs <- xss, claveValida xs]
```

5.2. Examen 2 (16 de Diciembre de 2013)

```haskell
-- Informática (1º del Grado en Matemáticas y en Física)
-- 2º examen de evaluación continua (16 de diciembre de 2013)
-- ---------------------------------------------------------------------

import Data.Char

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir las funciones
--    ultima, primera :: Int -> Int
-- que devuelven, respectivamente, la última y la primera cifra de un
-- entero positivo, Por ejemplo:
--    ultima  711 = 1
--    primera 711 = 7
-- ---------------------------------------------------------------------

ultima, primera :: Int -> Int
ultima n  = n `rem` 10
primera n = read [head (show n)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, el predicado
--    encadenadoR :: [Int] -> Bool
-- tal que (encadenadoR xs) se verifica si xs es una lista de
-- enteros positivos encadenados (es decir, la última cifra de cada
-- número coincide con la primera del siguiente en la lista). Por ejemplo:
--   encadenadoR [711,1024,413,367]  ==  True
--   encadenadoR [711,1024,213,367]  ==  False
-- ---------------------------------------------------------------------

encadenadoR :: [Int] -> Bool
encadenadoR (x:y:zs) = ultima x == primera y && encadenadoR (y:zs)
```

```
encadenadoR _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por comprensión, el predicado
--    encadenadoC :: [Int] -> Bool
-- tal que (encadenadoC xs) se verifica si xs es una lista de
-- enteros positivos encadenados (es decir, la última cifra de cada
-- número coincide con la primera del siguiente en la lista). Por ejemplo:
--   encadenadoC [711,1024,413,367]  ==  True
--   encadenadoC [711,1024,213,367]  ==  False
-- ---------------------------------------------------------------------

encadenadoC :: [Int] -> Bool
encadenadoC xs = and [ultima x == primera y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Un entero positivo se dirá semiperfecto si puede
-- obtenerse como la suma de un subconjunto de sus divisores propios (no
-- necesariamente todos). Por ejemplo, 18 es semiperfecto, pues sus
-- divisores propios son 1, 2, 3, 6 y 9 y además 18 = 1+2+6+9.
--
-- Define el predicado
--    semiperfecto :: Int -> Bool
-- tal que (semiperfecto x) se verifica si x es semiperfecto. Por
-- ejemplo,
--   semiperfecto 18  ==  True
--   semiperfecto 15  ==  False
-- ---------------------------------------------------------------------

semiperfecto :: Int -> Bool
semiperfecto n = not (null (sublistasConSuma (divisores n) n))

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

-- (sublistasConSuma xs n) es la lista de las sublistas de la lista de
-- números naturales xs que suman n. Por ejemplo,
--   sublistasConSuma [1,2,3,6,9] 18  ==  [[1,2,6,9],[3,6,9]]
```

```haskell
sublistasConSuma :: [Int] -> Int -> [[Int]]
sublistasConSuma [] 0 = [[]]
sublistasConSuma [] _ = []
sublistasConSuma (x:xs) n
    | x > n = sublistasConSuma xs n
    | otherwise = [x:ys | ys <- sublistasConSuma xs (n-x)] ++
                  sublistasConSuma xs n

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    semiperfectos :: Int -> [Int]
-- tal que (semiperfectos n) es la lista de los n primeros números
-- semiperfectos. Por ejemplo:
--    semiperfectos 10 == [6,12,18,20,24,28,30,36,40,42]
-- ---------------------------------------------------------------------

semiperfectos :: Int -> [Int]
semiperfectos n = take n [x | x <- [1..], semiperfecto x]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Formatear una cadena de texto consiste en:
--    1. Eliminar los espacios en blanco iniciales.
--    2. Eliminar los espacios en blanco finales.
--    3. Reducir a 1 los espacios en blanco entre palabras.
--    4. Escribir la primera letra en mayúsculas, si no lo estuviera.
--
-- Definir la función
--    formateada :: String -> String
-- tal que (formateada cs) es la cadena cs formateada. Por ejemplo,
--    formateada "   la   palabra   precisa "  ==  "La palabra precisa"
-- ---------------------------------------------------------------------

formateada :: String -> String
formateada cs = toUpper x : xs
    where (x:xs) = unwords (words cs)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El centro de masas de un sistema discreto es el punto
-- geométrico que dinámicamente se comporta como si en él estuviera
-- aplicada la resultante de las fuerzas externas al sistema.
```

```haskell
--
-- Representamos un conjunto de n masas en el plano mediante una lista
-- de n pares de la forma ((ai,bi),mi) donde (ai,bi) es la posición y mi
-- la masa puntual. Las coordenadas del centro de masas (a,b) se
-- calculan por
--    a = (a1*m1+a2*m2+ ... an*mn)/(m1+m2+...mn)
--    b = (b1*m1+b2*m2+ ... bn*mn)/(m1+m2+...mn)
--
-- Definir la función
--    masaTotal ::  [((Float,Float),Float)] -> Float
-- tal que (masaTotal xs) es la masa total de sistema xs. Por ejemplo,
--    masaTotal  [((-1,3),2),((0,0),5),((1,3),3)]  ==  10
-- ---------------------------------------------------------------------

masaTotal ::  [((Float,Float),Float)] -> Float
masaTotal xs = sum [m | (_,m) <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    centrodeMasas :: [((Float,Float),Float)] -> (Float,Float)
-- tal que (centrodeMasas xs) es las coordenadas del centro
-- de masas del sistema discreto xs. Por ejemplo:
--    centrodeMasas [((-1,3),2),((0,0),5),((1,3),3)] == (0.1,1.5)
-- ---------------------------------------------------------------------

centrodeMasas :: [((Float,Float),Float)] -> (Float,Float)
centrodeMasas xs =
    (sum [a*m | ((a,_),m) <- xs] / mt,
     sum [b*m | ((_,b),m) <- xs] / mt)
    where mt = masaTotal xs
```

5.3. Examen 3 (23 de Enero de 2014)

El examen es común con el del grupo 1 (ver página 79).

5.4. Examen 4 (19 de Marzo de 2014)

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

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

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

import Data.List

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

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

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

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

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

```haskell
-- 13333.
-- ---------------------------------------------------------------------

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

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

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

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

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

```haskell
-- 1ª definición (por comprensión)
expandida :: [(Int,a)] -> [a]
expandida ps = concat [replicate k x | (k,x) <- ps]

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

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

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

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

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

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

```haskell
-- Ejercicio 3.2. Definir la función
--    todosDistintos :: Eq a => Arbol a -> Bool
-- tal que (todosDistintos a) se verifica si todos los elementos del
-- árbol a son distintos entre sí. Por ejemplo,
--    todosDistintos ejArbol                                          == True
--    todosDistintos (Nodo 7 (Hoja 3) (Nodo 4 (Hoja 7) (Hoja 2)))  ==  False
-- ----------------------------------------------------------------------

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

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

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

```haskell
-- Se puede observar que es posible crear más de una trayectoria para la
-- misma  partícula.
--
-- Definir la función
--    calculaTrayectorias :: [[Int]] -> [(Int,[Int])]
-- que devuelva una lista con todas las trayectorias posibles para un
-- experimento. Por ejemplo,
--    ghci> calculaTrayectorias experimento
--    [(2,[-1,-1,1,-1]),  (2,[-1,1,-1,-1]), (2,[1,-1,-1,-1]),
--     (3,[0,-1,-1,-1]),
--     (5,[0,1,-1,-1]),   (5,[0,1,1,1]),
--     (8,[-1,-1,-1,-1]), (8,[-1,-1,1,1])]
-- ----------------------------------------------------------------------

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

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

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

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

posiblesTrayectoriasRecAcum i yss [] = yss
posiblesTrayectoriasRecAcum i yss (xs:xss) =
```

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

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

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

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

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

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

5.5. Examen 5 (21 de Mayo de 2014)

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

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

import Data.Array

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

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

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

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Representamos árboles binarios mediante el tipo de
-- dato
--    data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
-- Por ejemplo, los árboles
--        5           5
--       / \         / \
--      5   3       5   3
--     / \         / \
--    2   5       2   5
-- se representan por
--    arbol1, arbol2 :: Arbol Int
--    arbol1 = (Nodo 5 (Nodo 5 (Hoja 2) (Hoja 5)) (Hoja 3))
--    arbol2 = (Nodo 5 (Nodo 5 (Hoja 2) (Hoja 1)) (Hoja 3))
--
-- Definir el predicado
--    ramaIgual :: Eq a => Arbol a -> Bool
-- tal que (ramaIgual t) se verifica si el árbol t contiene, al menos,
-- una rama con todos sus elementos iguales. Por ejemplo,
--    ramaIgual arbol1 == True
--    ramaIgual arbol2 == False
-- ---------------------------------------------------------------------

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

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

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

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

-- (todosI xs) se verifica si todos los elementos de xs son iguales. Por
```

```haskell
-- ejemplo,
--    todosI [6,6,6]  ==  True
--    todosI [6,9,6]  ==  False
todosI :: Eq a => [a] -> Bool
todosI (x:y:xs) = x == y && todosI (y:xs)
todosI _        = True

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

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

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

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

pol :: Pol Int
```

```haskell
pol = Cons 5 3 (Cons 3 (-1) (Cons 2 7 PolCero))

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

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

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

```
--                                          2, 0, 0, 0, 2,
--                                          2, 2, 2, 2, 2])
--    matriz2 = listArray ((1,1),(5,5)) ([2.0, 2.0, 2.0, 2.0, 2.0,
--                                          2.0, 0.8, 0.4, 0.8, 2.0,
--                                          2.0, 0.4, 0.0, 0.4, 2.0,
--                                          2.0, 0.8, 0.4, 0.8, 2.0,
--                                          2.0, 2.0, 2.0, 2.0, 2.0])
--
-- Definir la función
--    iteracion_jacobi:: Matriz -> Matriz
-- tal que (iteracion_jacobi p) es la matriz obtenida aplicándole una
-- transformación de Jacobi a la matriz p. Por ejemplo,
--    iteracion_jacobi matriz1  ==  matriz2
-- ----------------------------------------------------------------------

type Matriz = Array (Int,Int) Float

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

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

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

-- 2ª definición:
```

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

5.6. Examen 6 (18 de Junio de 2014)

El examen es común con el del grupo 1 (ver página 96).

5.7. Examen 7 (4 de Julio de 2014)

El examen es común con el del grupo 3 (ver página 38).

5.8. Examen 8 (10 de Septiembre de 2014)

El examen es común con el del grupo 3 (ver página 45).

5.9. Examen 9 (20 de Noviembre de 2014)

El examen es común con el del grupo 3 (ver página 49).

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.