Exámenes de
“Programación funcional con Haskell”

Vol. 10 (Curso 2018–19)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 14 de septiembre de 2019

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 2018–19) 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 2018–19) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2018–19) [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 10º 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-18/temas/2018-19-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-18/ejercicios/ejercicios-I1M-2018.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol10
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

- Exámenes de “Programación funcional con Haskell”. Vol. 2 (Curso 2010–11) [6]
- Exámenes de “Programación funcional con Haskell”. Vol. 3 (Curso 2011–12) [7]
- Exámenes de “Programación funcional con Haskell”. Vol. 4 (Curso 2012–13) [8]
- Exámenes de “Programación funcional con Haskell”. Vol. 5 (Curso 2013–14) [9]
- Exámenes de “Programación funcional con Haskell”. Vol. 6 (Curso 2014–15) [10]
- Exámenes de “Programación funcional con Haskell”. Vol. 6 (Curso 2015–16) [11]
- Exámenes de “Programación funcional con Haskell”. Vol. 8 (Curso 2016–17) [12]
- Exámenes de “Programación funcional con Haskell”. Vol. 9 (Curso 2017–18) [13]

José A. Alonso
Sevilla, 14 de septiembre de 2019

[6] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol2
[7] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol3
[8] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol4
[9] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol5
[10] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol6
[11] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol7
[12] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol8
[13] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol9

1

Exámenes del grupo 1

José A. Alonso

1.1. Examen 1 (07 de noviembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (7 de noviembre de 2018)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. 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. Definir la función
--    esPotencia :: Integer -> Integer -> Bool
-- tal que (esPotencia x a) se verifica si x es una potencia de a. Por
-- ejemplo,
--    esPotencia 32 2  ==  True
--    esPotencia 42 2  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición
esPotencia :: Integer -> Integer -> Bool
esPotencia x a = x `elem` [a^n | n <- [0..x]]

-- 2ª definición
esPotencia2 :: Integer -> Integer -> Bool
esPotencia2 x a = aux x a 0
    where aux x a b | b > x     = False
                    | otherwise = x == a ^ b || aux x a (b+1)

-- 3ª definición
esPotencia3 :: Integer -> Integer -> Bool
esPotencia3 0 _ = False
esPotencia3 1 a = True
esPotencia3 _ 1 = False
esPotencia3 x a = rem x a == 0 && esPotencia3 (div x a) a

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Una forma de aproximar el valor del número e es usando
-- la siguiente igualdad:
--
--                 1        2        3        4        5
--           e = ------ + ------ + ------ + ------ + ------ + ...
--                2*0!     2*1!     2*2!     2*3!     2*4!
--
--
-- Definir la función
--    aproximaE :: Double -> Double
-- tal que (aproximaE n) es la aproximación del número e calculada con
-- la serie anterior hasta el término n-ésimo. Por ejemplo,
--    aproximaE 10  ==  2.718281663359788
--    aproximaE 15  ==  2.718281828458612
--    aproximaE 20  ==  2.718281828459045
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

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

-- 2ª definición
-- =============

aproximaE2 :: Double -> Double
aproximaE2 0 = 1/2
aproximaE2 n = (n+1)/(2 * factorial n) + aproximaE2 (n-1)
  where factorial i = product [1..i]
```

1.2. Examen 2 (19 de diciembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (19 de diciembre de 2018)
-- ---------------------------------------------------------------------

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

import Data.Char
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Se dice que un número natural n es una colina si su
-- primer dígito es igual a su último dígito, los primeros dígitos son
-- estrictamente creciente hasta llegar al máximo, el máximo se puede
-- repetir y los dígitos desde el máximo al final son estrictamente
-- decrecientes.
--
-- Definir la función
--    esColina :: Integer -> Bool
-- tal que (esColina n) se verifica si n es un número colina. Por
-- ejemplo,
--    esColina 12377731  ==  True
--    esColina 1237731   ==  True
--    esColina 123731    ==  True
--    esColina 122731    ==  False
--    esColina 12377730  ==  False
--    esColina 12374731  ==  False
--    esColina 12377730  ==  False
--    esColina 10377731  ==  False
--    esColina 12377701  ==  False
--    esColina 33333333  ==  True
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

esColina :: Integer -> Bool
```

```haskell
esColina n =
  head ds == last ds &&
  esCreciente xs &&
  esDecreciente ys
  where ds = digitos n
        m  = maximum ds
        xs = takeWhile (<m) ds
        ys = dropWhile (==m) (dropWhile (<m) ds)

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 425  ==  [4,2,5]
digitos :: Integer -> [Int]
digitos n = map digitToInt (show n)

-- (esCreciente xs) se verifica si la lista xs es estrictamente
-- creciente. Por ejemplo,
--    esCreciente [2,4,7]  ==  True
--    esCreciente [2,2,7]  ==  False
--    esCreciente [2,1,7]  ==  False
esCreciente :: [Int] -> Bool
esCreciente xs = and [x < y | (x,y) <- zip xs (tail xs)]

-- (esDecreciente xs) se verifica si la lista xs es estrictamente
-- decreciente. Por ejemplo,
--    esDecreciente [7,4,2]  ==  True
--    esDecreciente [7,2,2]  ==  False
--    esDecreciente [7,1,2]  ==  False
esDecreciente :: [Int] -> Bool
esDecreciente xs = and [x > y | (x,y) <- zip xs (tail xs)]

-- 2ª definición
-- =============

esColina2 :: Integer -> Bool
esColina2 n =
  head ds == last ds &&
  null (dropWhile (==(-1)) (dropWhile (==0) (dropWhile (==1) xs)))
  where ds = digitos n
        xs = [signum (y-x) | (x,y) <- zip ds (tail ds)]
```

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

-- La propiedad de equivalencia es
prop_esColina :: Integer -> Property
prop_esColina n =
  n >= 0 ==> esColina n == esColina2 n

-- La comprobación es
--    λ> quickCheck prop_esColina
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. La persistencia multiplicativa de un número  es la
-- cantidad de pasos requeridos para reducirlo a un dígito multiplicando
-- sus dígitos. Por ejemplo, la persistencia de 39 es 3 porque 3*9 = 27,
-- 2*7 = 14 y 1*4 = 4.
--
-- Definir la función
--    persistencia     :: Integer -> Integer
-- tal que (persistencia x) es la persistencia de x. Por ejemplo,
--    persistencia 39                               ==   3
--    persistencia 2677889                          ==   8
--    persistencia 26888999                         ==   9
--    persistencia 3778888999                       ==  10
--    persistencia 277777788888899                  ==  11
--    persistencia 77777733332222222222222222222    ==  11
-- ---------------------------------------------------------------------

persistencia :: Integer -> Integer
persistencia x
  | x < 10    = 0
  | otherwise = 1 + persistencia (productoDigitos x)

productoDigitos :: Integer -> Integer
productoDigitos x
  | x < 10    = x
  | otherwise = r * productoDigitos y
  where (y,r) = quotRem x 10
```

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

-- 1ª solución
producto :: [[a]] -> [[a]]
producto []       = [[]]
producto (xs:xss) = [x:ys | x <- xs, ys <- producto xss]

-- 2ª solución
producto2 :: [[a]] -> [[a]]
producto2 = foldr f [[]]
  where f xs xss = [x:ys | x <- xs, ys <- xss]

-- 3ª solución
producto3 :: [[a]] -> [[a]]
producto3 = foldr aux [[]]
  where aux [] _      = []
aux (x:xs) ys = map (x:) ys ++ aux xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las expresiones aritméticas. generales se contruyen con
-- las sumas generales (sumatorios) y productos generales
-- (productorios). Su tipo es
--    data Expresion = N Int
--                   | S [Expresion]
--                   | P [Expresion]
--      deriving Show
-- Por ejemplo, la expresión (2 * (1 + 2 + 1) * (2 + 3)) + 1 se
```

```haskell
-- representa por S [P [N 2, S [N 1, N 2, N 1], S [N 2, N 3]], N 1]
--
-- Definir la función
--    valor :: Expresion -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--    λ> valor (S [P [N 2, S [N 1, N 2, N 1], S [N 2, N 3]], N 1])
--    41
-- ---------------------------------------------------------------------

data Expresion = N Int
               | S [Expresion]
               | P [Expresion]
  deriving Show

valor :: Expresion -> Int
valor (N x)  = x
valor (S es) = sum (map valor es)
valor (P es) = product (map valor es)
```

1.3. Examen 3 (22 de enero de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 4 y 5)
-- 3º examen de evaluación continua (22 de enero de 2019)
-- ---------------------------------------------------------------------

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

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    nParejas :: Ord a => [a] -> Int
-- tal que (nParejas xs) es el número de parejas de elementos iguales en
-- xs. Por rjemplo,
--    nParejas [1,2,2,1,1,3,5,1,2]         ==  3
--    nParejas [1,2,1,2,1,3,2]             ==  2
--    nParejas [1..2*10^6]                 ==  0
--    nParejas2 ([1..10^6] ++ [1..10^6])   ==  1000000
```

```haskell
-- En el primer ejemplos las parejas son (1,1), (1,1) y (2,2). En el
-- segundo ejemplo, las parejas son (1,1) y (2,2).
-- ---------------------------------------------------------------------

-- 1ª solución
nParejas :: Ord a => [a] -> Int
nParejas []     = 0
nParejas (x:xs) | x `elem` xs = 1 + nParejas (xs \\ [x])
                | otherwise   = nParejas xs

-- 2ª solución
nParejas2 :: Ord a => [a] -> Int
nParejas2 xs =
  sum [length ys `div` 2 | ys <- group (sort xs)]

-- 3ª solución
nParejas3 :: Ord a => [a] -> Int
nParejas3 = sum . map (`div` 2) . map length . group . sort

-- 4ª solución
nParejas4 :: Ord a => [a] -> Int
nParejas4 = sum . map ((`div` 2) . length) . group . sort

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    maximos :: Ord a => [a] -> [a]
-- tal que (maximos xs) es la lista de los elementos de xs que son
-- mayores que todos sus anteriores. Por ejemplo,
--    maximos [1,-3,5,2,3,4,7,6,7]                         ==  [1,5,7]
--    maximos "bafcdegag"                                  ==  "bfg"
--    maximos (concat (replicate (10^6) "adxbcde")++"yz")  ==  "adxyz"
--    length (maximos [1..10^6])                           ==  1000000
-- ---------------------------------------------------------------------

-- 1ª solución
maximos :: Ord a => [a] -> [a]
maximos xs =
  [x | (ys,x) <- zip (inits xs) xs, all (<x) ys]

-- 2ª solución
```

```haskell
maximos2 :: Ord a => [a] -> [a]
maximos2 [] = []
maximos2 (x:xs) = x : maximos2 (filter (>x) xs)

-- 3ª solución
maximos3 :: Ord a => [a] -> [a]
maximos3 [] = []
maximos3 (x:xs) = aux xs [x] x
    where aux [] zs _ = reverse zs
          aux (y:ys) zs m | y > m     = aux ys (y:zs) y
                          | otherwise = aux ys zs m

-- 4ª solución
maximos4 :: Ord a => [a] -> [a]
maximos4 = nub . scanl1 max

-- ---------------------------------------------------------------------
-- Ejercicio 3. La sucesión de los primeros factoriales es
--    1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, ...
-- El factorial más cercano a un número x es el menor elemento y de la
-- sucesión de los factoriales tal que el valor absoluto de la
-- diferencia entre x e y es la menor posible. Por ejemplo,
-- + el factorial más cercano a 3 es 2 porque |3-2| < |3-6|
-- + el factorial más cercano a 4 es 2 porque |4-2| = |4-6| y 2 <
-- + el factorial más cercano a 5 es 6 porque |5-2| > |5-6|
-- + el factorial más cercano a 5 es 6 porque 6 es un factorial.
--
-- Definir la función
--    factorialMasCercano :: Integer -> Integer
-- tal que (factorialMasCercano n) es el factorial más cercano a n. Por
-- ejemplo,
--    factorialMasCercano 3  ==  2
--    factorialMasCercano 4  ==  2
--    factorialMasCercano 5  ==  6
--    factorialMasCercano 6  ==  6
--    factorialMasCercano 2019  ==  720
-- ---------------------------------------------------------------------

factorialMasCercano :: Integer -> Integer
factorialMasCercano n
```

```haskell
  | b == n                 = n
  | abs (n-a) <= abs (n-b) = a
  | otherwise              = b
  where (xs,b:ys) = span (<n) factoriales
        a = last xs

factoriales :: [Integer]
factoriales = scanl1 (*) [1..]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las expresiones aritméticas. generales se contruyen con
-- las sumas generales (sumatorios) y productos generales (productorios).
-- Su tipo es
--    data Expresion = N Int
--                   | S [Expresion]
--                   | P [Expresion]
--      deriving Show
-- Por ejemplo, la expresión (2 * (1 + 2 + 1) * (2 + 3)) + 1 se
-- representa por S [P [N 2, S [N 1, N 2, N 1], S [N 2, N 3]], N 1]
--
-- Definir la función
--    valor :: Expresion -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--    λ> valor (S [P [N 2, S [N 1, N 2, N 1], S [N 2, N 3]], N 1])
--    41
-- ---------------------------------------------------------------------

data Expresion = N Int
               | S [Expresion]
               | P [Expresion]
  deriving Show

-- 1ª solución
valor :: Expresion -> Int
valor (N x)  = x
valor (S es) = sum (map valor es)
valor (P es) = product (map valor es)

-- 2ª solución
valor2 :: Expresion -> Int
```

```haskell
valor2 (N x)  = x
valor2 (S es) = sum [valor2 e | e <- es]
valor2 (P es) = product [valor2 e | e <- es]
```

1.4. Examen 4 (13 de marzo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 4º examen de evaluación continua (13 de marzo de 2019)
-- ---------------------------------------------------------------------

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

import Data.Array
import Test.QuickCheck
import Text.Printf
import Data.List
import System.Timeout

-- ---------------------------------------------------------------------
-- Ejercicio 1 [2.5 puntos] Los árboles binarios se pueden representar
-- con
--    data Arbol a = H a
--                 | Nodo a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
--    arboles  :: Integer -> a -> [Arbol a]
-- tales que (arboles n x) es la lista de todos los árboles binarios con
-- n elementos iguales a x. Por ejemplo,
--    λ> arboles 0 7
--    []
--    λ> arboles 1 7
--    [H 7]
--    λ> arboles 2 7
--    []
--    λ> arboles 3 7
--    [Nodo 7 (H 7) (H 7)]
```

```haskell
--    λ> arboles 4 7
--    []
--    λ> arboles 5 7
--    [Nodo 7 (H 7) (Nodo 7 (H 7) (H 7)),
--     Nodo 7 (Nodo 7 (H 7) (H 7)) (H 7)]
--    λ> arboles 6 7
--    []
--    λ> arboles 7 7
--    [Nodo 7 (H 7) (Nodo 7 (H 7) (Nodo 7 (H 7) (H 7))),
--     Nodo 7 (H 7) (Nodo 7 (Nodo 7 (H 7) (H 7)) (H 7)),
--     Nodo 7 (Nodo 7 (H 7) (H 7)) (Nodo 7 (H 7) (H 7)),
--     Nodo 7 (Nodo 7 (H 7) (Nodo 7 (H 7) (H 7))) (H 7),
--     Nodo 7 (Nodo 7 (Nodo 7 (H 7) (H 7)) (H 7)) (H 7)]
-- ---------------------------------------------------------------------

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

-- 1ª definición
-- =============

arboles :: Integer -> a -> [Arbol a]
arboles 0 _ = []
arboles 1 x = [H x]
arboles n x = [Nodo x i d | k <- [0..n-1],
                            i <- arboles k x,
                            d <- arboles (n-1-k) x]

-- 2ª definición
-- =============

arboles2 :: Integer -> a -> [Arbol a]
arboles2 0 _ = []
arboles2 1 x = [H x]
arboles2 n x = [Nodo x i d | k <- [1,3..n-1],
                             i <- arboles2 k x,
                             d <- arboles2 (n-1-k) x]

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

```haskell
-- Ejercicio 2. [2.5 puntos]. Un camino es una sucesión de pasos en una
-- de las cuatros direcciones Norte, Sur, Este, Oeste. Ir en una
-- dirección y a continuación en la opuesta es un esfuerzo que se puede
-- reducir. Por ejemplo, el camino [Norte,Sur,Este,Sur] se puede reducir
-- a [Este,Sur].
--
-- Un camino se dice que es reducido si no tiene dos pasos consecutivos
-- en direcciones opuesta.
--
-- En Haskell, las direcciones y los caminos se pueden definir por
--    data Direccion = N | S | E | O deriving (Show, Eq)
--    type Camino = [Direccion]
--
-- Definir la función
--    reducido :: Camino -> Camino
-- tal que (reducido ds) es el camino reducido equivalente al camino
-- ds. Por ejemplo,
--    reducido []                              ==  []
--    reducido [N]                             ==  [N]
--    reducido [N,O]                           ==  [N,O]
--    reducido [N,O,E]                         ==  [N]
--    reducido [N,O,E,S]                       ==  []
--    reducido [N,O,S,E]                       ==  [N,O,S,E]
--    reducido [S,S,S,N,N,N]                   ==  []
--    reducido [N,S,S,E,O,N]                   ==  []
--    reducido [N,S,S,E,O,N,O]                 ==  [O]
--
-- Nótese que en el último ejemplo las reducciones son
--        [N,S,S,E,O,N,O]
--    --> [S,E,O,N,O]
--    --> [S,N,O]
--    --> [O]
-- ---------------------------------------------------------------------

data Direccion = N | S | E | O deriving (Show, Eq)

type Camino = [Direccion]

-- 1ª solución
-- ===========
```

```haskell
reducido :: Camino -> Camino
reducido [] = []
reducido (d:ds)
  | null ds'                 = [d]
  | d == opuesta (head ds') = tail ds'
  | otherwise                = d:ds'
  where ds' = reducido ds

opuesta :: Direccion -> Direccion
opuesta N = S
opuesta S = N
opuesta E = O
opuesta O = E

-- 2ª solución
-- ===========

reducido2 :: Camino -> Camino
reducido2 = foldr aux []
    where aux N (S:xs) = xs
          aux S (N:xs) = xs
          aux E (O:xs) = xs
          aux O (E:xs) = xs
          aux x xs     = x:xs

-- 3ª solución
-- ===========

reducido3 :: Camino -> Camino
reducido3 []       = []
reducido3 (N:S:ds) = reducido3 ds
reducido3 (S:N:ds) = reducido3 ds
reducido3 (E:O:ds) = reducido3 ds
reducido3 (O:E:ds) = reducido3 ds
reducido3 (d:ds)
  | null ds'                 = [d]
  | d == opuesta (head ds') = tail ds'
  | otherwise                = d:ds'
  where ds' = reducido3 ds
```

```haskell
-- 4ª solución
-- ===========

reducido4 :: Camino -> Camino
reducido4 ds = reverse (aux ([],ds)) where
    aux (N:xs, S:ys) = aux (xs,ys)
    aux (S:xs, N:ys) = aux (xs,ys)
    aux (E:xs, O:ys) = aux (xs,ys)
    aux (O:xs, E:ys) = aux (xs,ys)
    aux (  xs, y:ys) = aux (y:xs,ys)
    aux (  xs,   []) = xs

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

-- La comparación es
--    ghci> reducido (take (10^6) (cycle [N,E,O,S]))
--    []
--    (3.87 secs, 460160736 bytes)
--    ghci> reducido2 (take (10^6) (cycle [N,E,O,S]))
--    []
--    (1.16 secs, 216582880 bytes)
--    ghci> reducido3 (take (10^6) (cycle [N,E,O,S]))
--    []
--    (0.58 secs, 98561872 bytes)
--    ghci> reducido4 (take (10^6) (cycle [N,E,O,S]))
--    []
--    (0.64 secs, 176154640 bytes)
--
--    ghci> reducido3 (take (10^7) (cycle [N,E,O,S]))
--    []
--    (5.43 secs, 962694784 bytes)
--    ghci> reducido4 (take (10^7) (cycle [N,E,O,S]))
--    []
--    (9.29 secs, 1722601528 bytes)
--
--    ghci> length $ reducido3 (take 2000000 $ cycle [N,O,N,S,E,N,S,O,S,S])
--    400002
--    (4.52 secs, 547004960 bytes)
```

```haskell
--    ghci> length $ reducido4 (take 2000000 $ cycle [N,O,N,S,E,N,S,O,S,S])
--    400002
--
--    ghci> let n=10^6 in reducido (replicate n N ++ replicate n S)
--    []
--    (7.35 secs, 537797096 bytes)
--    ghci> let n=10^6 in reducido2 (replicate n N ++ replicate n S)
--    []
--    (2.30 secs, 244553404 bytes)
--    ghci> let n=10^6 in reducido3 (replicate n N ++ replicate n S)
--    []
--    (8.08 secs, 545043608 bytes)
--    ghci> let n=10^6 in reducido4 (replicate n N ++ replicate n S)
--    []
--    (1.96 secs, 205552240 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. [1.5 puntos] Una serie infinita para el cálculo de pi,
-- publicada por Nilakantha en el siglo XV, es
--                 4       4       4       4
--    pi = 3 + ----- - ----- + ----- - ------ + ···
--               2x3x4   4x5x6   6x7x8   8x9x10
--
-- Definir la función
--    aproximacionPi :: Int -> Double
-- tal que (aproximacionPi n) es la n-ésima aproximación de pi obtenida
-- sumando los n primeros términos de la serie de Nilakantha. Por
-- ejemplo,
--    aproximacionPi 0        ==  3.0
--    aproximacionPi 1        ==  3.1666666666666665
--    aproximacionPi 2        ==  3.1333333333333333
--    aproximacionPi 3        ==  3.145238095238095
--    aproximacionPi 4        ==  3.1396825396825396
--    aproximacionPi 5        ==  3.1427128427128426
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

aproximacionPi :: Int -> Double
```

```haskell
aproximacionPi n = serieNilakantha !! n

serieNilakantha :: [Double]
serieNilakantha = scanl1 (+) terminosNilakantha

terminosNilakantha :: [Double]
terminosNilakantha = zipWith (/) numeradores denominadores
  where numeradores   = 3 : cycle [4,-4]
        denominadores = 1 : [n*(n+1)*(n+2) | n <- [2,4..]]

-- 2ª solución
-- ===========

aproximacionPi2 :: Int -> Double
aproximacionPi2 = aux 3 2 1
  where aux x _ _ 0 = x
        aux x y z m =
          aux (x+4/product[y..y+2]*z) (y+2) (negate z) (m-1)

-- 3ª solución
-- ===========

aproximacionPi3 :: Int -> Double
aproximacionPi3 x =
  3 + sum [(((-1)**(n+1))*4)/(2*n*(2*n+1)*(2*n+2))
          | n <- [1..fromIntegral x]]

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

-- La comparación es
--    λ> aproximacionPi (10^6)
--    3.141592653589787
--    (1.35 secs, 729,373,160 bytes)
--    λ> aproximacionPi2 (10^6)
--    3.141592653589787
--    (2.96 secs, 2,161,766,096 bytes)
--    λ> aproximacionPi3 (10^6)
--    3.1415926535897913
--    (2.02 secs, 1,121,372,536 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.2 [1 punto]. Definir la función
--    tabla :: FilePath -> [Int] -> IO ()
-- tal que (tabla f ns) escribe en el fichero f las n-ésimas
-- aproximaciones de pi, donde n toma los valores de la lista ns, junto
-- con sus  errores. Por ejemplo, al evaluar la expresión
--    tabla "AproximacionesPi.txt" [0,10..100]
-- hace que el contenido del fichero "AproximacionesPi.txt" sea
--      +------+----------------+----------------+
--      | n    | Aproximación   | Error          |
--      +------+----------------+----------------+
--      |    0 | 3.000000000000 | 0.141592653590 |
--      |   10 | 3.141406718497 | 0.000185935093 |
--      |   20 | 3.141565734659 | 0.000026918931 |
--      |   30 | 3.141584272675 | 0.000008380915 |
--      |   40 | 3.141589028941 | 0.000003624649 |
--      |   50 | 3.141590769850 | 0.000001883740 |
--      |   60 | 3.141591552546 | 0.000001101044 |
--      |   70 | 3.141591955265 | 0.000000698325 |
--      |   80 | 3.141592183260 | 0.000000470330 |
--      |   90 | 3.141592321886 | 0.000000331704 |
--      |  100 | 3.141592410972 | 0.000000242618 |
--      +------+----------------+----------------+
-- ---------------------------------------------------------------------

tabla :: FilePath -> [Int] -> IO ()
tabla f ns = writeFile f (tablaAux ns)

tablaAux :: [Int] -> String
tablaAux ns =
     linea
  ++ cabecera
  ++ linea
  ++ concat [printf "| %4d | %.12f | %.12f |\n" n a e
            | n <- ns
            , let a = aproximacionPi n
            , let e = abs (pi - a)]
  ++ linea
```

```haskell
linea :: String
linea = "+------+---------------+---------------+\n"

cabecera :: String
cabecera = "| n    | Aproximación    | Error          |\n"

-- ---------------------------------------------------------------------
-- Ejercicio 4.1 [1 punto]. El número 7 tiene 4 descomposiciones como
-- suma de cuadrados de enteros positivos
--    7 = 1^2 + 1^2 + 1^2 + 2^2
--    7 = 1^2 + 1^2 + 2^2 + 1^2
--    7 = 1^2 + 2^2 + 1^2 + 1^2
--    7 = 2^2 + 1^2 + 1^2 + 1^2]]
--
-- Definir la función
--    nDescomposiciones       :: Int -> Int
-- tal que (nDescomposiciones x) es el número de listas de los cuadrados
-- de cuatro números enteros positivos cuya suma es x. Por ejemplo.
--      nDescomposiciones 7      ==  4
--      nDescomposiciones 4      ==  1
--      nDescomposiciones 5      ==  0
--      nDescomposiciones 10     ==  6
--      nDescomposiciones 15     ==  12
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

nDescomposiciones :: Int -> Int
nDescomposiciones = length . descomposiciones

-- (descomposiciones x) es la lista de las listas de los cuadrados de
-- cuatro números enteros positivos cuya suma es x. Por  ejemplo.
--    λ> descomposiciones 4
--    [[1,1,1,1]]
--    λ> descomposiciones 5
--    []
--    λ> descomposiciones 7
--    [[1,1,1,4],[1,1,4,1],[1,4,1,1],[4,1,1,1]]
--    λ> descomposiciones 10
```

```haskell
--    [[1,1,4,4],[1,4,1,4],[1,4,4,1],[4,1,1,4],[4,1,4,1],[4,4,1,1]]
--    λ> descomposiciones 15
--    [[1,1,4,9],[1,1,9,4],[1,4,1,9],[1,4,9,1],[1,9,1,4],[1,9,4,1],
--     [4,1,1,9],[4,1,9,1],[4,9,1,1],[9,1,1,4],[9,1,4,1],[9,4,1,1]]
descomposiciones :: Int -> [[Int]]
descomposiciones x = aux x 4
  where
    aux 0 1 = []
    aux 1 1 = [[1]]
    aux 2 1 = []
    aux 3 1 = []
    aux y 1 | esCuadrado y = [[y]]
            | otherwise    = []
    aux y n = [x^2 : zs | x <- [1..raizEntera y]
                        , zs <- aux (y - x^2) (n-1)]

-- (esCuadrado x) se verifica si x es un número al cuadrado. Por
-- ejemplo,
--    esCuadrado 25  ==  True
--    esCuadrado 26  ==  False
esCuadrado :: Int -> Bool
esCuadrado x = (raizEntera x)^2 == x

-- (raizEntera n) es el mayor entero cuya raíz cuadrada es menor o igual
-- que n. Por ejemplo,
--    raizEntera 15  ==  3
--    raizEntera 16  ==  4
--    raizEntera 17  ==  4
raizEntera :: Int -> Int
raizEntera = floor . sqrt . fromIntegral

-- 2ª solución
-- =============

nDescomposiciones2 :: Int -> Int
nDescomposiciones2 = length . descomposiciones2

descomposiciones2 :: Int -> [[Int]]
descomposiciones2 x = a ! (x,4)
  where
```

```
    a = array ((0,1),(x,4)) [((i,j), f i j) | i <- [0..x], j <- [1..4]]
    f 0 1 = []
    f 1 1 = [[1]]
    f 2 1 = []
    f 3 1 = []
    f i 1 | esCuadrado i = [[i]]
          | otherwise    = []
    f i j = [x^2 : zs | x <- [1..raizEntera i]
                      , zs <- a ! (i - x^2,j-1)]

-- 3ª solución
-- ===========

nDescomposiciones3 :: Int -> Int
nDescomposiciones3 x = aux x 4
  where
    aux 0 1 = 0
    aux 1 1 = 1
    aux 2 1 = 0
    aux 3 1 = 0
    aux y 1 | esCuadrado y = 1
            | otherwise    = 0
    aux y n = sum [aux (y - x^2) (n-1) | x <- [1..raizEntera y]]

-- 4ª solución
-- ===========

nDescomposiciones4 :: Int -> Int
nDescomposiciones4 x = a ! (x,4)
  where
    a = array ((0,1),(x,4)) [((i,j), f i j) | i <- [0..x], j <- [1..4]]
    f 0 1 = 0
    f 1 1 = 1
    f 2 1 = 0
    f 3 1 = 0
    f i 1 | esCuadrado i = 1
          | otherwise    = 0
    f i j = sum [a ! (i- x^2,j-1) | x <- [1..raizEntera i]]

-- Comprobación de equivalencia
```

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

-- La propiedad es
prop_nDescomposiciones :: Positive Int -> Bool
prop_nDescomposiciones (Positive x) =
  all (== nDescomposiciones x) [f x | f <- [ nDescomposiciones2
                                           , nDescomposiciones3
                                           , nDescomposiciones4]]

-- La comprobación es
--    λ> quickCheck prop_nDescomposiciones
--    +++ OK, passed 100 tests.

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

-- La comparación es
--    λ> nDescomposiciones 20000
--    1068
--    (3.69 secs, 3,307,250,128 bytes)
--    λ> nDescomposiciones2 20000
--    1068
--    (0.72 secs, 678,419,328 bytes)
--    λ> nDescomposiciones3 20000
--    1068
--    (3.94 secs, 3,485,725,552 bytes)
--    λ> nDescomposiciones4 20000
--    1068
--    (0.74 secs, 716,022,456 bytes)
--
--    λ> nDescomposiciones2 50000
--    5682
--    (2.64 secs, 2,444,206,000 bytes)
--    λ> nDescomposiciones4 50000
--    5682
--    (2.77 secs, 2,582,443,448 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2 [1.5 puntos]. Con la definición del apartado anterior,
-- evaluar (en menos de 2 segundos),
```

```haskell
--    nDescomposiciones (2*10^4)
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> timeout (2*10^6) (return $! (nDescomposiciones4 (2*10^4)))
--    Just 1068
--    (1.13 secs, 715,951,808 bytes)
```

1.5. Examen 5 (10 de abril de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 5º examen de evaluación continua (3 de mayo de 2019)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Array as A
import Data.Map as M
import Data.Set as S
import I1M.Grafo
import System.Timeout

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. [1 punto] Dado 4 puntos de un círculo se pueden
-- dibujar 2 cuerdas entre ellos de forma que no se corten. En efecto,
-- si se enumeran los puntos del 1 al 4 en sentido de las agujas del
-- reloj una forma tiene las cuerdas {1-2, 3-4} y la otra {1-4, 2-3}.
--
-- Definir la función
--    numeroFormas :: Integer -> Integer
-- tal que (numeroFormas n) es el número de formas que se pueden dibujar
-- n cuerdas entre 2xn puntos de un círculo sin que se corten. Por
-- ejemplo,
--    numeroFormas 1   ==  1
--    numeroFormas 2   ==  2
--    numeroFormas 4   ==  14
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
numeroFormas :: Integer -> Integer
numeroFormas 0 = 0
numeroFormas n = aux (2*n)
  where aux 0 = 1
        aux 2 = 1
        aux i = sum [aux j * aux (i-2-j) | j <- [0,2..i-1]]

-- 2ª definición
numeroFormas2 :: Integer -> Integer
numeroFormas2 0 = 0
numeroFormas2 n = v A.! (2*n)
  where v   = array (0,2*n) [(i, f i) | i <- [0..2*n]]
        f 0 = 1
        f 2 = 1
        f i = sum [v A.! j * v A.! (i-2-j) | j <- [0,2..i-1]]

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

--    λ> numeroFormas 15
--    9694845
--    (28.49 secs, 4,293,435,552 bytes)
--    λ> numeroFormas2 15
--    9694845
--    (0.01 secs, 184,552 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2 [1.5 puntos]. Con la definición del apartado anterior,
-- evaluar (en menos de 2 segundos), el número de dígitos de
-- (numeroFormas 700); es decir, evaluar la siguiente expresión para que
-- de un valor distinto de Nothing
--    timeout (2*10^6) (return $! (length (show (numeroFormas 700))))
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> timeout (2*10^6) (return $! (length (show (numeroFormas 700))))
--    Just 417
--    (1.65 secs, 230,243,040 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos]. Definir la función
--    mayoritarios :: Ord a => Set (Set a) -> [a]
-- tal que (mayoritarios f) es la lista de elementos que pertenecen al
-- menos a la mitad de los conjuntos de la familia f. Por ejemplo,
--    λ> mayoritarios (S.fromList [S.empty, S.fromList [1,3], S.fromList [3,5]])
--    [3]
--    λ> mayoritarios (S.fromList [S.empty, S.fromList [1,3], S.fromList [4,5]])
--    []
--    λ> mayoritarios (S.fromList [S.fromList [1..n] | n <- [1..7]])
--    [1,2,3,4]
--    λ> mayoritarios (S.fromList [S.fromList [1..n] | n <- [1..8]])
--    [1,2,3,4,5]
-- ---------------------------------------------------------------------

mayoritarios :: Ord a => Set (Set a) -> [a]
mayoritarios f =
  [x | x <- S.toList (elementosFamilia f)
     , nOcurrencias f x >= n]
  where n = (1 + S.size f) `div` 2

-- (elementosFamilia f) es el conjunto de los elementos de los elementos
-- de la familia f. Por ejemplo,
--    λ> elementosFamilia (S.fromList [S.fromList [1,2], S.fromList [2,5]])
--    fromList [1,2,5]
elementosFamilia :: Ord a => Set (Set a) -> Set a
elementosFamilia = S.unions . S.toList

-- (nOcurrencias f x) es el número de conjuntos de la familia f a los
-- que pertenece el elemento x. nOcurrencias :: Ord a => Set (Set a) -> a -> Int
nOcurrencias f x =
  length [c | c <- S.toList f, x `S.member` c]

-- ---------------------------------------------------------------------
-- Ejercicio 3 [2.5 puntos]. Los polinomios se pueden representar
-- mediante diccionarios con los exponentes como claves y los
-- coeficientes como valores.
--
-- El tipo de los polinomios con coeficientes de tipo a se define por
```

```haskell
--     type Polinomio a = M.Map Int a
--
-- Dos ejemplos de polinomios (que usaremos en los ejemplos) son
--     3 + 7x - 5x^3
--     4 + 5x^3 + x^5
-- se definen por
--    ejPol1, ejPol2 :: Polinomio Int
--    ejPol1 = M.fromList [(0,3),(1,7),(3,-5)]
--    ejPol2 = M.fromList [(0,4),(3,5),(5,1)]
--
-- Definir la función
--     multPol :: (Eq a, Num a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (multPol p q) es el producto de los polinomios p y q. Por ejemplo,
--     ghci> multPol ejPol1 ejPol2
--     fromList [(0,12),(1,28),(3,-5),(4,35),(5,3),(6,-18),(8,-5)]
--     ghci> multPol ejPol1 ejPol1
--     fromList [(0,9),(1,42),(2,49),(3,-30),(4,-70),(6,25)]
--     ghci> multPol ejPol2 ejPol2
--     fromList [(0,16),(3,40),(5,8),(6,25),(8,10),(10,1)]
-- ---------------------------------------------------------------------

type Polinomio a = M.Map Int a

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

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

-- (multPorTerm (n,a) p) es el producto del término ax^n por p. Por
-- ejemplo,
--     ghci> multPorTerm (2,3) (M.fromList [(0,4),(2,1)])
--     fromList [(2,12),(4,3)]
multPorTerm :: Num a => (Int,a) -> Polinomio a -> Polinomio a
multPorTerm (n,a) p =
  M.map (*a) (M.mapKeys (+n) p)
```

```haskell
-- (sumaPol p q) es la suma de los polinomios p y q. Por ejemplo,
--    ghci> sumaPol ejPol1 ejPol2
--    fromList [(0,7),(1,7),(5,1)]
--    ghci> sumaPol ejPol1 ejPol1
--    fromList [(0,6),(1,14),(3,-10)]
sumaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
sumaPol p q =
  M.filter (/=0) (M.unionWith (+) p q)

-- ---------------------------------------------------------------------
-- Ejercicio 4 [2.5 puntos] Definir las funciones
--    grafoD  :: [(Int,Int)] -> Grafo Int Int
--    grafoND :: Grafo Int Int -> Grafo Int Int
-- tales que
-- + (grafoD ps) es el grafo dirigido cuyas cuyos nodos son todos los
--   elementos comprendidos entre el menor y el mayor de las componentes
--   de los elementos de ps y las aristas son los elementos de ps
--   añadiéndole el peso cero. Por ejemplo,
--      λ> grafoD [(1,3),(1,4),(4,1)]
--      G D (array (1,4) [(1,[(3,0),(4,0)]),(2,[]),(3,[]),(4,[(1,0)])])
--      λ> grafoD [(1,1),(1,2),(2,2)]
--      G D (array (1,2) [(1,[(1,0),(2,0)]),(2,[(2,0)])])
-- + (grafoND g) es el grafo no dirigido correspondiente al grafo
--   dirigido g (en el que todos los pesos son 0); es decir, es un grafo
--   que tiene el mismo conjunto de nodos que g pero sus aristas son las
--   de g junto con sus inversas. Por ejemplo,
--      λ> grafoND (grafoD [(1,3),(1,4),(4,1)])
--      G ND (array (1,4) [(1,[(3,0),(4,0)]),(2,[]),(3,[(1,0)]),(4,[(1,0)])])
--      λ> grafoND (grafoD [(1,1),(1,2),(2,2)])
--      G ND (array (1,2) [(1,[(1,0),(2,0)]),(2,[(2,0),(1,0)])])
-- ---------------------------------------------------------------------

grafoD :: [(Int,Int)] -> Grafo Int Int
grafoD ps = creaGrafo D (1,n) [(x,y,0) | (x,y) <- ps]
  where n = maximum [max x y | (x,y) <- ps]

grafoND :: Grafo Int Int -> Grafo Int Int
grafoND g = creaGrafo ND (1,n) (nub [(x,y,0) | (x,y,_) <- as ++ bs])
  where n  = maximum (nodos g)
```

```haskell
        as = aristas g
        bs = [(y,x,p) | (x,y,p) <- as]
```

1.6. Examen 6 (10 de junio de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 1 y 4)
-- 6º examen de evaluación continua (10 de junio de 2019)
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    transformada :: [a] -> [a]
-- tal que (transformada xs) es la lista obtenida repitiendo cada
-- elemento tantas veces como indica su posición en la lista. Por
-- ejemplo,
--    transformada [7,2,5] == [7,2,2,5,5,5]
--    transformada "eco"   == "eccooo"
--
-- Comprobar con QuickCheck si la longitud de la transformada de una
-- lista de n elementos es ((n+1)^3 - n^3 - 1)/6.
-- ---------------------------------------------------------------------

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

-- La propiedad es
prop_transformada :: [Int] -> Bool
prop_transformada xs =
  length (transformada xs) == ((n+1)^3 - n^3 - 1) `div` 6
  where n = length xs
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. La sucesión de Loomis generada por un número entero
-- positivo x es la sucesión cuyos términos se definen por
-- + f(0) es x
-- + f(n) es la suma de f(n-1) y el producto de los dígitos no nulos de
--   f(n-1)
-- Los primeros términos de las primeras sucesiones de Loomis son
-- + Generada por 1: 1,2,4,8,16,22,26,38,62,74,102,104,108,116,122,...
-- + Generada por 2: 2,4,8,16,22,26,38,62,74,102,104,108,116,122,126,...
-- + Generada por 3: 3,6,12,14,18,26,38,62,74,102,104,108,116,122,126,...
-- + Generada por 4: 4,8,16,22,26,38,62,74,102,104,108,116,122,126,138,...
-- + Generada por 5: 5,10,11,12,14,18,26,38,62,74,102,104,108,116,122,...
--
-- Se observa que a partir de un término todas coinciden con la
-- generada por 1. Dicho término se llama el punto de convergencia. Por
-- ejemplo,
-- + la generada por 2 converge a 2
-- + la generada por 3 converge a 26
-- + la generada por 4 converge a 4
-- + la generada por 5 converge a 26
--
-- Definir las siguientes funciones
--    sucLoomis           :: Integer -> [Integer]
--    convergencia        :: Integer -> Integer
-- tales que
-- + (sucLoomis x) es la sucesión de Loomis generada por x. Por ejemplo,
--      λ> take 15 (sucLoomis 1)
--      [1,2,4,8,16,22,26,38,62,74,102,104,108,116,122]
--      λ> take 15 (sucLoomis 2)
--      [2,4,8,16,22,26,38,62,74,102,104,108,116,122,126]
--      λ> take 15 (sucLoomis 3)
--      [3,6,12,14,18,26,38,62,74,102,104,108,116,122,126]
--      λ> take 15 (sucLoomis 4)
--      [4,8,16,22,26,38,62,74,102,104,108,116,122,126,138]
--      λ> take 15 (sucLoomis 5)
```

```haskell
--      [5,10,11,12,14,18,26,38,62,74,102,104,108,116,122]
--      λ> take 15 (sucLoomis 20)
--      [20,22,26,38,62,74,102,104,108,116,122,126,138,162,174]
--      λ> take 15 (sucLoomis 100)
--      [100,101,102,104,108,116,122,126,138,162,174,202,206,218,234]
--      λ> sucLoomis 1 !! (2*10^5)
--      235180736652
-- + (convergencia x) es el término de convergencia de la sucesión de
--   Loomis generada por x con la generada por 1. Por ejemplo,
--      convergencia  2       ==  2
--      convergencia  3       ==  26
--      convergencia  4       ==  4
--      convergencia 17       ==  38
--      convergencia 19       ==  102
--      convergencia 43       ==  162
--      convergencia 27       ==  202
--      convergencia 58       ==  474
--      convergencia 63       ==  150056
--      convergencia 81       ==  150056
--      convergencia 89       ==  150056
--      convergencia (10^12) ==  1000101125092
-- ---------------------------------------------------------------------

-- 1ª definición de sucLoomis
-- ==========================

sucLoomis :: Integer -> [Integer]
sucLoomis x = map (loomis x) [0..]

loomis :: Integer -> Integer -> Integer
loomis x 0 = x
loomis x n = y + productoDigitosNoNulos y
  where y = loomis x (n-1)

productoDigitosNoNulos :: Integer -> Integer
productoDigitosNoNulos = product . digitosNoNulos

digitosNoNulos :: Integer -> [Integer]
digitosNoNulos x =
  [read [c] | c <- show x, c /= '0']
```

```haskell
-- 2ª definición de sucLoomis
-- ==========================

sucLoomis2 :: Integer -> [Integer]
sucLoomis2 = iterate siguienteLoomis

siguienteLoomis :: Integer -> Integer
siguienteLoomis y = y + productoDigitosNoNulos y

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

--    λ> sucLoomis 1 !! 30000
--    6571272766
--    (2.45 secs, 987,955,944 bytes)
--    λ> sucLoomis2 1 !! 30000
--    6571272766
--    (2.26 secs, 979,543,328 bytes)

-- 1ª definición de convergencia
-- =============================

convergencia1 :: Integer -> Integer
convergencia1 x =
  head (dropWhile noEnSucLoomisDe1 (sucLoomis x))

noEnSucLoomisDe1 :: Integer -> Bool
noEnSucLoomisDe1 x = not (pertenece x sucLoomisDe1)

sucLoomisDe1 :: [Integer]
sucLoomisDe1 = sucLoomis 1

pertenece :: Integer -> [Integer] -> Bool
pertenece x ys =
  x == head (dropWhile (<x) ys)

-- 2ª definición de convergencia
-- =============================
```

```haskell
convergencia2 :: Integer -> Integer
convergencia2 = aux (sucLoomis2 1) . sucLoomis2
  where aux as@(x:xs) bs@(y:ys) | x == y    = x
                                | x < y     = aux xs bs
                                | otherwise = aux as ys

-- 3ª definición de convergencia
-- ==============================

convergencia3 :: Integer -> Integer
convergencia3 x = perteneceA (sucLoomis2 x) 1
  where perteneceA (y:ys) n | y == c    = y
                            | otherwise = perteneceA ys c
          where c = head $ dropWhile (< y) $ sucLoomis2 n

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

--    λ> convergencia1 (10^4)
--    150056
--    (2.94 secs, 1,260,809,808 bytes)
--    λ> convergencia2 (10^4)
--    150056
--    (0.03 secs, 700,240 bytes)
--    λ> convergencia3 (10^4)
--    150056
--    (0.02 secs, 1,119,648 bytes)
--
--    λ> convergencia2 (10^12)
--    1000101125092
--    (1.81 secs, 714,901,080 bytes)
--    λ> convergencia3 (10^12)
--    1000101125092
--    (1.82 secs, 941,053,328 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los caminos desde el extremo superior izquierdo
-- (posición (1,1)) hasta el extremo inferior derecho (posición (3,4))
-- en la matriz
--    (  1  6 11  2 )
```

```haskell
--    (  7 12  3  8 )
--    (  3  8  4  9 )
-- moviéndose en cada paso una casilla hacia abajo o hacia la derecha,
-- son los siguientes:
--    1, 7,  3, 8, 4, 9
--    1, 7, 12, 8, 4, 9
--    1, 7, 12, 3, 4, 9
--    1, 7, 12, 3, 8, 9
--    1, 6, 12, 8, 4, 9
--    1, 6, 12, 3, 4, 9
--    1, 6, 12, 3, 8, 9
--    1, 6, 11, 3, 4, 9
--    1, 6, 11, 3, 8, 9
--    1, 6, 11, 2, 8, 9
-- Las sumas de los caminos son 32, 41, 36, 40, 40, 35, 39, 34, 38 y 37,
-- respectivamente. El camino de máxima suma es el segundo (1, 7, 12, 8,
-- 4, 9) que tiene una suma de 41.
--
-- Definir, usando programación dinámica, la función
--    caminoMaxSuma :: Matrix Int -> [Int]
-- tal que (caminoMaxSuma m) es un camino de máxima suma en la matriz m
-- desde el extremo superior izquierdo hasta el extremo inferior
-- derecho, moviéndose en cada paso una casilla hacia abajo o hacia la
-- derecha. Por ejemplo,
--    λ> caminoMaxSuma (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
--    [1,7,12,8,4,9]
--    λ> sum (caminoMaxSuma (fromList 400 500 [1..]))
--    139576149
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

caminoMaxSuma1 :: Matrix Int -> [Int]
caminoMaxSuma1 m =
  head [c | c <- cs, sum c == k]
  where cs = caminos1 m
        k  = maximum (map sum cs)

caminos1 :: Matrix Int -> [[Int]]
```

```
caminos1 m =
  map reverse (caminos1Aux m (nf,nc))
  where nf = nrows m
        nc = ncols m

-- (caminos1Aux p x) es la lista de los caminos invertidos en la matriz p
-- desde la posición (1,1) hasta la posición x. Por ejemplo,
caminos1Aux :: Matrix Int -> (Int,Int) -> [[Int]]
caminos1Aux m (1,1) = [[m!(1,1)]]
caminos1Aux m (1,j) = [[m!(1,k) | k <- [j,j-1..1]]]
caminos1Aux m (i,1) = [[m!(k,1) | k <- [i,i-1..1]]]
caminos1Aux m (i,j) = [m!(i,j) : xs
                      | xs <- caminos1Aux m (i,j-1) ++
                              caminos1Aux m (i-1,j)]

-- 2ª definición
-- =============

caminoMaxSuma2 :: Matrix Int -> [Int]
caminoMaxSuma2 m =
  head [c | c <- cs, sum c == k]
  where cs = caminos2 m
        k  = maximum (map sum cs)

caminos2 :: Matrix Int -> [[Int]]
caminos2 m =
  map reverse (matrizCaminos m ! (nrows m, ncols m))

matrizCaminos :: Matrix Int -> Matrix [[Int]]
matrizCaminos m = q
  where
    q = matrix (nrows m) (ncols m) f
    f (1,y) = [[m!(1,z) | z <- [y,y-1..1]]]
    f (x,1) = [[m!(z,1) | z <- [x,x-1..1]]]
    f (x,y) = [m!(x,y) : cs | cs <- q!(x-1,y) ++ q!(x,y-1)]

-- 3ª definición (con programación dinámica)
-- =========================================

caminoMaxSuma :: Matrix Int -> [Int]
```

```
caminoMaxSuma m = reverse (snd (q ! (nf,nc)))
  where nf = nrows m
        nc = ncols m
        q  = caminoMaxSumaAux m

caminoMaxSumaAux :: Matrix Int -> Matrix (Int,[Int])
caminoMaxSumaAux m = q
  where
    nf = nrows m
    nc = ncols m
    q  = matrix nf nc f
      where
        f (1,1) = (m!(1,1),[m!(1,1)])
        f (1,j) = (k + m!(1,j), m!(1,j):xs)
          where (k,xs) = q!(1,j-1)
        f (i,1) = (k + m!(i,1), m!(i,1):xs)
          where (k,xs) = q!(i-1,1)
        f (i,j) | k1 > k2   = (k1 + m!(i,j), m!(i,j):xs)
                | otherwise = (k2 + m!(i,j), m!(i,j):ys)
          where (k1,xs) = q!(i,j-1)
                (k2,ys) = q!(i-1,j)

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

--    λ> length (caminoMaxSuma1 (fromList 11 11 [1..]))
--    21
--    (10.00 secs, 1,510,120,328 bytes)
--    λ> length (caminoMaxSuma2 (fromList 11 11 [1..]))
--    21
--    (3.84 secs, 745,918,544 bytes)
--    λ> length (caminoMaxSuma3 (fromList 11 11 [1..]))
--    21
--    (0.01 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se dice que A es un subconjunto maximal de B si A ⊂ B y
-- no existe ningún C tal que A ⊂ C y C ⊂ B. Por ejemplo, {2,5} es un
-- subconjunto maximal de {2,3,5], pero {3] no lo es.
--
```

```haskell
-- El árbol de los subconjuntos de un conjunto A es el árbol que tiene
-- como raíz el conjunto A y cada nodo tiene como hijos sus subconjuntos
-- maximales. Por ejemplo, el árbol de subconjuntos de [2,3,5] es
--
--            [2, 3, 5]
--            /   |  \
--           /    |   \
--          /     |    \
--         /      |     \
--        /       |      \
--      [5,3]   [2,3]   [2,5]
--      /  \    /  \    /  \
--    [3] [5] [3] [2] [5] [2]
--     |   |   |   |   |   |
--    [ ] [ ] [ ] [ ] [ ] [ ]
--
-- Usando el tipo de dato
--    data Arbol = N Integer [Arbol]
--      deriving (Eq, Show)
-- el árbol anterior se representa por
--    N [2,5,3]
--      [N [5,3]
--         [N [3]
--            [N [] []],
--          N [5]
--            [N [] []]],
--       N [2,3]
--         [N [3]
--            [N [] []],
--          N [2]
--            [N [] []]],
--       N [2,5]
--         [N [5]
--            [N [] []],
--          N [2]
--            [N [] []]]]
-- Definir las funciones
--    arbolSubconjuntos             :: [Int] -> Arbol
--    nOcurrenciasArbolSubconjuntos :: [Int] -> [Int] -> Int
-- tales que
```

```haskell
-- + (arbolSubconjuntos x) es el árbol de los subconjuntos de xs. Por
--   ejemplo,
--      λ> arbolSubconjuntos [2,5,3]
--      N [2,5,3] [N [5,3] [N [3] [N [] []],N [5] [N [] []]],
--                 N [2,3] [N [3] [N [] []],N [2] [N [] []]],
--                 N [2,5] [N [5] [N [] []],N [2] [N [] []]]]
-- + (nOcurrenciasArbolSubconjuntos xs ys) es el número de veces que
--   aparece el conjunto xs en el árbol de los subconjuntos de ys. Por
--   ejemplo,
--      nOcurrenciasArbolSubconjuntos []      [2,5,3]  ==  6
--      nOcurrenciasArbolSubconjuntos [3]     [2,5,3]  ==  2
--      nOcurrenciasArbolSubconjuntos [3,5]   [2,5,3]  ==  1
--      nOcurrenciasArbolSubconjuntos [3,5,2] [2,5,3]  ==  1
-- ---------------------------------------------------------------------

data Arbol = N [Int] [Arbol]
  deriving (Eq, Show)

arbolSubconjuntos :: [Int] -> Arbol
arbolSubconjuntos xs =
  N xs (map arbolSubconjuntos (subconjuntosMaximales xs))

-- (subconjuntosMaximales xs) es la lista de los subconjuntos maximales
-- de xs. Por ejemplo,
--    subconjuntosMaximales [2,5,3]  ==  [[5,3],[2,3],[2,5]]
subconjuntosMaximales :: [Int] -> [[Int]]
subconjuntosMaximales xs =
  [delete x xs | x <- xs]

-- Definición de nOcurrenciasArbolSubconjuntos
-- ===========================================

nOcurrenciasArbolSubconjuntos :: [Int] -> [Int] -> Int
nOcurrenciasArbolSubconjuntos xs ys =
  nOcurrencias xs (arbolSubconjuntos ys)

-- (nOcurrencias x a) es el número de veces que aparece x en el árbol
-- a. Por ejemplo,
--    nOcurrencias 3 (arbolSubconjuntos 30)  ==  2
nOcurrencias :: [Int] -> Arbol -> Int
```

```haskell
nOcurrencias xs (N ys [])
  | conjunto xs == conjunto ys  = 1
  | otherwise                   = 0
nOcurrencias xs (N ys zs)
  | conjunto xs == conjunto ys = 1 + sum [nOcurrencias xs z | z <- zs]
  | otherwise                  = sum [nOcurrencias xs z | z <- zs]

-- (conjunto xs) es el conjunto ordenado correspondiente a xs. Por
-- ejemplo,
--    conjunto [3,2,5,2,3,7,2]  ==  [2,3,5,7]
conjunto :: [Int] -> [Int]
conjunto = nub . sort
```

1.7. Examen 7 (02 de julio de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de 1º convocatoria (2 de julio de 2019)
-- ---------------------------------------------------------------------

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

import Data.Array
import Data.List
import I1M.Grafo
import qualified Data.Set as S

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    eliminaUnitarias :: Char -> String -> String
-- tal que (eliminaUnitarias c cs) es la lista obtenida eliminando de la
-- cadena cs las ocurrencias unitarias del carácter c (es decir,
-- aquellas ocurrencias de c tales que su elemento anterior y posterior
-- es distinto de c). Por ejemplo,
--    eliminaUnitarias 'X' ""                == ""
--    eliminaUnitarias 'X' "X"               == ""
--    eliminaUnitarias 'X' "XX"              == "XX"
--    eliminaUnitarias 'X' "XXX"             == "XXX"
```

```haskell
--    eliminaUnitarias 'X' "abcd"              == "abcd"
--    eliminaUnitarias 'X' "Xabcd"             == "abcd"
--    eliminaUnitarias 'X' "XXabcd"            == "XXabcd"
--    eliminaUnitarias 'X' "XXXabcd"           == "XXXabcd"
--    eliminaUnitarias 'X' "abcdX"             == "abcd"
--    eliminaUnitarias 'X' "abcdXX"            == "abcdXX"
--    eliminaUnitarias 'X' "abcdXXX"           == "abcdXXX"
--    eliminaUnitarias 'X' "abXcd"             == "abcd"
--    eliminaUnitarias 'X' "abXXcd"            == "abXXcd"
--    eliminaUnitarias 'X' "abXXXcd"           == "abXXXcd"
--    eliminaUnitarias 'X' "XabXcdX"           == "abcd"
--    eliminaUnitarias 'X' "XXabXXcdXX"        == "XXabXXcdXX"
--    eliminaUnitarias 'X' "XXXabXXXcdXXX"     == "XXXabXXXcdXXX"
--    eliminaUnitarias 'X' "XabXXcdXeXXXfXx"   == "abXXcdeXXXfx"
-- ---------------------------------------------------------------------

-- 1ª solución (por comprensión):
eliminaUnitarias :: Char -> String -> String
eliminaUnitarias c cs = concat [xs | xs <- group cs, xs /= [c]]

-- 2ª solución (por composición):
eliminaUnitarias2 :: Char -> String -> String
eliminaUnitarias2 c = concat . filter (/=[c]) . group

-- 3ª solución (por recursión):
eliminaUnitarias3 :: Char -> String -> String
eliminaUnitarias3 _ [] = []
eliminaUnitarias3 c [x] | c == x    = []
                        | otherwise = [x]
eliminaUnitarias3 c (x:y:zs)
  | x /= c    = x : eliminaUnitarias3 c (y:zs)
  | y /= c    = y : eliminaUnitarias3 c zs
  | otherwise = takeWhile (==c) (x:y:zs) ++
                eliminaUnitarias3 c (dropWhile (==c) zs)

-- 4ª solución (por recursión con acumuladores):
eliminaUnitarias4 :: Char -> String -> String
eliminaUnitarias4 c cs = reverse (aux0 cs "")
  where aux0 [] cs2                 = cs2
        aux0 (x:cs1) cs2 | x == c   = aux1 cs1 cs2
```

```haskell
                         | otherwise = aux0 cs1 (x:cs2)
        aux1 [] cs2                  = cs2
        aux1 (x:cs1) cs2 | x == c    = aux2 cs1 (c:c:cs2)
                         | otherwise = aux0 cs1 (x:cs2)
        aux2 [] cs2                  = cs2
        aux2 (x:cs1) cs2 | x == c    = aux2 cs1 (c:cs2)
                         | otherwise = aux0 cs1 (x:cs2)

-- 5ª solución (por recursión con span)
eliminaUnitarias5 :: Char -> String -> String
eliminaUnitarias5 c [] = []
eliminaUnitarias5 c cs
  | ys == [c] = xs ++ eliminaUnitarias5 c zs
  | otherwise = xs ++ ys ++ eliminaUnitarias5 c zs
  where (xs,us) = span (/=c) cs
        (ys,zs) = span (==c) us

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir con programación dinámica la función
--    descomposiciones :: Int -> [[Int]]
-- tal que (descomposiciones x) es la lista de las listas de los
-- cuadrados de cuatro números enteros positivos cuya suma es x. Por
-- ejemplo.
--    λ> descomposiciones 4
--    [[1,1,1,1]]
--    λ> descomposiciones 5
--    []
--    λ> descomposiciones 7
--    [[1,1,1,4],[1,1,4,1],[1,4,1,1],[4,1,1,1]]
--    λ> descomposiciones 10
--    [[1,1,4,4],[1,4,1,4],[1,4,4,1],[4,1,1,4],[4,1,4,1],[4,4,1,1]]
--    λ> descomposiciones 15
--    [[1,1,4,9],[1,1,9,4],[1,4,1,9],[1,4,9,1],[1,9,1,4],[1,9,4,1],
--     [4,1,1,9],[4,1,9,1],[4,9,1,1],[9,1,1,4],[9,1,4,1],[9,4,1,1]]
--    λ> length (descomposiciones 30000)
--    4612
-- ---------------------------------------------------------------------

-- La definición por recursión es
-- ==============================
```

```haskell
descomposiciones1 :: Int -> [[Int]]
descomposiciones1 x = aux x 4
  where
    aux 0 1 = []
    aux 1 1 = [[1]]
    aux 2 1 = []
    aux 3 1 = []
    aux y 1 | esCuadrado y = [[y]]
            | otherwise    = []
    aux y n = [z^2 : zs | z <- [1..raizEntera y]
                        , zs <- aux (y - z^2) (n-1)]

-- (esCuadrado x) se verifica si x es un número al cuadrado. Por
-- ejemplo,
--    esCuadrado 25  ==  True
--    esCuadrado 26  ==  False
esCuadrado :: Int -> Bool
esCuadrado x = (raizEntera x)^2 == x

-- (raizEntera n) es el mayor entero cuya raíz cuadrada es menor o igual
-- que n. Por ejemplo,
--    raizEntera 15  ==  3
--    raizEntera 16  ==  4
--    raizEntera 17  ==  4
raizEntera :: Int -> Int
raizEntera = floor . sqrt . fromIntegral

-- La definición con programación dinámica es
-- ==========================================

descomposiciones :: Int -> [[Int]]
descomposiciones x = a ! (x,4)
  where
    a = array ((0,1),(x,4)) [((i,j), f i j) | i <- [0..x], j <- [1..4]]
    f 0 1 = []
    f 1 1 = [[1]]
    f 2 1 = []
    f 3 1 = []
    f i 1 | esCuadrado i = [[i]]
```

```haskell
          | otherwise    = []
    f i j = [z^2 : zs | z <- [1..raizEntera i]
                      , zs <- a ! (i - z^2,j-1)]

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

-- La comparación es
--    λ> length (descomposiciones1 (2*10^4))
--    1068
--    (3.70 secs, 3,307,251,704 bytes)
--    λ> length (descomposiciones (2*10^4))
--    1068
--    (0.72 secs, 678,416,144 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    minimales :: Ord a => S.Set (S.Set a) -> S.Set (S.Set a)
-- tal que (minimales xss) es el conjunto de los elementos de xss que no
-- contienen a ningún otro elemento de xss. Por ejemplo,
--    λ> minimales (S.fromList (map S.fromList [[1,3],[2,3,1],[3,2,5]]))
--    fromList [fromList [1,3],fromList [2,3,5]]
--    λ> minimales (S.fromList (map S.fromList [[1,3],[2,3,1],[3,1],[3,2,5]]))
--    fromList [fromList [1,3],fromList [2,3,5]]
--    λ> minimales (S.fromList (map S.fromList [[1,3],[2,3,1],[3,1,3],[3,2,5]]))
--    fromList [fromList [1,3],fromList [2,3,5]]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un ciclo en un grafo G es una secuencia de nodos
-- [v(1),v(2),v(3),...,v(n)] de G tal que:
--    1) (v(1),v(2)), (v(2),v(3)), (v(3),v(4)), ..., (v(n-1),v(n)) son
--       aristas de G,
--    2) v(1) = v(n), y
--    3) salvo v(1) = v(n), todos los v(i) son distintos entre sí.
--
```

```haskell
-- Definir la función
--    ciclos :: Grafo Int Int -> [[Int]]
-- tal que (ciclos g) es la lista de ciclos de g. Por ejemplo, si g1 y
-- g2 son los grafos definidos por
--    g1, g2 :: Grafo Int Int
--    g1 = creaGrafo D (1,4) [(1,2,0),(2,3,0),(2,4,0),(4,1,0)]
--    g2 = creaGrafo D (1,4) [(1,2,0),(2,1,0),(2,4,0),(4,1,0)]
-- entonces
--    ciclos g1  ==  [[1,2,4,1],[2,4,1,2],[4,1,2,4]]
--    ciclos g2  ==  [[1,2,1],[1,2,4,1],[2,1,2],[2,4,1,2],[4,1,2,4]]
-- Como se observa en los ejemplos, un mismo ciclo puede tener
-- distintas representaciones según su vértice inicial.
-- ---------------------------------------------------------------------

g1, g2 :: Grafo Int Int
g1 = creaGrafo D (1,4) [(1,2,0),(2,3,0),(2,4,0),(4,1,0)]
g2 = creaGrafo D (1,4) [(1,2,0),(2,1,0),(2,4,0),(4,1,0)]

-- 1ª definición
-- =============
ciclos :: Grafo Int Int -> [[Int]]
ciclos g =
  sort [ys | (x:xs) <- concatMap permutations (subsequences (nodos g))
           , let ys = (x:xs) ++ [x]
           , esCiclo ys g]

-- (esCiclo vs g) se verifica si vs es un ciclo en el grafo g. Por
-- ejemplo,
esCiclo :: [Int] -> Grafo Int Int -> Bool
esCiclo vs g =
  all (aristaEn g) (zip vs (tail vs)) &&
  head vs == last vs &&
  length (nub vs) == length vs - 1

-- 2ª definición
-- =============

ciclos2 :: Grafo Int Int -> [[Int]]
ciclos2 g = sort [ys | (x:xs) <- caminos g
                     , let ys = (x:xs) ++ [x]
```

```haskell
                      , esCiclo ys g]

-- (caminos g) es la lista de los caminos en el grafo g. Por ejemplo,
--    caminos g1  ==  [[1,2,3],[1,2,4],[2,3],[2,4,1],[3],[4,1,2,3]]
caminos :: Grafo Int Int -> [[Int]]
caminos g = concatMap (caminosDesde g) (nodos g)

-- (caminosDesde g v) es la lista de los caminos en el grafo g a partir
-- del vértice v. Por ejemplo,
--    caminosDesde g1 1  ==  [[1],[1,2],[1,2,3],[1,2,4]]
--    caminosDesde g1 2  ==  [[2],[2,3],[2,4],[2,4,1]]
--    caminosDesde g1 3  ==  [[3]]
--    caminosDesde g1 4  ==  [[4],[4,1],[4,1,2],[4,1,2,3]]
caminosDesde :: Grafo Int Int -> Int -> [[Int]]
caminosDesde g v =
  map (reverse . fst) $
  concat $
  takeWhile (not.null) (iterate (concatMap sucesores) [([v],[v])])
  where sucesores (x:xs,ys) = [(z:x:xs,z:ys) | z <- adyacentes g x
                                             , z `notElem` ys]
```

1.8. Examen 8 (11 de septiembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de 2º convocatoria (11 de septiembre de 2019)
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1 En la aritmética lunar la suma se hace como en la
-- terrícola salvo que sus tablas de sumar son distintas. La suma lunar
```

```haskell
-- de dos dígitos es su máximo (por ejemplo, 1 + 3 = 3 y 7 + 4 = 7). Por
-- tanto,
--       3 5 7
--     +   6 4
--     -------
--       3 6 7
--
-- Definir la función
--    suma :: Integer -> Integer -> Integer
-- tal que (suma x y) es la suma lunar de x e y. Por ejemplo,
--      suma 357 64  ==  367
--      suma 64 357  ==  367
--      suma 1 3     ==  3
--      suma 7 4     ==  7
-- ---------------------------------------------------------------------

suma :: Integer -> Integer -> Integer
suma 0 0 = 0
suma x y = max x2 y2 + 10 * suma x1 y1
  where (x1,x2) = x `divMod` 10
        (y1,y2) = y `divMod` 10

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que la suma lunar es
-- conmutativa.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_conmutativa :: Integer -> Integer -> Property
prop_conmutativa x y =
  x >= 0 && y >= 0 ==> suma x y == suma y x

-- La comprobación es
--    λ> quickCheck prop_conmutativa
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. La sucesión de Fibonacci es
--    0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,...
-- cuyos dos primeros términos son 0 y 1 y los restantes se obtienen
```

```
-- sumando los dos anteriores.
--
-- El árbol de computación de su 5º término es
--                  5
--                 / \
--                /   \
--               /     \
--              /       \
--             /         \
--            3           2
--           / \         / \
--          /   \       1   1
--         2     1     / \
--        / \   / \   1   0
--       1   1 1   0
--      / \
--     1   0
-- que, usando los árboles definidos por
--    data Arbol = H Int
--               | N Int Arbol Arbol
--      deriving (Eq, Show)
-- se puede representar por
--    N 5
--      (N 3
--         (N 2
--            (N 1 (H 1) (H 0))
--            (H 1))
--         (N 1 (H 1) (H 0)))
--      (N 2
--         (N 1 (H 1) (H 0))
--         (H 1))
--
-- Definir la función
--    arbolFib           :: Int -> Arbol
-- tal que (arbolFib n) es el árbol de computación del n-ésimo término
-- de la sucesión de Fibonacci. Por ejemplo,
--      λ> arbolFib 5
--      N 5
--        (N 3
--           (N 2
```

```haskell
--              (N 1 (H 1) (H 0))
--              (H 1))
--           (N 1 (H 1) (H 0)))
--        (N 2
--           (N 1 (H 1) (H 0))
--           (H 1))
--      λ> arbolFib 6
--      N 8
--        (N 5
--           (N 3
--              (N 2
--                 (N 1 (H 1) (H 0))
--                 (H 1))
--              (N 1 (H 1) (H 0)))
--           (N 2
--              (N 1 (H 1) (H 0))
--              (H 1)))
--        (N 3
--           (N 2
--              (N 1 (H 1) (H 0)) (H 1))
--           (N 1 (H 1) (H 0)))
-- ---------------------------------------------------------------------

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

-- 1ª definición
-- =============

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

-- (fib n) es el n-ésimo elemento de la sucesión de Fibonacci. Por
-- ejemplo,
--    fib 5  ==  5
--    fib 6  ==  8
fib :: Int -> Int
```

```haskell
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

-- 2ª definición
-- =============

arbolFib2 :: Int -> Arbol
arbolFib2 0 = H 0
arbolFib2 1 = H 1
arbolFib2 2 = N 1 (H 1) (H 0)
arbolFib2 3 = N 2 (N 1 (H 1) (H 0)) (H 1)
arbolFib2 n = N (a1 + a2) (N a1 i1 d1) (N a2 i2 d2)
  where (N a1 i1 d1) = arbolFib2 (n-1)
        (N a2 i2 d2) = arbolFib2 (n-2)

-- 3ª definición
-- =============

arbolFib3 :: Int -> Arbol
arbolFib3 0 = H 0
arbolFib3 1 = H 1
arbolFib3 2 = N 1 (H 1) (H 0)
arbolFib3 3 = N 2 (N 1 (H 1) (H 0)) (H 1)
arbolFib3 n = N (a + b) i d
  where i@(N a _ _) = arbolFib3 (n-1)
        d@(N b _ _) = arbolFib3 (n-2)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. El polinomio cromático de un grafo calcula el número
-- de maneras en las cuales puede ser coloreado el grafo usando un
-- número de colores dado, de forma que dos vértices adyacentes no
-- tengan el mismo color.
--
-- En el caso del grafo completo de n vértices, su polinomio cromático
-- es P(n)(x) = x(x-1)(x-2) ... (x-(n-1)). Por ejemplo,
--    P(3)(x) = x(x-1)(x-2)       = x^3 - 3*x^2 + 2*x
--    P(4)(x) = x(x-1)(x-2)(x-3) = x^4 - 6*x^3 + 11*x^2 - 6*x
-- Lo que significa que P(4)(x) es el número de formas de colorear el
-- grafo completo de 4 vértices con x colores. Por tanto,
```

```haskell
--    P(4)(2) =  0 (no se puede colorear con 2 colores)
--    P(4)(4) = 24 (hay 24 formas de colorearlo con 4 colores)
--
-- Definir la función
--    polGC :: Int -> Polinomio Int
-- tal que (polGC n) es el polinomio cromático del grafo completo de n
-- vértices. Por ejemplo,
--    polGC 4  ==  x^4 + -6*x^3 + 11*x^2 + -6*x
--    polGC 5  ==  x^5 + -10*x^4 + 35*x^3 + -50*x^2 + 24*x
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

polGC :: Int -> Polinomio Int
polGC 0 = consPol 0 1 polCero
polGC n = polGC (n-1) `multPol` consPol 1 1 (consPol 0 (-n+1) polCero)

-- 2ª solución
-- ===========

polGC2 :: Int -> Polinomio Int
polGC2 n = multLista (map polMon [0..n-1])

-- (polMon n) es el monomio x-n. Por ejemplo,
--    polMon 3   ==   1*x + -3
polMon :: Int -> Polinomio Int
polMon n = consPol 1 1 (consPol 0 (-n) polCero)

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

-- 3ª solución (por plegado)
-- =========================

polGC3 :: Int -> Polinomio Int
polGC3 n = foldl multPol polUnidad
           [consPol 1 1 (consPol 0 (-i) polCero) | i <- [0..n-1]]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que si el número de colores
-- (x) coincide con el número de vértices del grafo (n), el número de
-- maneras de colorear el grafo es n!.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    ghci> quickCheckWith (stdArgs {maxSize=7}) prop_polGC
--    +++ OK, passed 100 tests.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_polGC :: Int -> Property
prop_polGC n =
  n > 0 ==> valor (polGC n) n == product [1..n]

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Una partición de un conjunto A es un conjunto de
-- subconjuntos no vacíos de A, disjuntos dos a dos y cuya unión es
-- A. Por ejemplo, el conjunto {1, 2, 3} tiene exactamente 5
-- particiones:
--    {{1}, {2}, {3}}
--    {{1,2}, {3}}
--    {{1,3}, {2}}
--    {{1}, {2,3}}
--    {{1,2,3}}
--
-- Definir la función
--    nParticiones :: [a] -> Integer
-- tal que (nParticiones xs) es el número de particiones de xs. Por
-- ejemplo,
--    nParticiones [1,2]                     ==  2
--    nParticiones [1,2,3]                   ==  5
--    nParticiones "abcd"                    ==  15
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
-- =============

nParticiones :: [a] -> Integer
nParticiones xs = sum [a ! (n,k) | k <- [0..n]]
  where n = genericLength xs
        a = matrizNParticiones n

-- (matrizNParticiones n) es la matriz de dimensión ((0,0),(n,n)) que en
-- la posición (i,j) tiene el número de particiones de un conjunto de i
-- elementos en j subconjuntos. Por ejemplo,
--    λ> matrizNParticiones 3
--    array ((0,0),(3,3))
--          [((0,0),0),((0,1),0),((0,2),0),((0,3),0),
--           ((1,0),0),((1,1),1),((1,2),0),((1,3),0),
--           ((2,0),0),((2,1),1),((2,2),1),((2,3),0),
--           ((3,0),0),((3,1),1),((3,2),3),((3,3),1)]
--    λ> matrizNParticiones 4
--    array ((0,0),(4,4))
--          [((0,0),0),((0,1),0),((0,2),0),((0,3),0),((0,4),0),
--           ((1,0),0),((1,1),1),((1,2),0),((1,3),0),((1,4),0),
--           ((2,0),0),((2,1),1),((2,2),1),((2,3),0),((2,4),0),
--           ((3,0),0),((3,1),1),((3,2),3),((3,3),1),((3,4),0),
--           ((4,0),0),((4,1),1),((4,2),7),((4,3),6),((4,4),1)]
matrizNParticiones :: Integer -> Array (Integer,Integer) Integer
matrizNParticiones n = a
  where
    a = array ((0,0),(n,n)) [((i,j), f i j) | i <- [0..n], j <- [0..n]]
    f 0 0 = 1
    f 0 _ = 0
    f _ 0 = 0
    f _ 1 = 1
    f i j = a ! (i-1,j-1) + j * a ! (i-1,j)

-- 2ª definición
-- =============

nParticiones2 :: [a] -> Integer
```

```haskell
nParticiones2 xs = sum [a ! (n,k) | k <- [0..n]]
  where
    n = genericLength xs
    a = array ((0,0),(n,n)) [((i,j), f i j) | i <- [0..n], j <- [0..n]]
    f 0 0 = 1
    f 0 _ = 0
    f _ 0 = 0
    f _ 1 = 1
    f i j = a ! (i-1,j-1) + j * a ! (i-1,j)

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Calcular cuántos dígitos tiene el número de
-- particiones del conjunto con 600 elementos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> length (show (nParticiones [1..600]))
--    1050
--    (0.93 secs, 675,285,608 bytes)
--    λ> length (show (nParticiones2 [1..600]))
--    1050
--    (0.92 secs, 675,287,656 bytes)
```

2

Exámenes del grupo 2

Francisco J. Martín

2.1. Examen 1 (09 de noviembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 1º examen de evaluación continua (9 de noviembre de 2018)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías                                                        --
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Dados tres números reales estrictamente positivos y
-- distintos, r, s y t, construimos una diana con tres círculos
-- concéntricos de centro el origen de coordenadas. Al disparar una
-- flecha a la diana, obtenemos 5 puntos si acertamos en el interior
-- (estricto, sin contar el borde) del círculo más pequeño; 3 puntos si
-- acertamos en el interior (estricto, sin contar el borde) del círculo
-- de tamaño medio, pero fuera  del círculo más pequeño; y 1 punto si
-- acertamos en el interior (estricto, sin contar el borde) del círculo
-- más grande, pero fuera del círculo de tamaño medio. En cualquier otro
-- caso obtenemos 0 puntos.
--
-- Definir la función
--   puntosDisparoDiana :: Double -> Double -> Double -> (Double,Double)
```

```haskell
--                          -> Int
-- tal que (puntosDisparoDiana r s t (a,b)) es la puntuación que
-- obtendríamos al acertar con una flecha en el punto de coordenadas
-- (a,b) en la diana construida con los números reales estrictamente
-- positivos y distintos, r, s y t. Por ejemplo:
--   puntosDisparoDiana 1 3 7 (0,0)    ==  5
--   puntosDisparoDiana 3 1 7 (-2,-2)  ==  3
--   puntosDisparoDiana 7 1 3 (-4,4)   ==  1
--   puntosDisparoDiana 1 7 3 (5,-5)   ==  0
--   puntosDisparoDiana 3 7 1 (1,0)    ==  3
--   puntosDisparoDiana 7 3 1 (0,-3)   ==  1
-- ---------------------------------------------------------------------------

-- 1ª solución
puntosDisparoDiana :: Double -> Double -> Double -> (Double,Double) -> Int
puntosDisparoDiana r s t (a,b)
  | d < x^2   = 5
  | d < y^2   = 3
  | d < z^2   = 1
  | otherwise = 0
  where x = minimum [r,s,t]
        z = maximum [r,s,t]
        y = (r+s+t)-(x+z)
        d = a^2+b^2

-- 2ª solución:
puntosDisparoDiana2 :: Double -> Double -> Double -> (Double,Double) -> Int
puntosDisparoDiana2 r s t (a,b)
  | d < x^2   = 5
  | d < y^2   = 3
  | d < z^2   = 1
  | otherwise = 0
  where [x,y,z] = sort [r,s,t]
        d       = a^2+b^2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Decimos que una lista es posicionalmente divisible si
-- todos sus elementos son divisibles por el número de la posición que
-- ocupan (contando desde 1). Por ejemplo, la lista [2,10,9] es
-- posicionalmente divisible pues 2 es divisible por 1, 10 es divisible
```

```haskell
-- por 2 y 9 es divisible  por 3. Por otro lado, la lista [1,3,5] no es
-- posicionalmente divisible pues aunque 1 es divisible por 1, 3 no es
-- divisible por 2.
--
-- Definir la función
--   listaPosDivisible :: [Integer] -> Bool
-- tal que (listaPosDivisible xs) se verifica si xs es una lista
-- posicionalmente divisible. Por ejemplo:
--   listaPosDivisible [1,2,3,4]    ==  True
--   listaPosDivisible [5,8,15,20]  ==  True
--   listaPosDivisible [2,10,9,8]   ==  True
--   listaPosDivisible [1,3,5]      ==  False
--   listaPosDivisible [1,6,6,6]    ==  False
--   listaPosDivisible []           ==  True
-- ---------------------------------------------------------------------------

listaPosDivisible :: [Integer] -> Bool
listaPosDivisible xs =
  and [mod x i == 0 | (x,i) <- zip xs [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una forma de aproximar el valor del número e es usando
-- la siguiente igualdad:
--
--          1         1      1      1      1      1
--         --- = 1 - ---- + ---- - ---- + ---- - ---- ...
--          e         1!     2!     3!     4!     5!
--
-- Es decir, la serie cuyo término general n-ésimo es:
--
--                   (-1)^n
--          s(n) =  --------
--                     n!
--
-- Definir la función
--    aproximaE :: Double -> Double
-- tal que (aproximaE n) es la aproximación del número e calculada con la
-- serie anterior hasta el término n-ésimo. Por ejemplo,
--    aproximaE  0  ==  1.0
--    aproximaE  1  ==  Infinity
```

```haskell
--    aproximaE  2  ==  2.0
--    aproximaE  3  ==  2.9999999999999996
--    aproximaE  4  ==  2.6666666666666666
--    aproximaE  5  ==  2.727272727272727
--    aproximaE 10  ==  2.718281657666403
--    aproximaE 15  ==  2.718281828459377
--    aproximaE 20  ==  2.718281828459044
-- ---------------------------------------------------------------------

-- 1ª definición:
aproximaE :: Double -> Double
aproximaE n =
  1 / sum [(-1)**i / product [1..i] | i <- [0..n] ]

-- 2ª definición
aproximaE2 :: Double -> Double
aproximaE2 n = 1 / aux n
  where aux :: Double -> Double
        aux 0 = 1
        aux n = (-1)**n / product [1..n] + aux (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    numeroCambiosParidad :: Integer -> Integer
-- tal que (numeroCambiosParidad n) es el número de veces que aparecen
-- en el número positivo n dos cifras consecutivas de paridad diferente
-- (una par y otra impar). Por ejemplo,
--    numeroCambiosParidad 1357  ==  0
--    numeroCambiosParidad 2468  ==  0
--    numeroCambiosParidad 1346  ==  1
--    numeroCambiosParidad 1234  ==  3
--    numeroCambiosParidad 12    ==  1
--    numeroCambiosParidad 5     ==  0
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

numeroCambiosParidad :: Integer -> Integer
numeroCambiosParidad n
```

```haskell
  | n < 10       = 0
  | odd (n1+n2) = 1 + numeroCambiosParidad (sinUltimoDigito n)
  | otherwise    = numeroCambiosParidad (sinUltimoDigito n)
  where n1 = ultimoDigito n
        n2 = penultimoDigito n

-- (ultimoDigito n) es el último dígito del número n. Por ejemplo.
--    ultimoDigito 2018  ==  8
ultimoDigito :: Integer -> Integer
ultimoDigito n = n `mod` 10

-- (penultimoDigito n) es el penúltimo dígito del número n. Por ejemplo.
--    penultimoDigito 2018  ==  1
penultimoDigito :: Integer -> Integer
penultimoDigito n = (n `div` 10) `mod` 10

-- (sinUltimoDigito n) es el número n sin el último dígito. Por ejemplo,
--    sinUltimoDigito 2018  ==  201
sinUltimoDigito :: Integer -> Integer
sinUltimoDigito n = (n `div` 10)

-- 2ª solución
-- ===========

numeroCambiosParidad2 :: Integer -> Integer
numeroCambiosParidad2 n =
  numeroCambiosParidadLista (digitos n)

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
digitos ::Integer -> [Integer]
digitos n = [read [c] | c <- show n]

-- (numeroCambiosParidadLista ns) es el número de veces que aparecen
-- en la lista ns n dos elementos consecutivos de paridad diferente
-- (una par y otra impar). Por ejemplo,
--    numeroCambiosParidadLista [1,3,4,6]  ==  1
--    numeroCambiosParidadLista [1,2,3,4]  ==  3
numeroCambiosParidadLista :: [Integer] -> Integer
numeroCambiosParidadLista (x:y:zs)
  | odd (x+y) = 1 + numeroCambiosParidadLista (y:zs)
```

```haskell
  | otherwise = numeroCambiosParidadLista (y:zs)
numeroCambiosParidadLista _ = 0

-- 3ª solución
-- ===========

numeroCambiosParidad3 :: Integer -> Integer
numeroCambiosParidad3 n =
  numeroCambiosParidadLista2 (digitos n)

-- (numeroCambiosParidadLista2 ns) es el número de veces que aparecen
-- en la lista ns n dos elementos consecutivos de paridad diferente
-- (una par y otra impar). Por ejemplo,
--    numeroCambiosParidadLista2 [1,3,4,6]  ==  1
--    numeroCambiosParidadLista2 [1,2,3,4]  ==  3
numeroCambiosParidadLista2 :: [Integer] -> Integer
numeroCambiosParidadLista2 ns =
  sum [1 | (x,y) <- zip ns (tail ns)
         , odd (x+y)]
```

2.2. Examen 2 (14 de diciembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 2º examen de evaluación continua (14 de diciembre de 2018)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. El resultado de agregar una lista consiste en construir
-- otra lista obtenida sumando elementos consecutivos de la primera. Por
-- ejemplo, el resultado de agregar la lista [1,2,3,4] es la lista
-- [3,5,7] que se obtiene sumando 1 y 2; 2 y 3; y 3 y 4.
--
-- Definir la función
--    agregarLista :: [Int] -> [Int]
-- tal que (agregarLista xs) es la lista obtenida agregando la lista xs.
-- Por ejemplo,
--    agregarLista [1,2,3,4]  ==  [3,5,7]
--    agregarLista [3,5,7]    ==  [8,12]
--    agregarLista [8,12]     ==  [20]
--    agregarLista [20]       ==  []
```

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

-- 1ª definición
agregarLista :: [Int] -> [Int]
agregarLista xs =
  zipWith (+) xs (tail xs)

-- 2ª definición
agregarLista2 :: [Int] -> [Int]
agregarLista2 xs =
  [x + y | (x,y) <- zip xs (tail xs)]

-- 3ª definición
agregarLista3 :: [Int] -> [Int]
agregarLista3 (x:y:zs) = (x+y) : agregarLista3 (y:zs)
agregarLista3 _        = []

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    incrementaDigitos :: Integer -> Integer
-- tal que (incrementaDigitos n) es el número que se obtiene
-- incrementando en una unidad todos los dígitos del número n (pasando
-- del 9 al 0). Por ejemplo,
--    incrementaDigitos 13579  ==  24680
--    incrementaDigitos  8765  ==  9876
--    incrementaDigitos   249  ==  350
--    incrementaDigitos    89  ==  90
--    incrementaDigitos    98  ==  9
--    incrementaDigitos    99  ==  0
--    incrementaDigitos     9  ==  0
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

incrementaDigitos :: Integer -> Integer
incrementaDigitos =
  digitosAnumero . map incrementaDigito . digitos
```

```
-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 325  ==  [3,2,5]
digitos :: Integer -> [Integer]
digitos n = [read [c] | c <- show n]

-- (incrementaDigito n) es el número que se obtiene incrementando en una
-- unidad el dígito n (pasando del 9 al 0). Por ejemplo,
--    incrementaDigito 3  ==  4
--    incrementaDigito 9  ==  0
incrementaDigito :: Integer -> Integer
incrementaDigito n = (n + 1) `mod` 10

-- (digitosAnumero ns) es el número correspondiente a la lista de
-- dígitos ns. Por ejemplo,
--    digitosAnumero [3,2,5]  ==  325
digitosAnumero :: [Integer] -> Integer
digitosAnumero = read . concatMap show

-- 2ª definición
-- =============

incrementaDigitos2 :: Integer -> Integer
incrementaDigitos2 n =
  read [incrementaDigito2 c | c <- show n]

-- (incrementaDigito2 c) es el dígito que se obtiene incrementando en una
-- unidad el dígito c (pasando del 9 al 0). Por ejemplo,
--    incrementaDigito '3'  ==  '4'
--    incrementaDigito '9'  ==  '0'
incrementaDigito2 :: Char -> Char
incrementaDigito2 '9' = '0'
incrementaDigito2 d   = succ d

-- 3ª definición
-- =============

incrementaDigitos3 :: Integer -> Integer
incrementaDigitos3 =
  read . map incrementaDigito2 . show
```

```haskell
-- 4ª definición
-- =============

incrementaDigitos4 :: Integer -> Integer
incrementaDigitos4 =
  read . map incrementaDigito4 . show

-- (incrementaDigito4 c) es el dígito que se obtiene incrementando en una
-- unidad el dígito c (pasando del 9 al 0). Por ejemplo,
--    incrementaDigito '3'  ==  '4'
--    incrementaDigito '9'  ==  '0'
incrementaDigito4 :: Char -> Char
incrementaDigito4 c =
  head [y | (x,y) <- zip "0123456789" "1234567890"
          , x == c]

-- ---------------------------------------------------------------------
-- Ejercicio 3. La distancia Manhattan entre dos listas de números es la
-- suma de las diferencias en valor absoluto entre los pares de
-- elementos que ocupan la misma posición, descartando los elementos
-- desparejados. Por ejemplo, la distancia Manhattan entre las listas
-- [1,5,9] y [8,7,6,4] es |1-8|+|5-7|+|9-6| = 12, pues el 4 se descarta
-- ya que está desparejado.
--
-- Definir la función
--    distanciaManhattan :: [Int] -> [Int] -> Int
-- tal que (distanciaManhattan xs ys) es la distancia Manhattan entre
-- las listas xs e ys. Por ejemplo,
--    distanciaManhattan [1,5,9] [8,7,6,4]  ==  12
--    distanciaManhattan [1,-1] [-2,2]      ==  6
--    distanciaManhattan [1,2,3] [2,1]      ==  2
--    distanciaManhattan [3,-2] [3,4,5]     ==  6
--    distanciaManhattan [] [1,2,3]         ==  0
--    distanciaManhattan [1,2] []           ==  0
-- ---------------------------------------------------------------------

-- 1ª definición
distanciaManhattan :: [Int] -> [Int] -> Int
distanciaManhattan [] _          = 0
distanciaManhattan _ []          = 0
```

```
distanciaManhattan (x:xs) (y:ys) = abs (x-y) + distanciaManhattan xs ys

-- 2ª definición
distanciaManhattan2 :: [Int] -> [Int] -> Int
distanciaManhattan2 (x:xs) (y:ys) = abs (x-y) + distanciaManhattan xs ys
distanciaManhattan2 _ _           = 0

-- 3ª definición
distanciaManhattan3 :: [Int] -> [Int] -> Int
distanciaManhattan3 xs ys =
  sum [abs (x-y) | (x,y) <- zip xs ys]

-- 4ª definición
distanciaManhattan4 :: [Int] -> [Int] -> Int
distanciaManhattan4 xs ys =
  sum (zipWith (\x y -> abs (x-y)) xs ys)

-- 5ª definición
distanciaManhattan5 :: [Int] -> [Int] -> Int
distanciaManhattan5 =
  (sum .) . zipWith ((abs .) . (-))

-- 6ª definición
distanciaManhattan6 :: [Int] -> [Int] -> Int
distanciaManhattan6 xs ys =
  foldr (\(x,y) r -> abs (x-y) + r) 0 (zip xs ys)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    ultimoMaximal :: (Int -> Int) -> [Int] -> Int
-- tal que (ultimoMaximal f xs) es el último elemento de la lista no
-- vacía xs donde la función f alcanza un valor máximo. Por ejemplo,
--   ultimoMaximal abs [4,-2,3,-4,-3]  ==  -4
--   ultimoMaximal (^2) [-2,3,2,-3]    ==  -3
--   ultimoMaximal (3-) [1,2,3,4]      ==  1
-- ---------------------------------------------------------------------

-- 1ª definición
ultimoMaximal :: (Int -> Int) -> [Int] -> Int
ultimoMaximal f xs =
```

```haskell
  last [x | x <- xs, f x == m]
  where m = maximum [f x | x <- xs]

-- 2ª definición
ultimoMaximal2 :: (Int -> Int) -> [Int] -> Int
ultimoMaximal2 f xs = aux (head xs) xs
  where aux y [] = y
        aux y (x:xs) | f x >= f y = aux x xs
                     | otherwise  = aux y xs

-- 3ª definición
ultimoMaximal3 :: (Int -> Int) -> [Int] -> Int
ultimoMaximal3 f xs =
  foldl (\y x -> if f x >= f y then x else y) (head xs) xs

-- 4ª definición
ultimoMaximal4 :: (Int -> Int) -> [Int] -> Int
ultimoMaximal4 f xs =
  foldl1 (\y x -> if f x >= f y then x else y) xs
```

2.3. Examen 3 (22 de enero de 2019)

El examen es común con el del grupo 4 (ver página 133).

2.4. Examen 4 (15 de marzo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 4º examen de evaluación continua (15 de marzo de 2019)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Librerías                                                          --
-- ---------------------------------------------------------------------

import Data.List
import System.Timeout
import Data.Array
import qualified Data.Matrix as M
import qualified Data.Set as S
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Dada una relación de orden r, se dice que el elemento
-- e2 es mayor que el elemento e1 con respecto a r si se cumple
-- (r e1 e2).
--
-- Definir la función
--    subconjuntoMaximal :: S.Set a -> (a -> a -> Bool) -> S.Set a
-- tal que (subconjuntoMaximal s r) es el subconjunto del conjunto s
-- formado por todos aquellos elementos que no tienen en el conjunto s
-- ningún elemento mayor con respecto a la relación de orden r. Por
-- ejemplo,
--     λ> subconjuntoMaximal (S.fromList [1,2,3,4]) (<)
--     fromList [4]
--     λ> subconjuntoMaximal (S.fromList [1..4]) (\x y -> even (x+y) && x<y)
--     fromList [3,4]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

subconjuntoMaximal :: S.Set a -> (a -> a -> Bool) -> S.Set a
subconjuntoMaximal s r =
  S.filter (\e1 -> S.null (S.filter (\e2 -> r e1 e2) s)) s

-- 2ª solución
-- ===========

subconjuntoMaximal2 :: S.Set a -> (a -> a -> Bool) -> S.Set a
subconjuntoMaximal2 s r =
  S.filter (\e1 -> S.null (S.filter (r e1) s)) s

-- 3ª solución
-- ===========

subconjuntoMaximal3 :: S.Set a -> (a -> a -> Bool) -> S.Set a
subconjuntoMaximal3 s r = S.filter (esMaximal s r) s

-- (esMaximal s r x) se verifica si x es maximal en s repecto de r.
esMaximal :: S.Set a -> (a -> a -> Bool) -> a -> Bool
```

```haskell
esMaximal s r x = not (any (x `r`) s)

-- ---------------------------------------------------------------------
-- Ejercicio 2. La búsqueda dicotómica es una forma eficiente de buscar
-- un elemento en una tabla cuyos elementos están ordenados. Este
-- proceso busca un elemento en un trozo de la tabla delimitado entre
-- dos índices Min y Max y actúa de la siguiente forma:
-- + Si Min y Max son iguales, entonces se comprueba si el elemento
--   buscado está en la posición Min de la tabla y se termina
-- + En caso contrario, se calcula el índice medio entre Min y Max:
--   Med = (Min+Max)/2
-- + Si el elemento buscado está en la posición Med de la tabla,
--   entonces lo hemos encontrado
-- + Si el elemento buscado es menor que el que se encuentra en la
--   posición Med de la tabla, entonces continuamos buscando en el trozo
--   de la tabla delimitado por Min y la posición anterior a Med
-- + Si el elemento buscado es mayor que el que se encuentra en la
--   posición Med de la tabla, entonces continuamos buscando en el trozo
--   de la tabla delimitado por la posición siguiente a Med y Max
-- Si se llega a una situación en la que Max < Min, entonces el elemento
-- buscado no está en la tabla.
--
-- Para comenzar la búsqueda, el valor inicial de Min es el índice más
-- pequeño de la tabla y el valor inicial de Max es el índice más grande
-- de la tabla.
--
-- Por ejemplo, para buscar el elemento 6 en una tabla de tamaño 10 que
-- contiene los números pares ordenados del 2 al 20 se procedería como
-- sigue:
-- + Comenzamos la búsqueda con Min = 1 y Max = 10
-- + Se calcula el índice medio entre Min y Max: Med = 5
-- + El elemento de la tabla que está en la posición 5 es el 10 > 6
-- + Se continúa la búsqueda con Min = 1 y Max = 4
-- + Se calcula el índice medio entre Min y Max: Med = 2
-- + El elemento de la tabla que está en la posición 2 es el 4 < 6
-- + Se continúa la búsqueda con Min = 3 y Max = 4
-- + Se calcula el índice medio entre Min y Max: Med = 3
-- + El elemento de la tabla que está en la posición 3 es el 6
-- + La búsqueda termina con éxito.
--
```

```haskell
-- Definir la función
--   busquedaDicotomica :: Ord a => Array Int a -> a -> Bool
-- tal que (busquedaDicotomica m v) realiza una búsqueda dicotómica del
-- elemento v en la tabla m cuyos elementos están ordenados en orden
-- creciente. Por ejemplo,
--   busquedadicotomica (listarray (1,10) [2,4..]) 8          ==  true
--   busquedaDicotomica (listArray (1,10) [2,4..]) 9          ==  False
--   busquedaDicotomica (listArray (1,10) [2,4..]) 0          ==  False
--   busquedaDicotomica (listArray (1,10) [2,4..]) 22         ==  False
--   busquedaDicotomica (listArray (1,10^6) [2,4..]) 123456   ==  True
--   busquedaDicotomica (listArray (1,10^6) [2,4..]) 234567   ==  False
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

busquedaDicotomica :: Ord a => Array Int a -> a -> Bool
busquedaDicotomica v x =
  busquedaDicotomicaAux v x min max
  where (min,max) = bounds v

busquedaDicotomicaAux :: Ord a => Array Int a -> a -> Int -> Int -> Bool
busquedaDicotomicaAux v x min max
  | max == min    = v ! min == x
  | max < min     = False
  | v ! med == x = True
  | x < v ! med  = busquedaDicotomicaAux v x min (med-1)
  | otherwise     = busquedaDicotomicaAux v x (med+1) max
  where med = (min+max) `div` 2

-- 2ª solución
-- ===========

busquedaDicotomica2 :: Ord a => Array Int a -> a -> Bool
busquedaDicotomica2 v x = max == min && v!min == x
  where (min,max) = until imposible reduce (bounds v)
        imposible (a,b) = b <= a
        reduce (a,b) | v!c == x  = (c,c)
                     | x < v!c   = (a,c-1)
                     | otherwise = (c+1, b)
```

```haskell
        where c = (a+b) `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--   subconjuntosDivisibles :: [Int] -> [[Int]]
-- tal que (subconjuntosDivisibles xs) es la lista de todos los
-- subconjuntos de xs en los que todos los elementos tienen un factor
-- común mayor que 1. Por ejemplo,
--   subconjuntosDivisibles []         ==  [[]]
--   subconjuntosDivisibles [1]        ==  [[]]
--   subconjuntosDivisibles [3]        ==  [[3],[]]
--   subconjuntosDivisibles [1,3]      ==  [[3],[]]
--   subconjuntosDivisibles [3,6]      ==  [[3,6],[3],[6],[]]
--   subconjuntosDivisibles [1,3,6]    ==  [[3,6],[3],[6],[]]
--   subconjuntosDivisibles [2,3,6]    ==  [[2,6],[2],[3,6],[3],[6],[]]
--   subconjuntosDivisibles [2,3,6,8]  ==
--     [[2,6,8],[2,6],[2,8],[2],[3,6],[3],[6,8],[6],[8],[]]
--   length (subconjuntosDivisibles [1..10])  ==  41
--   length (subconjuntosDivisibles [1..20])  ==  1097
--   length (subconjuntosDivisibles [1..30])  ==  33833
--   length (subconjuntosDivisibles [1..40])  ==  1056986
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

subconjuntosDivisibles :: [Int] -> [[Int]]
subconjuntosDivisibles xs = filter esDivisible (subsequences xs)

-- (esDivisible xs) se verifica si todos los elementos de xs tienen un
-- factor común mayor que 1. Por ejemplo,
--    esDivisible [6,10,22]  ==  True
--    esDivisible [6,10,23]  ==  False
esDivisible :: [Int] -> Bool
esDivisible [] = True
esDivisible xs = mcd xs > 1

-- (mcd xs) es el máximo común divisor de xs. Por ejemplo,
--    mcd [6,10,22]  ==  2
--    mcd [6,10,23]  ==  1
```

```
mcd :: [Int] -> Int
mcd = foldl1' gcd

-- 2ª solución
-- ===========

subconjuntosDivisibles2 :: [Int] -> [[Int]]
subconjuntosDivisibles2 []     = [[]]
subconjuntosDivisibles2 (x:xs) = [x:ys | ys <- yss, esDivisible (x:ys)] ++ yss
  where yss = subconjuntosDivisibles2 xs

-- 3ª solución
-- ===========

subconjuntosDivisibles3 :: [Int] -> [[Int]]
subconjuntosDivisibles3 []     = [[]]
subconjuntosDivisibles3 (x:xs) = filter esDivisible (map (x:) yss) ++ yss
  where yss = subconjuntosDivisibles3 xs

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

-- La comparación es
--    λ> length (subconjuntosDivisibles [1..21])
--    1164
--    (3.83 secs, 5,750,416,768 bytes)
--    λ> length (subconjuntosDivisibles2 [1..21])
--    1164
--    (0.01 secs, 5,400,232 bytes)
--    λ> length (subconjuntosDivisibles3 [1..21])
--    1164
--    (0.01 secs, 5,264,928 bytes)
--
--    λ> length (subconjuntosDivisibles2 [1..40])
--    1056986
--    (6.95 secs, 8,845,664,672 bytes)
--    λ> length (subconjuntosDivisibles3 [1..40])
--    1056986
--    (6.74 secs, 8,727,141,792 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejrcicio 4. Definir la función
--    matrizGirada180 :: M.Matrix a -> M.Matrix a
-- tal que (matrizGirada180 p) es la matriz obtenida girando 180 grados la
-- matriz p. Por ejemplo,
--    λ> M.fromList 4 3 [1..]
--    (  1  2  3 )
--    (  4  5  6 )
--    (  7  8  9 )
--    ( 10 11 12 )
--
--    λ> matrizGirada180 (M.fromList 4 3 [1..])
--    ( 12 11 10 )
--    (  9  8  7 )
--    (  6  5  4 )
--    (  3  2  1 )
--
--    λ> M.fromList 3 4 [1..]
--    (  1  2  3  4 )
--    (  5  6  7  8 )
--    (  9 10 11 12 )
--
--    λ> matrizGirada180 (M.fromList 3 4 [1..])
--    ( 12 11 10  9 )
--    (  8  7  6  5 )
--    (  4  3  2  1 )
-- ---------------------------------------------------------------------------

-- 1ª solución
matrizGirada180 :: M.Matrix a -> M.Matrix a
matrizGirada180 p = M.matrix m n f
  where m       = M.nrows p
        n       = M.ncols p
        f (i,j) = p M.! (m-i+1,n-j+1)

-- 2ª solución
matrizGirada180b :: M.Matrix a -> M.Matrix a
matrizGirada180b p =
  M.fromLists (reverse (map reverse (M.toLists p)))
```

```
-- 3ª solución
matrizGirada180c :: M.Matrix a -> M.Matrix a
matrizGirada180c =
  M.fromLists . reverse . map reverse . M.toLists
```

2.5. Examen 5 (12 de abril de 2019)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 5º examen de evaluación continua (12 de abril de 2019)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Librerías                                                          --
-- ---------------------------------------------------------------------

import Data.List
import qualified Data.Vector as V
import qualified Data.Matrix as M
import qualified Data.Array  as A
import System.Timeout
import I1M.Cola

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Una matriz equis es una matriz en la que hay una
-- posición (i,j) tal que todos los elementos que están fuera de las
-- diagonales que pasan por dicha posición son nulos. Por ejemplo,
--    ( 0 2 0 2 )      ( 2 0 0 0 1 )      ( 3 0 0 0 5 )
--    ( 0 0 4 0 )      ( 0 0 0 3 0 )      ( 0 4 0 2 0 )
--    ( 0 3 0 7 )      ( 0 0 1 0 0 )      ( 0 0 1 0 0 )
--    ( 5 0 0 0 )      ( 0 7 0 6 0 )
--
-- Definir la función
--    esMatrizEquis :: M.Matrix Int -> Bool
-- tal que (esMatrizEquis a) se verifica si la matriz a es una matriz
-- equis. Por ejemplo, dadas las matrices
--    m1 = M.matrix 3 3 (\(i,j) -> if (all odd [i,j]) then 1 else 0)
--    m2 = M.matrix 3 4 (\(i,j) -> i+j)
-- entonces
--    esMatrizEquis m1  ==  True
--    esMatrizEquis m2  ==  False
```

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

m1, m2 :: M.Matrix Int
m1 = M.matrix 3 3 (\(i,j) -> if (all odd [i,j]) then 1 else 0)
m2 = M.matrix 3 4 (\(i,j) -> i+j)

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

esMatrizEquisAux :: M.Matrix Int -> (Int,Int) -> Bool
esMatrizEquisAux a (f,c) =
  all (== 0) [a M.! (i,j) | i <- [1..M.nrows a]
                          , j <- [1..M.ncols a]
                          , abs (f-i) /= abs (c-j)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    matrizEquis :: Int -> Int -> (Int,Int) -> M.Matrix Int
-- tal que (matrizEquis m n f c) es la matriz equis de dimensión (m,n)
-- con respecto a la posición (f,c), en la que el valor de cada elemento
-- no nulo es la distancia en línea recta a la posición (f,c), contando
-- también esta última. Por ejemplo,
--    λ> matrizEquis 3 3 (2,2)
--    ( 2 0 2 )
--    ( 0 1 0 )
--    ( 2 0 2 )
--
--    λ> matrizEquis 4 5 (2,3)
--    ( 0 2 0 2 0 )
--    ( 0 0 1 0 0 )
--    ( 0 2 0 2 0 )
--    ( 3 0 0 0 3 )
--
--    λ> matrizEquis 5 3 (2,3)
--    ( 0 2 0 )
--    ( 0 0 1 )
--    ( 0 2 0 )
```

```haskell
--    ( 3 0 0 )
--    ( 0 0 0 )
-- ---------------------------------------------------------------------

matrizEquis :: Int -> Int -> (Int,Int) -> M.Matrix Int
matrizEquis m n p =
  M.matrix m n (\(i,j) -> generadorMatrizEquis p i j)

generadorMatrizEquis :: (Int,Int) -> Int -> Int -> Int
generadorMatrizEquis (f,c) i j
  | abs (f-i) == abs (c-j) = 1 + abs (f-i)
  | otherwise              = 0

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Sheldon Cooper tiene que pagar 10# por el aparcamiento,
-- pero sólo tiene monedas de 1#, de 2# y de 5#. Entonces dice: ”podría
-- hacer esto de 125 formas distintas y necesitaría un total de 831
-- monedas para hacerlo de todas las formas posibles”. Está contando las
-- formas de disponer las monedas en la máquina para pagar los 10# y el
-- número de monedas que necesita para hacerlas todas. Por ejemplo, si
-- tuviese que pagar 4# entonces habría 5 formas de pagar:
--    [2,2], [2,1,1], [1,2,1], [1,1,2] y [1,1,1,1]
-- y el total de monedas que se necesitan para hacerlas todas es
--    2 + 3 + 3 + 3 + 4 = 15.
--
-- Definir la función
--   distribuciones :: Integer -> Integer
-- tal que (distribuciones n) es el número de formas distintas de
-- distribuir la cantidad n como una secuencia de monedas de 1#, 2# y
-- 5#, con un valor total igual a n. Por ejemplo,
--   map distribuciones [0..10] ==  [1,1,2,3,5,9,15,26,44,75,128]
--   distribuciones 5           ==  9
--   distribuciones 10          ==  128
--   distribuciones 100         ==  91197869007632925819218
--   length (show (distribuciones 1000))  ==  232
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========
```

```haskell
distribuciones1 :: Integer -> Integer
distribuciones1 = genericLength . formas

-- (formas n) es la lista de las formas distintas de distribuir la
-- cantidad n como una secuencia de monedas de 1#, 2# y 5#, con un valor
-- total igual a n. Por ejemplo,
--    λ> formas 4
--    [[1,1,1,1],[1,1,2],[1,2,1],[2,1,1],[2,2]]
formas :: Integer -> [[Integer]]
formas n | n < 0     = []
         | n == 0    = [[]]
         | otherwise = [k:xs | k <- [1,2,5], xs <- formas (n-k)]

-- 2ª solución
-- ===========

distribuciones2 :: Integer -> Integer
distribuciones2 0 = 1
distribuciones2 1 = 1
distribuciones2 2 = 2
distribuciones2 3 = 3
distribuciones2 4 = 5
distribuciones2 n = sum [distribuciones2 (n-k) | k <- [1,2,5]]

-- 3ª solución
-- ===========

distribuciones3 :: Integer -> Integer
distribuciones3 n = v A.! n
  where v = A.array (0,n) [(i, f i) | i <- [0..n]]
        f 0 = 1
        f 1 = 1
        f 2 = 2
        f 3 = 3
        f 4 = 5
        f m = sum [v A.! (m-k) | k <- [1,2,5]]

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

```haskell
--    λ> distribuciones1 26
--    651948
--    (2.72 secs, 2,650,412,856 bytes)
--    λ> distribuciones2 26
--    651948
--    (0.30 secs, 153,096,224 bytes)
--    λ> distribuciones3 26
--    651948
--    (0.00 secs, 163,744 bytes)
--
--    λ> distribuciones2 30
--    5508222
--    (2.46 secs, 1,292,545,008 bytes)
--    λ> distribuciones3 30
--    5508222
--    (0.00 secs, 172,048 bytes)

-- En lo que sigue se usará la 3ª definición
distribuciones :: Integer -> Integer
distribuciones = distribuciones3

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Con la definición del apartado anterior, evaluar (en
-- menos de 2 segundos), la siguiente expresión para que de un valor
-- distinto de Nothing
--    timeout (2*10^6) (return $! (length (show (distribuciones 100000))))
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> timeout (2*10^6) (return $! (length (show (distribuciones 100000))))
--    Just 23170
--    (0.90 secs, 1,143,930,488 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    cuentaMonedas :: Integer -> Integer
-- tal que (cuentaMonedas n) es el número de monedas que le hacen falta
-- a Sheldon Cooper para construir todas las distribuciones de la
-- cantidad n como una secuencia de monedas de 1#, 2# y 5#, con un valor
-- total igual a n. Por ejemplo,
```

```haskell
--    map cuentaMonedas1 [0..10] == [0,1,3,7,15,31,62,122,235,447,841]
--    cuentaMonedas 5            ==  31
--    cuentaMonedas 10           ==  841
--    cuentaMonedas 100          ==  5660554507743281845750870
--    length (show (cuentaMonedas 1000))  ==  235
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

cuentaMonedas1 :: Integer -> Integer
cuentaMonedas1 n = sum (map genericLength (formas n))


-- 2ª definición
-- =============

cuentaMonedas2 :: Integer -> Integer
cuentaMonedas2 0 = 0
cuentaMonedas2 1 = 1
cuentaMonedas2 2 = 3
cuentaMonedas2 3 = 7
cuentaMonedas2 4 = 15
cuentaMonedas2 n =
  cuentaMonedas2 (n-1) + cuentaMonedas2 (n-2) + cuentaMonedas2 (n-5) +
  distribuciones n

-- 3ª definición
-- =============

cuentaMonedas3 :: Integer -> Integer
cuentaMonedas3 n = v A.! n
  where v = A.array (0,n) [(i, f i) | i <- [0..n]]
        f 0 = 0
        f 1 = 1
        f 2 = 3
        f 3 = 7
        f 4 = 15
        f m = sum [v A.! (m-k) | k <- [1,2,5]] + distribuciones m
```

```haskell
-- 4ª definición
-- =============

cuentaMonedas4 :: Integer -> Integer
cuentaMonedas4 n =  c A.! n
  where
    d = A.array (0,n) [(i, f i) | i <- [0..n]]
    f 0 = 1
    f 1 = 1
    f 2 = 2
    f 3 = 3
    f 4 = 5
    f m = sum [d A.! (m-k) | k <- [1,2,5]]
    c = A.array (0,n) [(i, g i) | i <- [0..n]]
    g 0 = 0
    g 1 = 1
    g 2 = 3
    g 3 = 7
    g 4 = 15
    g m = sum [c A.! (m-k) | k <- [1,2,5]] +
          sum [d A.! (m-k) | k <- [1,2,5]]

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

--    λ> cuentaMonedas1 25
--    6050220
--    (3.81 secs, 1,980,897,632 bytes)
--    λ> cuentaMonedas2 25
--    6050220
--    (0.79 secs, 452,597,096 bytes)
--    λ> cuentaMonedas3 25
--    6050220
--    (0.00 secs, 596,400 bytes)
--    λ> cuentaMonedas4 25
--    6050220
--    (0.00 secs, 228,800 bytes)
--
--    λ> cuentaMonedas2 30
--    104132981
```

```haskell
--    (10.94 secs, 6,518,022,552 bytes)
--    λ> cuentaMonedas3 30
--    104132981
--    (0.00 secs, 821,976 bytes)
--    λ> cuentaMonedas4 30
--    104132981
--    (0.00 secs, 256,160 bytes)
--
--    λ> length (show (cuentaMonedas3 2000))
--    467
--    (4.84 secs, 3,715,286,088 bytes)
--    λ> length (show (cuentaMonedas4 2000))
--    467
--    (0.04 secs, 11,454,248 bytes)

-- En lo que sigue usaremos la 4ª definición
cuentaMonedas :: Integer -> Integer
cuentaMonedas = cuentaMonedas4

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Con la definición del apartado anterior, evaluar (en
-- menos de 10 segundos), la siguiente expresión para que de un valor
-- distinto de Nothing
--    timeout (10^7) (return $! (length (show (cuentaMonedas 100000))))
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> timeout (10^7) (return $! (length (show (cuentaMonedas 100000))))
--    Just 23175
--    (2.21 secs, 3,888,064,112 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    reduceCola :: (Eq a) => Cola a -> Cola a
-- tal que (reduceCola c) es la cola obtenida eliminando los elementos
-- consecutivos repetidos de la cola c. Por ejemplo,
--    λ> c = foldr inserta vacia [1,1,2,2,2,3,4,4,2,2,3,3]
--    λ> c
--    C [3,3,2,2,4,4,3,2,2,2,1,1]
--    λ> reduceCola c
```

```haskell
--     C [3,2,4,3,2,1]
-- ---------------------------------------------------------------------

reduceCola :: Eq a => Cola a -> Cola a
reduceCola = aux vacia
  where aux r c | esVacia c  = r
                | esVacia rc = inserta pc r
                | pc == sc   = aux r rc
                | otherwise  = aux (inserta pc r) rc
          where rc = resto c
                pc = primero c
                sc = primero rc
```

2.6. Examen 6 (10 de junio de 2019)

El examen es común con el del grupo 3 (ver página 121).

2.7. Examen 7 (02 de julio de 2019)

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

2.8. Examen 8 (11 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 59).

3

Exámenes del grupo 3

María J. Hidalgo

3.1. Examen 1 (09 de noviembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 1º examen de evaluación continua (2 de noviembre de 2017)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías                                                        --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La suma de la serie geométrica de razón r cuyo primer
-- elemento es y es
--      1 + r + r² + r³ + ...

-- Definir la función
--    sumaSG :: Double -> Double -> Double
-- tal que (sumaSG r n) es la suma de los n+1 primeros términos de dicha
-- serie. Por ejemplo,
--    sumaSG 2 0        ==  1.0
--    sumaSG 2 1        ==  3.0
--    sumaSG 2 2        ==  7.0
--    sumaSG (1/2) 100  ==  2.0
```

```haskell
--    sumaSG (1/3) 100  ==  1.5
--    sumaSG (1/4) 100  ==  1.3333333333333333
--
-- Indicación: usar la función (**) para la potencia.
-- ---------------------------------------------------------------------

sumaSG :: Double -> Double -> Double
sumaSG r n = sum [r**k | k <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. La serie converge cuando |r| < 1 y su límite es
-- 1/(1-r).
--
-- Definir la función
--    errorSG :: Double -> Double -> Double
-- tal que (errorSG r x) es el menor número de términos de la serie
-- anterior necesarios para obtener su límite con un error menor que
-- x. Por ejemplo,
--    errorSG (1/2) 0.001 == 10.0
--    errorSG (1/4) 0.001 == 5.0
--    errorSG (1/8) 0.001 == 3.0
--    errorSG (1/8) 1e-6  == 6.0
-- ---------------------------------------------------------------------

errorSG :: Double -> Double -> Double
errorSG r x = head [m | m <- [0..]
                      , abs (sumaSG r m - y) < x]
  where y = 1/(1-r)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    todosDiferentes :: Eq a => [a] -> Bool
-- tal que (todosDiferentes xs) se verifica si todos los elementos de la
-- lista xs son diferentes. Por ejemplo,
--    todosDiferentes [1..20]         == True
--    todosDiferentes (7:[1..20])     == False
--    todosDiferentes "Buenas"        == True
--    todosDiferentes "Buenas tardes" == False
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición:
todosDiferentes :: Eq a => [a] -> Bool
todosDiferentes (x:xs) = x `notElem` xs && todosDiferentes xs
todosDiferentes _      = True

-- 2ª definición:
todosDiferentes2 :: Eq a => [a] -> Bool
todosDiferentes2 xs = nub xs == xs

-- Equivalencia:
prop_todosDiferentes :: [Int] -> Bool
prop_todosDiferentes xs =
  todosDiferentes xs == todosDiferentes2 xs

-- Comprobación:
--    λ> quickCheck prop_todosDiferentes
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una lista se denomina "cuadrada" si se puede obtener
-- concatenando dos copias de una misma lista. Por ejemplo, "abab" y
-- "aa" son listas cuadradas pero "aaa" y "abba" no lo son.
--
-- Definir la función
--    esCuadrada :: (Eq a, Ord a) => [a] -> Bool
-- tal que (esCuadrada xs) se verifica si xs es una lista cuadrada.
-- Por ejemplo,
--    esCuadrada "aa"   == True
--    esCuadrada "aaa"  == False
--    esCuadrada "abab" == True
--    esCuadrada "abba" == False
-- ---------------------------------------------------------------------

esCuadrada :: (Eq a, Ord a) => [a] -> Bool
esCuadrada xs | odd m      = False
              | otherwise  = as == bs
  where m = length xs
        n = m `div` 2
        (as, bs) = splitAt n xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Un niño quiere subir saltando una escalera. Con cada
-- salto que da puede subir 1, 2 o 3 peldaños. Por ejemplo, si la
-- escalera tiene 3 peldaños la puede subir de 4 formas distintas:
--    1, 1, 1
--    1, 2
--    2, 1
--    3
--
-- Definir la función
--    numeroFormas :: Int -> Int
-- tal que (numeroFormas n) es el número de formas en que puede subir
-- una escalera de n peldaños. Por ejemplo,
--    numeroFormas 3  == 4
--    numeroFormas 4  == 7
--    numeroFormas 10 == 274
-- ---------------------------------------------------------------------

numeroFormas :: Int -> Int
numeroFormas 1 = 1
numeroFormas 2 = 2
numeroFormas 3 = 4
numeroFormas n =
  numeroFormas (n-1) + numeroFormas (n-2) + numeroFormas (n-3)
```

3.2. Examen 2 (14 de diciembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 2º examen de evaluación continua (14 de diciembre de 2018)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías                                                        --
-- ---------------------------------------------------------------------

import Data.List
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1. Representamos una matriz como una lista de listas, en la
```

```haskell
-- que cada lista es una fila de la matriz. Por ejemplo, las matrices
--    ( 1, 2, 3, 4 )      ( 1, 2, 3, 4, 5 )
--    ( 0, 9, 2, 5 )      ( 0, 9, 2, 5, 6 )
--    ( 7, 8,-1, 3 )      ( 7, 8,-1, 3, 7 )
--    ( 4, 7, 1, 0 )      ( 4, 7, 1, 0, 8 )
--                        ( 0, 1, 0, 2, 0 )
-- las representamos mediante
--    m1 = [[1, 2, 3, 4],
--          [0, 9, 2, 5],
--          [7, 8,-1, 3],
--          [4, 7, 1, 0]]
--
--    m2 = [[1,2,3,4,5],
--          [0,9,2,5,6],
--          [7,8,-1,3,7],
--          [4,7,1,0,8],
--          [0,1,0,2,0]]
--
-- Definir la función
--    columnas :: [[a]] -> [[a]]
-- tal que (columnas m) es la lista de las columnas de la matriz. Por
-- ejemplo,
--    λ> columnas m1
--    [[1,0,7,4],[2,9,8,7],[3,2,-1,1],[4,5,3,0]]
--    λ> columnas m2
--    [[1,0,7,4,0],[2,9,8,7,1],[3,2,-1,1,0],[4,5,3,0,2],[5,6,7,8,0]]
-- ---------------------------------------------------------------------

m1, m2 :: [[Int]]
m1 = [[1, 2, 3, 4],
      [0, 9, 2, 5],
      [7, 8,-1, 3],
      [4, 7, 1, 0]]
m2 = [[1,2,3,4,5],
      [0,9,2,5,6],
      [7,8,-1,3,7],
      [4,7,1,0,8],
      [0,1,0,2,0]]

-- 1ª solución
```

```
columnas :: [[a]] -> [[a]]
columnas []     = []
columnas ([]:_) = []
columnas xss    = map head xss : columnas (map tail xss)

-- 2ª solución
columnas2 :: [[a]] -> [[a]]
columnas2 = transpose

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir las funciones
--    esPrimoSumaDeDosPrimos :: Integer -> Bool
--    primosSumaDeDosPrimos :: [Integer]
-- tales que
-- + (esPrimoSumaDeDosPrimos x) se verifica si x es un número primo que
--   se puede escribir como la suma de dos números primos. Por ejemplo,
--      esPrimoSumaDeDosPrimos 19        ==  True
--      esPrimoSumaDeDosPrimos 20        ==  False
--      esPrimoSumaDeDosPrimos 23        ==  False
--      esPrimoSumaDeDosPrimos 18409541  ==  False
-- + primosSumaDeDosPrimos es la lista de los números primos que se
--   pueden escribir como la suma de dos números primos. Por ejemplo,
--      take 7 primosSumaDeDosPrimos    == [5,7,13,19,31,43,61]
--      primosSumaDeDosPrimos !! (10^4) == 1261081
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

esPrimoSumaDeDosPrimos :: Integer -> Bool
esPrimoSumaDeDosPrimos x =
  (x == last ps) && not (null [p | p <- ps, isPrime (x-p)])
  where ps = takeWhile (<=x) primes

primosSumaDeDosPrimos :: [Integer]
primosSumaDeDosPrimos = filter esPrimoSumaDeDosPrimos primes

-- 2ª solución
-- ===========
```

```haskell
-- Teniendo en cuenta que si se suman dos primos ambos distintos de 2
-- nunca puede ser primo (porque sería par).

esPrimoSumaDeDosPrimos2 :: Integer -> Bool
esPrimoSumaDeDosPrimos2 x = isPrime x && isPrime (x - 2)

primosSumaDeDosPrimos2 :: [Integer]
primosSumaDeDosPrimos2 = [x | x <- primes, isPrime (x - 2)]

-- 3ª solución
-- ===========

primosSumaDeDosPrimos3 :: [Integer]
primosSumaDeDosPrimos3 =
  [y | (x,y) <- zip primes (tail primes), y == x + 2]

esPrimoSumaDeDosPrimos3 :: Integer -> Bool
esPrimoSumaDeDosPrimos3 x =
  x == head (dropWhile (<x) primosSumaDeDosPrimos3)

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

--    λ> primosSumaDeDosPrimos2 !! 300
--    17293
--    (0.10 secs, 30,061,168 bytes)
--    λ> primosSumaDeDosPrimos3 !! 300
--    17293
--    (0.06 secs, 10,797,448 bytes)
--
--    λ> primosSumaDeDosPrimos2 !! (5*10^3)
--    557731
--    (2.46 secs, 1,180,627,296 bytes)
--    λ> primosSumaDeDosPrimos3 !! (5*10^3)
--    557731
--    (0.99 secs, 362,595,072 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios se puede representar mediante el
-- siguiente tipo
```

```haskell
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
--    nodosNivelesImpar :: Arbol a -> [a]
-- tal que (nodosNivelesImpar ar) es la lista de los nodos de nivel
-- impar de ar. Por ejemplo, si
--   ejA = N 10 (N (-2) (N 8 (H 0) (H 1))
--                      (H (-4)))
--              (N 6 (H 7) (H 5))
-- entonces
--   nodosNivelesImpar ejA == [-2,6,0,1]
-- ---------------------------------------------------------------------

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

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

nodosNivelesImpar :: Arbol a -> [a]
nodosNivelesImpar ar =
  concat $ takeWhile (not . null) [nivel k ar | k <-[1,3..]]

-- ---------------------------------------------------------------------
-- Ejercicio 4. La sucesión de Loomis generada por un número entero
-- positivo x es la sucesión cuyos términos se definen por
-- + sucL(0) es x
-- + sucL(n) es la suma de sucL(n-1) y el producto de los dígitos no
--   nulos de sucL(n-1)
--
-- Los primeros términos de las primeras sucesiones de Loomis son
-- + Generada por 1: 1, 2, 4, 8, 16, 22, 26, 38, 62, 74, 102, 104, 108,
-- + Generada por 2: 2, 4, 8, 16, 22, 26, 38, 62, 74, 102, 104, 108,
-- + Generada por 3: 3, 6, 12, 14, 18, 26, 38, 62, 74, 102, 104, 108,
```

```haskell
-- + Generada por 4: 4, 8, 16, 22, 26, 38, 62, 74, 102, 104, 108, 116,
-- + Generada por 5: 5, 10, 11, 12, 14, 18, 26, 38, 62, 74, 102, 104,
--
-- Definir la función
--    sucL :: Integer -> [Integer]
-- tal que (sucL x) es la sucesión de Loomis generada por x. Por
-- ejemplo,
--    λ> take 15 (sucL 1)
--    [1,2,4,8,16,22,26,38,62,74,102,104,108,116,122]
--    λ> take 15 (sucL 2)
--    [2,4,8,16,22,26,38,62,74,102,104,108,116,122,126]
--    λ> take 15 (sucL 3)
--    [3,6,12,14,18,26,38,62,74,102,104,108,116,122,126]
--    λ> take 15 (sucL 20)
--    [20,22,26,38,62,74,102,104,108,116,122,126,138,162,174]
--    λ> take 15 (sucL 100)
--    [100,101,102,104,108,116,122,126,138,162,174,202,206,218,234]
--    λ> sucL 1 !! (2*10^5)
--    235180736652
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

sucL :: Integer -> [Integer]
sucL x = map (loomis x) [0..]

-- (loomis x n) es el n-ésimo término de la sucesión de Loomis generada
-- por x. Por ejemplo,
--    loomis 2 3  ==  16
--    loomis 3 2  ==  12
loomis :: Integer -> Integer -> Integer
loomis x 0 = x
loomis x n = y + productoDigitosNoNulos y
  where y = loomis x (n-1)

-- (productoDigitosNoNulos n) es el producto de los dígitos no nulos de
-- n. Por ejemplo,
--    productoDigitosNoNulos 3005040  ==  60
productoDigitosNoNulos :: Integer -> Integer
```

```haskell
productoDigitosNoNulos = product . digitosNoNulos

-- (digitosNoNulos x) es la lista de los dígitos no nulos de x. Por
-- ejemplo,
--    digitosNoNulos 3005040  ==  [3,5,4]
digitosNoNulos :: Integer -> [Integer]
digitosNoNulos x = [read [c] | c <- show x, c /= '0']

-- 2ª definición
-- =============

sucL2 :: Integer -> [Integer]
sucL2 = iterate siguienteL
  where siguienteL y = y + productoDigitosNoNulos y

-- 3ª definición
-- =============

sucL3 :: Integer -> [Integer]
sucL3 = iterate (\y -> y + productoDigitosNoNulos y)

-- 4ª definición
-- =============

sucL4 :: Integer -> [Integer]
sucL4 = iterate ((+) <*> productoDigitosNoNulos)
```

3.3. Examen 3 (22 de enero de 2019)

El examen es común con el del grupo 1 (ver página 133).

3.4. Examen 4 (15 de marzo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 4º examen de evaluación continua (15 de marzo de 2019)
-- ---------------------------------------------------------------------

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

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

import Data.List
import Data.Maybe
import Data.Array

-- ---------------------------------------------------------------------
-- Ejercicio 1. El problema del número perdido consiste en, dada una
-- lista de números consecutivos (creciente o decreciente) en la que
-- puede faltar algún número, hacer lo siguiente:
-- + si falta un único número z, devolver Just z
-- + si no falta ninguno, devolver Nothing
--
-- Definir la función
--    numeroPerdido :: [Int] -> Maybe Int
-- tal que (numeroPerdido xs) es la solución del problema del número
-- perdido en la lista xs. Por ejemplo,
--    numeroPerdido [7,6,4,3]                     == Just 5
--    numeroPerdido [1,2,4,5,6]                   == Just 3
--    numeroPerdido [6,5..3]                      == Nothing
--    numeroPerdido [1..6]                        == Nothing
--    numeroPerdido ([5..10^6] ++ [10^6+2..10^7]) == Just 1000001
-- ---------------------------------------------------------------------

-- 1ª solución
numeroPerdido :: [Int] -> Maybe Int
numeroPerdido (x:y:xs)
  | abs (y - x) == 1 = numeroPerdido (y:xs)
  | otherwise        = Just (div (x+y) 2)
numeroPerdido _      = Nothing

-- 2ª solución
numeroPerdido2 :: [Int] -> Maybe Int
numeroPerdido2 xs = aux z (z:zs)
  where (z:zs) = sort xs
        aux _ [] = Nothing
        aux y (x:xs) | y == x    = aux (y+1) xs
                     | otherwise = Just y

-- 3ª solución
```

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

numeroPerdido3 :: [Int] -> Maybe Int
numeroPerdido3 xs =
  listToMaybe [(a+b) `div` 2 | (a:b:_) <- tails xs, abs(a-b) /= 1]

-- ---------------------------------------------------------------------
-- Ejercicio 2. 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           1             1
--       / \         / \           /|\
--      2   3      -2   3         / | \
--     / \          |           -2  7  3
--    4   5        -4           / \
--                             4   5
-- se representan por
--    ej1, ej2, ej3 :: Arbol Int
--    ej1 = N (-1) [N 2 [N 4 [], N 5 []], N 3 []]
--    ej2 = N 1 [N (-2) [N (-4) []], N 3 []]
--    ej3 = N 1 [N (-2) [N 4 [], N 5 []], N 7 [], N 3 []]
--
-- Definir la función
--    todasDesdeAlguno :: (a -> Bool) -> Arbol a -> Bool
-- tal que (todasDesdeAlguno p ar) se verifica si para toda rama existe un
-- elemento a partir del cual todos los elementos de la rama verifican
-- la propiedad p. Por ejemplo,
--    todasDesdeAlguno (>0) ej1 == True
--    todasDesdeAlguno (>0) ej2 == False
--    todasDesdeAlguno (>0) ej3 == True
-- ---------------------------------------------------------------------

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

ej1, ej2, ej3 :: Arbol Int
ej1 = N (-1) [N 2 [N 4 [], N 5 []], N 3 []]
ej2 = N 1 [N (-2) [N (-4) []], N 3 []]
```

```haskell
ej3 = N 1 [N (-2) [N 4 [], N 5 []], N 7 [], N 3 []]

-- 1ª solución
-- ===========

todasDesdeAlguno :: (b -> Bool) -> Arbol b -> Bool
todasDesdeAlguno p a = all (desdeAlguno p) (ramas a)

-- (desdeAlguno p xs) se verifica si la propiedad xs tiene un elementemo
-- a partir del cual todos los siguientes cumplen la propiedad p. Por
-- ejemplo,
--    desdeAlguno (>0) [-1,2,4]   ==  True
--    desdeAlguno (>0) [1,-2,-4]  ==  False
--    desdeAlguno (>0) [1,-2,4]   ==  True

-- 1ª definición de desdeAlguno
desdeAlguno1 :: (a -> Bool) -> [a] -> Bool
desdeAlguno1 p xs =
  not (null (takeWhile p (reverse xs)))

-- 2ª definición de desdeAlguno
desdeAlguno2 :: (a -> Bool) -> [a] -> Bool
desdeAlguno2 p xs = any (all p) (init (tails xs))

-- Comparación de eficiencia:
--    λ> desdeAlguno1 (>10^7) [1..1+10^7]
--    True
--    (4.36 secs, 960,101,896 bytes)
--    λ> desdeAlguno2 (>10^7) [1..1+10^7]
--    True
--    (5.62 secs, 3,600,101,424 bytes)

-- Usaremos la 1ª definición de desdeAlguno
desdeAlguno :: (a -> Bool) -> [a] -> Bool
desdeAlguno = desdeAlguno1

-- (ramas a) es la lista de las ramas de a. Por ejemplo,
--    ramas ej1  ==  [[-1,2,4],[-1,2,5],[-1,3]]
--    ramas ej2  ==  [[1,-2,-4],[1,3]]
--    ramas ej3  ==  [[1,-2,4],[1,-2,5],[1,7],[1,3]]
```

```haskell
ramas :: Arbol a -> [[a]]
ramas (N x []) = [[x]]
ramas (N x as) = map (x:) (concatMap ramas as)

-- 2ª solución
-- ===========

todasDesdeAlguno2 :: (b -> Bool) -> Arbol b -> Bool
todasDesdeAlguno2 p (N x []) = p x
todasDesdeAlguno2 p (N _ as) = all (todasDesdeAlguno2 p) as

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. El fichero sucPrimos.txt http://bit.ly/2J49Qjl
-- contiene una sucesión de números primos escritos uno a continuación
-- de otro. Por ejemplo,
--    λ> xs <- readFile "sucPrimos.txt"
--    λ> take 50 xs
--    "23571113171923293137414347535961677173798389971011"
--    λ> take 60 xs
--    "235711131719232931374143475359616771737983899710110310710911"
--
-- Definir la función
--    posicion :: String -> IO (Maybe Int)
-- tal que (posicion n) es (Just k) si k es la posición de n en la
-- sucesión almacenada en el fichero sucPrimos.txt y Nothing si n no
-- ocurre en dicha sucesión. Por ejemplo,
--    posicion 1959  == Just 5740
--    posicion 19590 == Nothing
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

posicion :: Int -> IO (Maybe Int)
posicion n = do
  ds <- readFile "sucPrimos.txt"
  return (posicionEnLista ds (show n))

posicionEnLista :: Eq a => [a] -> [a] -> Maybe Int
posicionEnLista xs ys = aux xs 0
```

```haskell
  where aux [] _ = Nothing
        aux (x:xs) n | ys `isPrefixOf` (x:xs) = Just n
                     | otherwise              = aux xs (n+1)

-- 2ª definición
-- =============

posicion2 :: Int -> IO (Maybe Int)
posicion2 n = do
  ds <- readFile "sucPrimos.txt"
  return (findIndex (show n `isPrefixOf`) (tails ds))

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    posicionInteractivo ::  IO ()
-- tal que solicite que se introduzca por teclado un número sin ceros y
-- escriba como respuesta la posición de dicho número. Por ejemplo,
--    λ> posicionInteractivo
--    Escribe un número sin ceros:
--    1959
--    La posición es: 5740
--    λ> posicionInteractivo
--    Escribe un número sin ceros:
--    12345
--    No aparece
-- ---------------------------------------------------------------------

posicionInteractivo ::  IO ()
posicionInteractivo = do
  putStrLn "Escribe un número sin ceros: "
  ds <- readFile "sucPrimos.txt"
  c <- getLine
  let n = read c :: Int
  p <- posicion n
  if isJust p then putStrLn ("La posición es: " ++ show (fromJust p))
            else putStrLn "No aparece"

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. El triángulo de Euler se construye a partir de las
-- siguientes relaciones
```

```haskell
--    A(n,1) = A(n,n) = 1
--    A(n,m) = (n-m)A(n-1,m-1) + (m+1)A(n-1,m).
-- Sus primeros términos son
--    1
--    1 1
--    1 4   1
--    1 11  11    1
--    1 26  66    26    1
--    1 57  302   302   57     1
--    1 120 1191  2416  1191   120   1
--    1 247 4293  15619 15619  4293  247   1
--    1 502 14608 88234 156190 88234 14608 502 1
--
-- Definir las siguientes funciones:
--   numeroEuler        :: Integer -> Integer -> Integer
--   filaTrianguloEuler :: Integer -> [Integer]
--   trianguloEuler     :: [[Integer]]
-- tales que
-- + (numeroEuler n k) es el número de Euler A(n,k). Por ejemplo,
--      numeroEuler 8 3  == 15619
--      numeroEuler 20 6 == 21598596303099900
--      length (show (numeroEuler 1000 500)) == 2567
-- + (filaTrianguloEuler n) es la n-ésima fila del triángulo de
--   Euler. Por ejemplo,
--      filaTrianguloEuler 7  ==  [1,120,1191,2416,1191,120,1]
--      filaTrianguloEuler 8  ==  [1,247,4293,15619,15619,4293,247,1]
--      length (show (maximum (filaTrianguloEuler 1000)))  ==  2567
-- + trianguloEuler es la lista con las filas del triángulo de Euler
--      λ> take 6 trianguloEuler
--      [[1],[1,1],[1,4,1],[1,11,11,1],[1,26,66,26,1],[1,57,302,302,57,1]]
--      λ> length (show (maximum (trianguloEuler !! 999)))
--      2567
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

trianguloEuler :: [[Integer]]
trianguloEuler = iterate siguiente [1]
```

```haskell
-- (siguiente xs) es la fila siguiente a la xs en el triángulo de
-- Euler. Por ejemplo,
--    λ> siguiente [1]
--    [1,1]
--    λ> siguiente it
--    [1,4,1]
--    λ> siguiente it
--    [1,11,11,1]
siguiente :: [Integer] -> [Integer]
siguiente xs = zipWith (+) us vs
  where n = genericLength xs
        us = zipWith (*) (0:xs) [n+1,n..1]
        vs = zipWith (*) (xs++[0]) [1..n+1]

filaTrianguloEuler :: Integer -> [Integer]
filaTrianguloEuler n = trianguloEuler `genericIndex` (n-1)

numeroEuler :: Integer -> Integer -> Integer
numeroEuler n k = filaTrianguloEuler n `genericIndex` k

-- 2ª solución
-- ===========

numeroEuler2 :: Integer -> Integer -> Integer
numeroEuler2 n 0 = 1
numeroEuler2 n m
  | n == m    = 0
  | otherwise = (n-m) * numeroEuler2 (n-1) (m-1) + (m+1) * numeroEuler2 (n-1) m

filaTrianguloEuler2 :: Integer -> [Integer]
filaTrianguloEuler2 n = map (numeroEuler2 n) [0..n-1]

trianguloEuler2 :: [[Integer]]
trianguloEuler2 = map filaTrianguloEuler2 [1..]

-- 3ª solución
-- ===========

numeroEuler3 :: Integer -> Integer -> Integer
numeroEuler3 n k = matrizEuler n k ! (n,k)
```

```haskell
-- (matrizEuler n m) es la matriz de n+1 filas y m+1 columnsa formada
-- por los números de Euler. Por ejemplo,
--    λ> [[matrizEuler 6 6 ! (i,j) | j <- [0..i-1]] | i <- [1..6]]
--    [[1],[1,1],[1,4,1],[1,11,11,1],[1,26,66,26,1],[1,57,302,302,57,1]]
matrizEuler :: Integer -> Integer -> Array (Integer,Integer) Integer
matrizEuler n m = q
  where q = array ((0,0),(n,m)) [((i,j), f i j) | i <- [0..n], j <- [0..m]]
        f i 0 = 1
        f i j
          | i == j    = 0
          | otherwise = (i-j) * q!(i-1,j-1) + (j+1)* q!(i-1,j)

filaTrianguloEuler3 :: Integer -> [Integer]
filaTrianguloEuler3 n = map (numeroEuler3 n) [0..n-1]

trianguloEuler3 :: [[Integer]]
trianguloEuler3 = map filaTrianguloEuler3 [1..]

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

-- La comparación es
--    λ> numeroEuler 22 11
--    301958232385734088196
--    (0.01 secs, 118,760 bytes)
--    λ> numeroEuler2 22 11
--    301958232385734088196
--    (3.96 secs, 524,955,384 bytes)
--    λ> numeroEuler3 22 11
--    301958232385734088196
--    (0.01 secs, 356,296 bytes)
--
--    λ> length (show (numeroEuler 800 400))
--    1976
--    (0.01 secs, 383,080 bytes)
--    λ> length (show (numeroEuler3 800 400))
--    1976
--    (2.13 secs, 508,780,696 bytes)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.2 Definir la función
--    propFactorial:: Integer -> Bool
-- para expresar que la suma de la n-ésima fila del triángulo de Eules es
-- el factorial de n. Y calcular (propFactorial 50).
-- ---------------------------------------------------------------------

propFactorial :: Integer -> Bool
propFactorial n =
  and [sum (filaTrianguloEuler m) == product [1..m] | m <- [1..n]]

-- El cálculo es
--    λ> propFactorial 50
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 5. El valor de a^b módulo c es el resto de la división
-- entera de a^b entre c.
--
-- Por ejemplo, si a = 179, b = 12, c = 13, una forma de calcular la
-- exponenciación modular sería directamente: (179^12) `mod` 13, con
-- resultado 1.  Ahora bien, el cálculo directo no es eficiente para
-- valores grandes, por ejemplo cuando a = 4, b = 411728896 y
-- c = 1000000000.
--
-- Un algoritmo más eficiente para calcular la exponenciación modular de
-- a^b módulo c es el siguiente:
-- + Se comienza con x(0) = 1, y(0) = a y z(0) = b.
-- + Hasta que z(n) sea 0 se realiza lo siguiente:
--   + Si z(n) es par,   entonces x(n+1) = resto de x(n) entre c
--                                y(n+1) = resto de y(n)^2 entre c
--                                z(n+1) = z(n)/2
--   + Si z(n) es impar, entonces x(n+1) = resto de (x(n) * y(n)) entre c
--                                y(n+1) = y(n)
--                                z(n+1) = z(n) - 1
-- + Finalmente, el resultado es el valor de x(n).
--
-- El cálculo de 179^12 (mod 13) mediante el algoritmo anterior es:
--    +---+-----+-----+
--    | x |  y  |  z  |
```

```
--    +---+-----+-----+
--    | 1 | 179 | 12  |
--    | 1 |   9 |  6  |
--    | 1 |   3 |  3  |
--    | 3 |   3 |  2  |
--    | 3 |   9 |  1  |
--    | 1 |   9 |  0  |
--    +---+-----+-----+
--
-- Definir la función
--    expModulo :: Integer -> Integer -> Integer -> Integer
-- tal que (expModulo a b c) sea el exponente modular de a^b, módulo c,
-- con el algoritmo anterior. Por ejemplo,
--    expModulo 179 12 13              == 1
--    expModulo 179 13 10              == 9
--    expModulo 13789 722341 2345      == 2029
--    expModulo 4 411728896 1000000000 == 411728896
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

expModulo :: Integer -> Integer -> Integer -> Integer
expModulo a b c = aux 1 a b
  where
    aux x y z | z == 0    = x
              | even z    = aux (rem x c) (rem (y^2) c) (z `div` 2)
              | otherwise = aux (rem (x*y) c) y (z-1)

-- 2ª definición
-- =============

expModulo2 :: Integer -> Integer -> Integer -> Integer
expModulo2 a b c = x1
  where
    (x1,y1,z1) = until p f (1,a,b)
    p (x,y,z) = z == 0
    f (x,y,z) | even z    = (rem x c, rem (y^2) c, z `div` 2)
              | otherwise = (rem (x*y) c, y, z-1)
```

```haskell
-- 3ª definición
-- =============

expModulo3 :: Integer -> Integer -> Integer -> Integer
expModulo3 a b c = x1
  where
    (x1,y1,z1) = head (dropWhile p (iterate f (1,a,b)))
    p (x,y,z) = z /= 0
    f (x,y,z) | even z    = (rem x c, rem (y^2) c, z `div` 2)
              | otherwise = (rem (x*y) c, y, z-1)

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

--    λ> let a = 10000^10000 in expModulo a a a
--    0
--    (1.12 secs, 1,580,023,160 bytes)
--    λ> let a = 10000^10000 in expModulo2 a a a
--    0
--    (1.28 secs, 1,604,449,976 bytes)
--    λ> let a = 10000^10000 in expModulo3 a a a
--    0
--    (1.18 secs, 1,614,503,344 bytes)
```

3.5. Examen 5 (03 de mayo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 5º examen de evaluación continua (3 de mayo de 2019)
-- ---------------------------------------------------------------------

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

import Data.Numbers.Primes (primeFactors)
import Data.List
import Data.Array
import I1M.Cola
import I1M.Grafo
import qualified Data.Matrix as M
```

```haskell
import Data.Maybe
import System.Timeout

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una partición buena de una lista de enteros positivos xs
-- es un par de listas (ys,zs) tal que son ys y zs disjuntas, la
-- unión de ys y zs es xs, y el máximo común divisor de los elementos de
-- ys coincide con el máximo común divisor de los elementos de zs.

-- Definir la función
--    particionesBuenas :: [Int] -> [([Int],[Int])]
-- tal que (particionesBuenas xs) es la lista de las particiones buenas
-- de xs. Por ejemplo,
--    particionesBuenas [1..10]               == [([1],[2,3,4,5,6,7,8,9,10])]
--    particionesBuenas [3,5..20]             == []
--    particionesBuenas [12,34,10,1020,2040] == [([12,34,10],[1020,2040])]
-- ---------------------------------------------------------------------

particionesBuenas :: [Int] -> [([Int],[Int])]
particionesBuenas xs = filter buena (particiones xs)
  where buena (as,bs) = foldr1 lcm as == foldr1 gcd bs

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Sea S un conjunto finito de números naturales y m un
-- número natural. El problema consiste en determinar si existe un
-- subconjunto de S cuya suma es m. Por ejemplo, si S = [3,34,4,12,5,2]
-- y m = 9, existe un subconjunto de S, [4,5], cuya suma es 9. En
-- cambio, no hay ningún subconjunto de S que sume 13.

-- Definir una función
--    existeSubSuma :: [Int] -> Int -> Bool
```

```haskell
-- tal que (existeSubSuma xs m) compruebe se existe algún subconjunto de
-- xs que sume m. Por ejemplo,
--    existeSubSuma [3,34,4,12,5,2] 9                  == True
--    existeSubSuma [3,34,4,12,5,2] 13                 == False
--    existeSubSuma ([3,34,4,12,5,2]++[20..400]) 13  == False
--    existeSubSuma ([3,34,4,12,5,2]++[20..400]) 654 == True
-- ---------------------------------------------------------------------

-- 1ª definición (Calculando todos los subconjuntos)
-- =================================================

existeSubSuma1 :: [Int] -> Int -> Bool
existeSubSuma1 xs n =
  any (\ys -> sum ys == n) (subsequences xs)

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

existeSubSuma2 :: [Int] -> Int -> Bool
existeSubSuma2 _  0 = True
existeSubSuma2 [] _ = False
existeSubSuma2 (x:xs) n
  | n < x     = existeSubSuma2 xs n
  | otherwise = existeSubSuma2 xs (n-x) || existeSubSuma2 xs n

-- 3ª definición (por programación dinámica)
-- =========================================

existeSubSuma3 :: [Int] -> Int -> Bool
existeSubSuma3 xs n =
  matrizExisteSubSuma3 xs n ! (length xs,n)

-- (matrizExisteSubSuma3 xs m) es la matriz q tal que q(i,j) se verifica
-- si existe algún subconjunto de (take i xs) que sume j. Por ejemplo,
--    λ> elems (matrizExisteSubSuma3 [1,3,5] 9)
--    [True,False,False,False,False,False,False,False,False,False,
--     True,True, False,False,False,False,False,False,False,False,
--     True,True, False,True, True, False,False,False,False,False,
--     True,True, False,True, True, True, True, False,True, True]
-- Con las cabeceras,
```

```haskell
--           0     1     2     3     4     5     6     7     8     9
--    []     [True,False,False,False,False,False,False,False,False,False,
--    [1]     True,True, False,False,False,False,False,False,False,False,
--    [1,3]   True,True, False,True, True, False,False,False,False,False,
--    [1,3,5] True,True, False,True, True, True, True, False,True, True]
matrizExisteSubSuma3 :: [Int] -> Int -> Array (Int,Int) Bool
matrizExisteSubSuma3 xs n = q
  where m = length xs
        v = listArray (1,m) xs
        q = array ((0,0),(m,n)) [((i,j), f i j) | i <- [0..m],
                                                  j <- [0..n]]
        f _ 0 = True
        f 0 _ = False
        f i j | j < v ! i = q ! (i-1,j)
              | otherwise = q ! (i-1,j-v!i) || q ! (i-1,j)

-- 4ª definición (ordenando y por recursión)
-- =========================================

existeSubSuma4 :: [Int] -> Int -> Bool
existeSubSuma4 xs = aux (sort xs)
  where aux _  0 = True
        aux [] _ = False
        aux (y:ys) n
          | n < y     = False
          | otherwise = aux ys (n-y) || aux ys n

-- 5ª definición (ordenando y dinámica)
-- ====================================

existeSubSuma5 :: [Int] -> Int -> Bool
existeSubSuma5 xs n =
  matrizExisteSubSuma5 (reverse (sort xs)) n ! (length xs,n)

matrizExisteSubSuma5 :: [Int] -> Int -> Array (Int,Int) Bool
matrizExisteSubSuma5 xs n = q
  where m = length xs
        v = listArray (1,m) xs
        q = array ((0,0),(m,n)) [((i,j), f i j) | i <- [0..m],
                                                  j <- [0..n]]
```

```haskell
        f _ 0 = True
        f 0 _ = False
        f i j | v ! i <= j = q ! (i-1,j-v!i) || q ! (i-1,j)
              | otherwise  = False

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

--    λ> let xs = [1..22] in existeSubSuma1 xs (sum xs)
--    True
--    (7.76 secs, 3,892,403,928 bytes)
--    λ> let xs = [1..22] in existeSubSuma2 xs (sum xs)
--    True
--    (0.02 secs, 95,968 bytes)
--    λ> let xs = [1..22] in existeSubSuma3 xs (sum xs)
--    True
--    (0.03 secs, 6,055,200 bytes)
--    λ> let xs = [1..22] in existeSubSuma4 xs (sum xs)
--    True
--    (0.01 secs, 98,880 bytes)
--    λ> let xs = [1..22] in existeSubSuma5 xs (sum xs)
--    True
--    (0.02 secs, 2,827,560 bytes)

--    λ> let xs = [1..200] in existeSubSuma2 xs (sum xs)
--    True
--    (0.01 secs, 182,280 bytes)
--    λ> let xs = [1..200] in existeSubSuma3 xs (sum xs)
--    True
--    (8.89 secs, 1,875,071,968 bytes)
--    λ> let xs = [1..200] in existeSubSuma4 xs (sum xs)
--    True
--    (0.02 secs, 217,128 bytes)
--    λ> let xs = [1..200] in existeSubSuma5 xs (sum xs)
--    True
--    (8.66 secs, 1,875,087,976 bytes)
--
--    λ> and [existeSubSuma2 [1..20] n | n <- [1..sum [1..20]]]
--    True
--    (2.82 secs, 323,372,512 bytes)
```

```
--    λ> and [existeSubSuma3 [1..20] n | n <- [1..sum [1..20]]]
--    True
--    (0.65 secs, 221,806,376 bytes)
--    λ> and [existeSubSuma4 [1..20] n | n <- [1..sum [1..20]]]
--    True
--    (4.12 secs, 535,153,152 bytes)
--    λ> and [existeSubSuma5 [1..20] n | n <- [1..sum [1..20]]]
--    True
--    (0.73 secs, 238,579,696 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Ulises, en sus ratos libres, juega a un pasatiempo que
-- consiste en, dada una serie de números naturales positivos en una
-- cola, sacar un elemento y, si es distinto de 1, volver a meter el
-- mayor de sus divisores propios. Si el número que saca es el 1,
-- entonces lo deja fuera y no mete ningún otro. El pasatiempo continúa
-- hasta que la cola queda vacía.
--
-- Por ejemplo, a partir de una cola con los números 10, 20 y 30, el
-- pasatiempo se desarrollaría como sigue:
--   C [30,20,10]
--   C [20,10,15]
--   C [10,15,10]
--   C [15,10,5]
--   C [10,5,5]
--   C [5,5,5]
--   C [5,5,1]
--   C [5,1,1]
--   C [1,1,1]
--   C [1,1]
--   C [1]
--   C []
--
-- Definir la función
--    numeroPasos :: Cola Int -> Int
-- tal que (numeroPasos c) es el número de veces que Ulises saca algún
-- número de la cola c al utilizarla en su pasatiempo. Por ejemplo,
--    numeroPasos (foldr inserta vacia [30])        ==  4
--    numeroPasos (foldr inserta vacia [20])        ==  4
--    numeroPasos (foldr inserta vacia [10])        ==  3
```

```haskell
--    numeroPasos (foldr inserta vacia [10,20,30])  ==  11
-- ---------------------------------------------------------------------

numeroPasos :: Cola Int -> Int
numeroPasos c
  | esVacia c = 0
  | pc == 1   = 1 + numeroPasos rc
  | otherwise = 1 + numeroPasos (inserta (mayorDivisorPropio pc) rc)
  where pc = primero c
        rc = resto c

-- (mayorDivisorPropio n) es el mayor divisor propio de n. Por ejemplo,
--    mayorDivisorPropio 30  ==  15

-- 1ª definición de mayorDivisorPropio
mayorDivisorPropio1 :: Int -> Int
mayorDivisorPropio1 n =
  head [x | x <- [n-1,n-2..], n `mod` x == 0]

-- 2ª definición de mayorDivisorPropio
mayorDivisorPropio :: Int -> Int
mayorDivisorPropio n =
  n `div` head (primeFactors n)

-- Comparación de eficiencia:
--    ghci> sum (map mayorDivisorPropio1 [2..3000])
--    1485659
--    (3.91 secs, 618,271,360 bytes)
--    ghci> sum (map mayorDivisorPropio [2..3000])
--    1485659
--    (0.04 secs, 22,726,600 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. El complementario del grafo G es un grafo G' del mismo
-- tipo que G (dirigido o no dirigido), con el mismo conjunto de nodos y
-- tal que dos nodos de G' son adyacentes si y sólo si no son adyacentes
-- en G. Los pesos de todas las aristas del complementario es igual a 0.
--
-- Definir la función
--    complementario :: Grafo Int Int -> Grafo Int Int
```

```
-- tal que (complementario g) es el complementario de g. Por ejemplo,
--    λ> complementario (creaGrafo D (1,3) [(1,3,0),(3,2,0),(2,2,0),(2,1,0)])
--    G D (array (1,3) [(1,[(1,0),(2,0)]),(2,[(3,0)]),(3,[(1,0),(3,0)])])
--    λ> complementario (creaGrafo D (1,3) [(3,2,0),(2,2,0),(2,1,0)])
--    G D (array (1,3) [(1,[(1,0),(2,0),(3,0)]),(2,[(3,0)]),(3,[(1,0),(3,0)])])
-- ---------------------------------------------------------------------

complementario :: Grafo Int Int -> Grafo Int Int
complementario g =
  creaGrafo d (1,n) [(x,y,0) | x <- xs, y <- xs, not (aristaEn g (x,y))]
  where d  = if dirigido g then D else ND
        xs = nodos g
        n  = length xs
```

3.6. Examen 6 (10 de junio de 2019)

```
-- Informática (1º del Grado en Matemáticas, Grupos 2, 3 y 5)
-- 6º examen de evaluación continua (10 de junio de 2019)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Char
import Data.Array
import Data.Numbers.Primes
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número de n dígitos es un número de Armstrong si es
-- igual a la suma de las n-ésimas potencias de sus dígitos. Por
-- ejemplo, 371, 8208 y 4210818 son números de Armstrong ya que
--        371 = 3^3 + 7^3 + 1^3 y
--       8208 = 8^4 + 2^4 + 0^4 + 8^4
--    4210818 = 4^7 + 2^7 + 1^7 + 0^7 + 8^7 + 1^7 + 8^7
--
-- Definir las funciones
--    esArmstrong :: Integer -> Bool
```

```haskell
--    armstrong :: [Integer]
-- tales que
-- + (esArmstrong x) se verifica si x es un número de Armstrong. Por
--   ejemplo,
--      esArmstrong 371                                  ==  True
--      esArmstrong 8208                                 ==  True
--      esArmstrong 4210818                              ==  True
--      esArmstrong 2015                                 ==  False
--      esArmstrong 115132219018763992565095597973971522401  ==  True
--      esArmstrong 115132219018763992565095597973971522402  ==  False
-- + armstrong es la lista cuyos elementos son los números de
--   Armstrong. Por ejemplo,
--      λ> take 18 armstrong
--      [1,2,3,4,5,6,7,8,9,153,370,371,407,1634,8208,9474,54748,92727]
--
-- Comprobar con QuickCheck que los números mayores que
-- 115132219018763992565095597973971522401 no son números de Armstrong.
-- ---------------------------------------------------------------------

esArmstrong :: Integer -> Bool
esArmstrong x =
  x == sum [d^n | d <- digitos x]
  where n = length (show x)

-- (digitos x) es la lista de los dígitos de x. Por ejemplo,
--    digitos 325  ==  [3,2,5]
digitos :: Integer -> [Integer]
digitos x = [read [d] | d <- show x]

armstrong :: [Integer]
armstrong = [n | n <- [1..], esArmstrong n]

-- La propiedad es
prop_Armstrong :: Integer -> Bool
prop_Armstrong n =
  not (esArmstrong (115132219018763992565095597973971522401 + abs n + 1))

-- La comprobación es
--    λ> quickCheck prop_Armstrong
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Un número n es k-belga si la sucesión cuyo primer
-- elemento es k y cuyos elementos se obtienen sumando reiteradamente
-- los dígitos de n contiene a n.
--
-- El 18 es 0-belga, porque a partir del 0 vamos a ir sumando
-- sucesivamente 1, 8, 1, 8, ... hasta llegar o sobrepasar el 18: 0, 1,
-- 9, 10, 18, ... Como se alcanza el 18, resulta que el 18 es 0-belga.
--
-- El 19 no es 1-belga, porque a partir del 1 vamos a ir sumando
-- sucesivamente 1, 9, 1, 9, ... hasta llegar o sobrepasar el 18: 1, 2,
-- 11, 12, 21, ... Como no se alcanza el 19, resulta que el 19 no es
-- 1-belga.
--
-- Definir la función
--    esBelga :: Integer -> Integer -> Bool
-- tal que (esBelga k n)  se verifica si n es k-belga. Por ejemplo,
--    esBelga 0 18                               ==  True
--    esBelga 1 19                               ==  False
--    esBelga 0 2016                             ==  True
--    [x | x <- [0..30], esBelga 7 x]            ==  [7,10,11,21,27,29]
--    [x | x <- [0..30], esBelga 10 x]           ==  [10,11,20,21,22,24,26]
--    length [n | n <- [1..9000], esBelga 0 n]   ==  2857
--
-- Comprobar con QuickCheck que para todo número entero positivo n, si
-- k es el resto de n entre la suma de los dígitos de n, entonces n es
-- k-belga.
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

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

-- 2ª solución
-- ===========
```

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

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

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

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

-- La propiedad es
prop_equivalencia :: Integer -> Integer -> Property
prop_equivalencia k n =
  n > 0 ==> esBelga1 k n == esBelga2 k n

-- La comprobación es
--    λ> quickCheck prop_equivalencia
--    +++ OK, passed 100 tests.

-- Verificación de la propiedad
-- ============================

-- La propiedad es
prop_Belga :: Integer -> Property
prop_Belga n =
  n > 0 ==> esBelga2 k n
  where k = n `mod` sum (digitos n)

-- La comprobación es
--    λ> quickCheck prop_Belga
```

```haskell
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3. El problema del número de emparejamiento de amigos
-- consiste en calcular el número de formas de emparejar n amigos
-- teniendo en cuenta que cada uno puede permanecer soltero o puede ser
-- emparejado con algún otro amigo y que cada amigo puede ser emparejado
-- sólo una vez. Por ejemplo, los 4 posibles emparejamientos de 3 amigos
-- son
--    {1}, {2}, {3}
--    {1}, {2, 3}
--    {1, 2}, {3}
--    {1, 3}, {2}
--
-- Definir, usando programación dinámica, la función
--    nEmparejamientos :: Integer -> Integer
-- tal que (nEmparejamientos n) es el número de formas de emparejar a
-- los n amigos. Por ejemplo,
--    nEmparejamientos 2   ==  2
--    nEmparejamientos 3   ==  4
--    nEmparejamientos 4   ==  10
--    nEmparejamientos 10  ==  9496
--    nEmparejamientos 30  ==  606917269909048576
--    length (show (nEmparejamientos3 (10^4)))  ==  17872
-- ---------------------------------------------------------------------

nEmparejamientos :: Integer -> Integer
nEmparejamientos n = (vectorEmparejamientos n) ! n

-- (vectorEmparejamientos n) es el vector con índices de 0 a n tal que el valor
-- de la posición i es el número de formas de emparejar a i amigos. Por ejemplo,
--    λ> vectorEmparejamientos 7
--    array (0,7) [(0,1),(1,1),(2,2),(3,4),(4,10),(5,26),(6,76),(7,232)]
vectorEmparejamientos :: Integer -> Array Integer Integer
vectorEmparejamientos n = v where
  v = array (0,n) [(i,f i) | i <- [0..n]]
  f 0 = 1
  f 1 = 1
  f 2 = 2
  f k = v!(k-1) + (k-1)*v!(k-2)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Se dice que a es un divisor propio maximal de un número
-- b si a es un divisor de b distinto de b y no existe ningún número c
-- tal que a < c < b, a es un divisor de c y c es un divisor de b. Por
-- ejemplo, 15 es un divisor propio maximal de 30, pero 5 no lo es.
--
-- El árbol de los divisores de un número x es el árbol que tiene como
-- raíz el número x y cada nodo tiene como hijos sus divisores propios
-- maximales. Por ejemplo, el árbol de divisores de 30 es
--
--              30
--              /|\
--             / | \
--            /  |  \
--           /   |   \
--          /    |    \
--         /     |     \
--        6     10     15
--       / \    / \    / \
--      2   3  2   5  3   5
--      |   |  |   |  |   |
--      1   1  1   1  1   1
--
-- Usando el tipo de dato
--    data Arbol = N Integer [Arbol]
--      deriving (Eq, Show)
-- el árbol anterior se representa por
--    N 30
--      [N 6
--         [N 2 [N 1 []],
--          N 3 [N 1 []]],
--       N 10
--         [N 2 [N 1 []],
--          N 5 [N 1 []]],
--       N 15
--         [N 3 [N 1 []],
--          N 5 [N 1 []]]]
--
-- Definir las funciones
```

```haskell
--    arbolDivisores             :: Integer -> Arbol
--    nOcurrenciasArbolDivisores :: Integer -> Integer -> Integer
-- tales que
-- + (arbolDivisores x) es el árbol de los divisores del número x. Por
--   ejemplo,
--      λ> arbolDivisores 30
--      N 30 [N 6  [N 2 [N 1 []],N 3 [N 1 []]],
--            N 10 [N 2 [N 1 []],N 5 [N 1 []]],
--            N 15 [N 3 [N 1 []],N 5 [N 1 []]]]
-- + (nOcurrenciasArbolDivisores x y) es el número de veces que aparece
--   el número x en el árbol de los divisores del número y. Por ejemplo,
--      nOcurrenciasArbolDivisores  3 30  ==  2
--      nOcurrenciasArbolDivisores  6 30  ==  1
--      nOcurrenciasArbolDivisores 30 30  ==  1
--      nOcurrenciasArbolDivisores  1 30  ==  6
--      nOcurrenciasArbolDivisores  9 30  ==  0
--      nOcurrenciasArbolDivisores  2 (product [1..10])  ==  360360
--      nOcurrenciasArbolDivisores  3 (product [1..10])  ==  180180
--      nOcurrenciasArbolDivisores  5 (product [1..10])  ==  90090
--      nOcurrenciasArbolDivisores  7 (product [1..10])  ==  45045
--      nOcurrenciasArbolDivisores  6 (product [1..10])  ==  102960
--      nOcurrenciasArbolDivisores 10 (product [1..10])  ==  51480
--      nOcurrenciasArbolDivisores 14 (product [1..10])  ==  25740
-- ---------------------------------------------------------------------

data Arbol = N Integer [Arbol]
  deriving (Eq, Show)

-- Definición de arbolDivisores
-- ============================

arbolDivisores :: Integer -> Arbol
arbolDivisores x =
  N x (map arbolDivisores (divisoresPropiosMaximales x))

-- (divisoresPropiosMaximales x) es la lista de los divisores propios
-- maximales de x. Por ejemplo,
--    divisoresPropiosMaximales 30   ==  [6,10,15]
--    divisoresPropiosMaximales 420  ==  [60,84,140,210]
--    divisoresPropiosMaximales 7    ==  [1]
```

```haskell
divisoresPropiosMaximales :: Integer -> [Integer]
divisoresPropiosMaximales x =
  reverse [x `div` y | y <- nub (primeFactors x)]

-- Definición de nOcurrenciasArbolDivisores
-- ========================================

nOcurrenciasArbolDivisores :: Integer -> Integer -> Integer
nOcurrenciasArbolDivisores x y =
  nOcurrencias x (arbolDivisores y)

-- (nOcurrencias x a) es el número de veces que aprece x en el árbol
-- a. Por ejemplo,
--    nOcurrencias 3 (arbolDivisores 30)  ==  2
nOcurrencias :: Integer -> Arbol -> Integer
nOcurrencias x (N y [])
  | x == y    = 1
  | otherwise = 0
nOcurrencias x (N y zs)
  | x == y    = 1 + sum [nOcurrencias x z | z <- zs]
  | otherwise = sum [nOcurrencias x z | z <- zs]
```

3.7. Examen 7 (02 de julio de 2019)

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

3.8. Examen 8 (11 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 59).

4

Exámenes del grupo 4

Antonia M. Chávez

4.1. Examen 1 (08 de noviembre de 2018)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (8 de noviembre de 2017)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    porTramos :: (Num a, Ord a) => [a] -> Bool
-- tal que (porTramos xs) se verifica si la diferencia entre numeros
-- consecutivos de xs es siempre, en valor absoluto, mayor que 2. Por
-- ejemplo,
--    porTramos [1,5,8,23,5] == True
--    porTramos [3,6,8,1]    == False
--    porTramos [-5,-2,4 ]   == True
--    porTramos [5,2]        == True
-- ---------------------------------------------------------------------

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

-- 2ª definición
porTramos2 :: (Num a, Ord a) => [a] -> Bool
porTramos2 []        = True
porTramos2 [_]       = True
```

```haskell
porTramos2 (x:y:xs) = abs (x-y) > 2 && porTramos2 (y:xs)

-- 3ª definición
porTramos3 :: (Num a, Ord a) => [a] -> Bool
porTramos3 (x:y:xs) = abs (x-y) > 3 && porTramos3 (y:xs)
porTramos3 _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la funcion
--    sonMenores :: Ord a => [a] -> [a] -> Int
-- tal que (sonMenores xs ys) es el número de elementos de xs son
-- menores que los que ocupan la misma posicion en ys hasta el primero
-- que no lo sea. Por ejemplo,
--    sonMenores "prueba" "suspenso"       == 2
--    sonMenores [1,2,3,4,5] [6,5,4,3,8,9] == 3
-- ---------------------------------------------------------------------

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

-- 2ª definición
sonMenores2 :: Ord a => [a] -> [a] -> Int
sonMenores2 xs ys =
  length (takeWhile (\(x,y) -> x < y) (zip xs ys))

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Se dice que un número entero positivo es raro si al
-- sumar cada una de sus cifras elevadas al numero de cifras que lo
-- forman, obtenemos el propio numero. Por ejemplo, el 153 (que tiene 3
-- cifras) es raro ya que 153 es 1^3 + 5^3 + 3^3.
--
-- Definir la función
--    esRaro :: Integer -> Bool
-- tal que (esRaro x) se verifica si x es raro. Por ejemplo,
--    esRaro 3     == True
--    esRaro 153   == True
```

```haskell
--    esRaro 12   == False
--    esRaro 8208 == True
-- ---------------------------------------------------------------------

esRaro :: Integer -> Bool
esRaro x = sum [y^n | y <- ds] == x
  where ds = digitos x
        n  = length ds

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la funcion
--    primerosRaros :: Int -> [Integer]
-- tal que (primerosRaros n) que es la lista de los n primeros números
-- raros. Por ejemplo,
--    primerosRaros 15 == [1,2,3,4,5,6,7,8,9,153,370,371,407,1634,8208]
-- ---------------------------------------------------------------------

primerosRaros :: Int -> [Integer]
primerosRaros n = take n [x | x <- [1 ..], esRaro x]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la funcion
--    quitaCentro :: [a] -> [a]
-- tal que (quitaCentro xs) es la lista obtenida eliminando en xs su
-- elemento central si xs es de longitud impar y sus dos elementos
-- centrales si es de longitud par. Por ejemplo,
--    quitaCentro [1,2,3,4]             == [1,4]
--    quitaCentro [1,2,3]               == [1,3]
--    quitaCentro "examen1"             == "exaen1"
--    quitaCentro [[1,2],[3,2,5],[5,4]] == [[1,2],[5,4]]
--    quitaCentro [6]                   == []
-- ---------------------------------------------------------------------

-- 1ª solución
quitaCentro :: [a] -> [a]
quitaCentro [] = []
quitaCentro xs
```

```haskell
  | even n      = init (take m xs) ++ drop (m+1) xs
  | otherwise = take m xs ++ drop (m+1) xs
  where n = length xs
        m = n `div` 2

-- 2ª solución
quitaCentro2 :: [a] -> [a]
quitaCentro2 xs =
  [x | (x,y) <- zip xs [1..], y `notElem` numeroscentro xs]

numeroscentro :: [a] -> [Int]
numeroscentro xs | even n      = [m,m+1]
                 | otherwise = [m+1]
  where n = length xs
        m = n `div` 2

-- 2ª solución
quitaCentro3 :: [a] -> [a]
quitaCentro3 [] = []
quitaCentro3 xs
  | even n      = init as ++ bs
  | otherwise = as ++ bs
  where n          = length xs
        m          = n `div` 2
        (as,_:bs) = splitAt m xs
```

4.2. Examen 2 (11 de diciembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (11 de noviembre de 2018)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función:
```

```haskell
--   deshacer :: [(a,b)] -> ([a],[b])
-- tal que (deshacer ps) es el par de listas que resultan de separar los
-- pares de ps. Por ejemplo,
--   ghci> deshacer [(3,'l'),(2,'u'),(5,'i'),(9,'s')]
--   ([3,2,5,9],"luis")
--
-- Comprobar con QuickCheck que deshacer es equivalente a la función
-- predefinida unzip.
-- ---------------------------------------------------------------------

-- 1ª definición
deshacer :: [(a,b)] -> ([a],[b])
deshacer ps = ([x | (x,_) <- ps], [y | (_,y) <- ps])

-- 2ª definición:
deshacer2 :: [(a,b)] -> ([a],[b])
deshacer2 ps = (map fst ps, map snd ps)

-- 3ª definición:
deshacer3 :: [(a,b)] -> ([a],[b])
deshacer3 []         = ([],[])
deshacer3 ((x,y):ps) = (x:xs,y:ys)
  where (xs,ys) = deshacer3 ps

-- 4ª definición:
deshacer4 :: [(a,b)] -> ([a],[b])
deshacer4 = foldr f ([],[])
  where f (x,y) (xs,ys) = (x:xs, y:ys)

-- 5ª definición:
deshacer5 :: [(a,b)] -> ([a],[b])
deshacer5 = aux ([],[])
  where aux r []               = r
        aux (as,bs) ((x,y):ps) = aux (as++[x],bs++[y]) ps

-- 6ª definición:
deshacer6 :: [(a,b)] -> ([a],[b])
deshacer6 ps = (reverse us, reverse vs)
  where (us,vs) = foldl f ([],[]) ps
        f (xs,ys) (x,y) = (x:xs,y:ys)
```

```haskell
-- La propiedad es
prop_deshacer :: [(Int,Int)] -> Bool
prop_deshacer ps =
  all (== unzip ps) [f ps | f <- [ deshacer
                                 , deshacer2
                                 , deshacer3
                                 , deshacer4
                                 , deshacer5
                                 , deshacer6
                                 ]]

-- La comprobación es
--    λ> quickCheck prop_deshacer
--    +++ OK, passed 100 tests.

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

-- 1ª definición:
nPlica :: Int -> [a] -> [a]
nPlica k xs = concat [replicate k x | x <- xs]

-- 2ª definición:
nPlica2 :: Int -> [a] -> [a]
nPlica2 k xs = concat (map (replicate k) xs)

-- 3ª definición:
nPlica3 :: Int -> [a] -> [a]
nPlica3 k = concatMap (replicate k)

-- 4ª definición:
nPlica4 :: Int -> [a] -> [a]
nPlica4 k []     = []
nPlica4 k (x:xs) = replicate k x ++ nPlica4 k xs
```

```haskell
-- 5ª definición:
nPlica5 :: Int -> [a] -> [a]
nPlica5 k = foldr ((++) . replicate k) []

-- 6ª definición:
nPlica6 :: Int -> [a] -> [a]
nPlica6 k = foldr f []
  where f prim recur = replicate k prim ++ recur

-- 7ª definición:
nPlica7 :: Int -> [a] -> [a]
nPlica7 k = aux []
  where aux ys []     = ys
        aux ys (x:xs) = aux (ys ++ replicate k x) xs

-- 8ª definición:
nPlica8 :: Int -> [a] -> [a]
nPlica8 k xs = foldl f [] xs
  where f ys prim = ys ++ replicate k prim

-- Equivalencia:
prop_nPlica :: Int -> [Int] -> Bool
prop_nPlica n xs =
  all (== nPlica n xs) [f n xs | f <- [ nPlica2
                                      , nPlica3
                                      , nPlica4
                                      , nPlica5
                                      , nPlica6
                                      , nPlica7
                                      , nPlica8
                                      ]]

-- La comprobación es
--    λ> quickCheck prop_nPlica
--    +++ OK, passed 100 tests.
```

4.3. Examen 3 (22 de enero de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 1, 2 y 3)
-- 3º examen de evaluación continua (22 de enero de 2019)
-- ---------------------------------------------------------------------

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

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    divisoresConFinal :: Integer -> Integer -> [Integer]
-- tal que (divisoresConFinal n m) es la lista de los divisores de n
-- cuyos dígitos finales coincide con m. por ejemplo,
--    divisoresConFinal 84 4    ==  [4,14,84]
--    divisoresConFinal 720 20  ==  [20,120,720]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

divisoresConFinal :: Integer -> Integer -> [Integer]
divisoresConFinal n m =
  [x | x <- [1..n], n `rem` x == 0, final x m]

final :: Integer -> Integer -> Bool
final x y = take n xs == ys
    where xs = reverse (show x)
          ys = reverse (show y)
          n  = length ys

-- 2ª solución
-- ===========

divisoresConFinal2 :: Integer -> Integer -> [Integer]
divisoresConFinal2 n m =
    [x | x <- [1..n], n `rem` x == 0, show m `isSuffixOf` show x]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    alternativa :: (a -> b) -> (a -> b) -> [a] -> [b]
-- tal que (alternativa f g xs) es la lista obtenida aplicando
-- alternativamente las funciones f y g a los elementos de xs. por
-- ejemplo,
--    alternativa (+1)  (+10) [1,2,3,4]    ==  [2,12,4,14]
--    alternativa (+10) (*10) [1,2,3,4,5]  ==  [11,20,13,40,15]
-- ---------------------------------------------------------------------

-- 1ª solución
alternativa :: (a -> b) -> (a -> b) -> [a] -> [b]
alternativa f g []     = []
alternativa f g (x:xs) = f x : alternativa g f xs

-- 2ª solución
alternativa2 :: (a -> b) -> (a -> b) -> [a] -> [b]
alternativa2 f g xs =
  [h x | (h,x) <- zip (cycle [f,g]) xs]

-- ---------------------------------------------------------------------
-- Ejercicio 3. La sucesión de los primeros números de Fibonacci es
--    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
-- el fibonacci más cercano a un número x es el menor elemento y de la
-- sucesión de Fibonacci tal que el valor absoluto de la diferencia
-- entre x e y es la menor posible. por ejemplo,
-- + el fibonacci más cercano a 16 es 13 porque |16-13| < |16-21|
-- + el fibonacci más cercano a 17 es 13 porque |17-13| = |17-21| y 17 < 21
-- + el fibonacci más cercano a 18 es 21 porque |18-13| > |18-21|
-- + el fibonacci más cercano a 21 es 21 porque 21 es un número de fibonacci.
--
-- Definir la función
--    fibonacciMasCercano :: Integer -> Integer
-- tal que (fibonacciMasCercano n) es el número de fibonacci más cercano
-- a n. por ejemplo,
--    fibonacciMasCercano 16    ==  13
--    fibonacciMasCercano 17    ==  13
--    fibonacciMasCercano 18    ==  21
--    fibonacciMasCercano 21    ==  21
```

```
--    fibonacciMasCercano 2019  ==  1597
-- ---------------------------------------------------------------------

fibonacciMasCercano :: Integer -> Integer
fibonacciMasCercano n
  | b == n                  = n
  | abs (n-a) <= abs (n-b) = a
  | otherwise              = b
  where (xs,b:ys) = span (<n) fibs
        a         = last xs

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

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

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

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

-- 1ª solución
ramas :: Arbol b -> [[b]]
ramas (N x []) = [[x]]
ramas (N x as) = [x : xs | a <- as, xs <- ramas a]

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

-- 3ª solución
ramas3 :: Arbol b -> [[b]]
ramas3 (N x []) = [[x]]
ramas3 (N x as) = concatMap (map (x:)) (map ramas3 as)
```

4.4. Examen 4 (26 de marzo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 4º examen de evaluación continua (26 de marzo de 2019)
-- ---------------------------------------------------------------------

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

import Data.Array
import Data.Maybe
import I1M.Cola

-- --------------------------------------------------------------------
-- Ejercicio 1.1. Se considera el tipo de dato de las matrices:
--    type Matriz a = Array (Int,Int) a
--
-- Dada una posicion (i,j) en una matriz, los vecinos a distancia d de la
```

```haskell
-- posicion son aquellos valores que están a d pasos de la posicion (i,j)
-- contando los pasos en vertical y/o horizontal.
--
-- Definir la función
--    vecinosD :: (Int,Int) -> Matriz Integer -> Int -> [Integer]
-- tal que (vecinosD p a d) es la lista de los vecinos en la matriz a
-- de la posición p que están a distancia d de p. Por ejemplo, sea ejM la
-- definida por
--    ejM1 = listArray ((1,1),(5,5)) [1..25]
--    ejM2 = listArray ((1,1),(3,4)) ['a'..]
-- entonces,
--    vecinosD (1,1) ejM1 1  ==  [2,6]
--    vecinosD (1,1) ejM1 2  ==  [3,7,11]
--    vecinosD (1,1) ejM1 3  ==  [4,8,12,16]
--    vecinosD (5,5) ejM1 1  ==  [20,24]
--    vecinosD (5,4) ejM1 1  ==  [19,23,25]
--    vecinosD (5,4) ejM1 3  ==  [9,13,15,17,21]
--    vecinosD (2,2) ejM1 6  ==  [25]
--    vecinosD (2,2) ejM1 7  ==  []
--    vecinosD (2,3) ejM2 2  ==  "bdejl"
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

ejM1 :: Matriz Int
ejM1 = listArray ((1,1),(5,5)) [1..25]

ejM2 :: Matriz Char
ejM2 = listArray ((1,1),(3,4)) ['a'..]

vecinosD :: (Int,Int) -> Matriz a -> Int -> [a]
vecinosD (i,j) a d =
  [a!(k,l) | (k,l) <- indices a
           ,  abs(i-k) + abs(j-l) == d]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    circuloDeVecinos :: (Int,Int) -> Matriz Integer -> [[Integer]]
-- tal que (circuloDeVecinos p a) es la lista de listas de los vecinos
-- en a de p que están a distancia 1, 2, ... mientras tenga sentido.
```

```haskell
-- Por ejemplo,
--   λ> circuloDeVecinos (2,3) ejM1
--   [[3,7,9,13],[2,4,6,10,12,14,18],[1,5,11,15,17,19,23],[16,20,22,24],[21,25]]
--   λ> circuloDeVecinos (2,3) ejM2
--   ["cfhk","bdejl","ai"]
-- ---------------------------------------------------------------------

circuloDeVecinos :: (Int,Int) -> Matriz a -> [[a]]
circuloDeVecinos p a =
  takeWhile (not . null) [vecinosD p a d | d <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    minimasDistancias :: Eq a => Matriz a -> a -> Matriz Int
-- tal que, (minimasDistancias a x) es la matriz de las distancias de
-- cada elemento al elemento x más cercano, donde x es un elemento de la
-- matriz a. Por ejemplo,
--    λ> elems (minimasDistancias ejM1 7)
--    [2,1,2,3,4,
--     1,0,1,2,3,
--     2,1,2,3,4,
--     3,2,3,4,5,
--     4,3,4,5,6]
--    λ> elems (minimasDistancias ejM1 12)
--    [3,2,3,4,5,
--     2,1,2,3,4,
--     1,0,1,2,3,
--     2,1,2,3,4,
--     3,2,3,4,5]
--    λ> elems (minimasDistancias ejM2 'c')
--    [2,1,0,1,
--     3,2,1,2,
--     4,3,2,3]
-- ---------------------------------------------------------------------

minimasDistancias :: Eq a => Matriz a -> a -> Matriz Int
minimasDistancias a x =
  array (bounds a) [((i,j), f i j) | (i,j) <- indices a]
    where f i j | a!(i,j) == x = 0
                | otherwise    = minimaDistancia (i,j) a x
```

```haskell
minimaDistancia :: Eq a => (Int,Int) -> Matriz a -> a -> Int
minimaDistancia p a x =
  head [d | (d,vs) <- zip [1..] (circuloDeVecinos p a)
          , x `elem` vs]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Se considera el siguiente tipo de dato de los árboles
-- binarios:
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
--    existenciaYunicidad :: Arbol a -> (a -> Bool) -> Bool
-- tal que (existenciaYunicidad a p) se verifica si el árbol a tiene
-- exactamente  un elemento que cumple la propiedad p. Por ejemplo,
--   existenciaYunicidad (N 2 (N 4 (H 1)(H 2))(N 3 (H 1) (H 0))) (>3) == True
--   existenciaYunicidad (N 2 (N 4 (H 1)(H 2))(N 6 (H 1) (H 0))) (>3) == False
--   existenciaYunicidad (N 2 (N 4 (H 1)(H 2))(N 6 (H 1) (H 0))) (>7) == False
-- ---------------------------------------------------------------------

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

existenciaYunicidad :: Arbol a -> (a -> Bool) -> Bool
existenciaYunicidad a p = length (cumplidores a p) == 1

cumplidores :: Arbol a -> (a -> Bool) -> [a]
cumplidores (H x) p | p x       = [x]
                    | otherwise = []
cumplidores (N x i d) p | p x       = x : cumplidores i p ++ cumplidores d p
                        | otherwise = cumplidores i p ++ cumplidores d p

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    unico :: Arbol a -> (a -> Bool) -> Maybe a
-- tal que (unico a p) es el único elemento del árbol a que cumple la
-- propiedad p, si existe un único elemento que la cumple y Nothing, en
```

```haskell
-- caso contrario. Por ejemplo,
--    unico (N 2 (N 4 (H 1)(H 2))(N 3 (H 1) (H 0))) (>3)  == Just 4
--    unico (N 2 (N 4 (H 1)(H 2))(N 3 (H 1) (H 0))) (<1)  == Just 0
--    unico (N 2 (N 4 (H 1)(H 2))(N 3 (H 1) (H 0))) (<2)  == Nothing
--    unico (N 2 (N 4 (H 1)(H 2))(N 3 (H 1) (H 0))) (>4)  == Nothing
-- ---------------------------------------------------------------------

unico :: Arbol a -> (a -> Bool) -> Maybe a
unico a p | existenciaYunicidad a p = aux a p
          | otherwise               = Nothing
  where aux (H x) p | p x       = Just x
                    | otherwise = Nothing
        aux (N x i d) p | p x       = Just x
                        | isJust r  = r
                        | otherwise = aux d p
          where r = aux i p

-- --------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    intercambiaPrimero :: [a] -> [[a]]
-- tal que (intercambiaPrimero xs) es la lista de las listas obtenidas
-- intercambiando el primer elemento de xs por cada uno de los
-- demás. Por ejemplo,
-- intercambiaPrimero [1,2,3,4]  == [[2,1,3,4],[2,3,1,4],[2,3,4,1]]
-- intercambiaPrimero [3]        == [[3]]
-- ---------------------------------------------------------------------

intercambiaPrimero :: [a] -> [[a]]
intercambiaPrimero []     = []
intercambiaPrimero [x]    = [[x]]
intercambiaPrimero (x:xs) = [coloca x n xs | n <- [1.. length xs]]

coloca :: a -> Int -> [a] -> [a]
coloca x n xs = ys ++ x : zs
  where (ys,zs) = splitAt n xs

-- --------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    sustituyeEnCola :: Eq a => Cola a -> a -> a -> Cola a
-- tal que (sustituyeEnCola c x y) es la cola obtenida sustiyendo x por
```

```haskell
-- y en c. Por ejemplo,
--    λ> c = foldr inserta vacia [1,0,2,0,3,0,4,0,5,0]
--    λ> c
--    C [0,5,0,4,0,3,0,2,0,1]
--    λ> sustituyeEnCola c 0 8
--    C [8,5,8,4,8,3,8,2,8,1]
--    λ> sustituyeEnCola c 3 7
--    C [0,5,0,4,0,7,0,2,0,1]
--    λ> sustituyeEnCola c 6 7
--    C [0,5,0,4,0,3,0,2,0,1]
-- ---------------------------------------------------------------------

sustituyeEnCola :: Eq a => Cola a -> a -> a -> Cola a
sustituyeEnCola c x y = aux c vacia
  where aux c' r
          | esVacia c' = r
          | pc' == x   = aux rc' (inserta y r)
          | otherwise  = aux rc' (inserta pc' r)
          where pc' = primero c'
                rc' = resto c'
```

4.5. Examen 5 (16 de mayo de 2019)

```haskell
-- Informatica (1º del Grado en Matematicas) Grupo 1
-- Examen 5 de evaluacion alternativa (3 de mayo de 2019)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Librerias auxiliares
-- ---------------------------------------------------------------------

import Data.List
import Data.Array
import I1M.Cola
import I1M.PolOperaciones
import I1M.Grafo

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número natural x es un cuadrado perfecto si x es
-- el cuadrado de algún natural. Por ejemplo, 36 es un cuadrado perfecto
```

```haskell
-- y 12 no lo es.
--
-- Un número entero positivo sera ordenado si sus dígitos
-- aparecen en orden creciente o bien en orden decreciente. Por
-- ejemplo, 11468 y 974000 son ordenados y 16832 no lo es.
--
-- Definir la lista infinita
--    cuadradosOrdenados :: [Integer]
-- cuyos elementos son los cuadrados que son ordenados. Por ejemplo,
--    Main> take 19 cuadradosOrdenados
--    [1,4,9,16,25,36,49,64,81,100,144,169,225,256,289,400,441,841,900]
-- ---------------------------------------------------------------------

cuadradosOrdenados :: [Integer]
cuadradosOrdenados = filter ordenado [i^2 | i <- [1..]]

ordenado :: Integer -> Bool
ordenado x = cs == ds || cs == reverse ds
  where cs = digitos x
        ds = sort cs

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcula el mayor cuadrado ordenado de 7 dígitos.
-- ---------------------------------------------------------------------

-- El cálculo es
--   Main> last (takeWhile (<=10^7-1) cuadradosOrdenados)
--   9853321

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    posicionesPares :: Cola a -> Cola a
-- tal que (posicionesPares p) devuelve la cola obtenida tomando los
-- elementos que ocupan las posiciones pares en la cola p. Por ejemplo,
--    Main> c = (foldr inserta vacia [0..9])
--    Main> d = (foldr inserta vacia [0..10])
--    Main> c
```

```haskell
--    C [9,8,7,6,5,4,3,2,1,0]
--    Main> colaPosPares c
--    C [9,7,5,3,1]
--    Main> d
--    C [10,9,8,7,6,5,4,3,2,1,0]
--    Main> colaPosPares d
--    C [10,8,6,4,2,0]
--    Main> colaPosPares (foldr inserta  vacia "salamanca")
--    C "anmls"
-- ---------------------------------------------------------------------

colaPosPares :: Cola a -> Cola a
colaPosPares p = aux p vacia
  where aux c r
          | esVacia c  = r
          | esVacia rc = inserta pc r
          | otherwise  = aux (resto rc) (inserta pc r)
          where (pc,rc) = (primero c, resto c)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una matriz de enteros p se dira "de unos en línea" si
-- cumple que:
--    toda fila de p contiene, como maximo un 1;
--    o bien toda columna de p contiene, como maximo un 1.
--
--  Definir la función
--     unoEnlinea :: Matrix Int -> Bool
--  tal que (unoEnlinea p) se verifica si la matriz p es de unos en
--  linea según la definición anterior. Por ejemplo,
--     Main> unoEnlinea (fromLists [[0,0,1,0,0],[1,0,0,0,0],[0,0,0,0,1]])
--     True
--     Main> unoEnlinea (fromLists [[0,0,0,0],[0,1,0,0],[0,0,0,1],[0,1,1,0]])
--     False
--     Main> unoEnlinea (fromLists [[0,1],[2,0]])
--     True
-- ---------------------------------------------------------------------

unoEnlinea :: Array (Int,Int) Int -> Bool
unoEnlinea p =
  all tieneComoMaximoUnUno (filas ++ columnas)
```

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

tieneComoMaximoUnUno :: [Int] -> Bool
tieneComoMaximoUnUno xs = length (filter (==1) xs) <= 1

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    corte :: (Eq a,Num a) => Int -> Polinomio a -> Polinomio a
-- tal que (corte k p) es el polinomio formado por los términos de p de
-- grado mayor o igual que k. Por ejemplo,
--    Main> corte 3 (consPol 5 2 (consPol 3 (-7) (consPol 2 1 polCero)))
--    2*x^5 + -7*x^3
--    Main> corte 2 (consPol 5 2 (consPol 3 (-7) (consPol 2 1 polCero)))
--    2*x^5 + -7*x^3 + x^2
--    Main> corte 4 (consPol 5 2 (consPol 3 (-7) (consPol 2 1 polCero)))
--    2*x^5
-- ---------------------------------------------------------------------

corte :: (Eq a, Num a) => Int -> Polinomio a -> Polinomio a
corte k p
  | k < 0        = polCero
  | esPolCero p = polCero
  | k > n        = polCero
  | otherwise    = consPol n a (corte k (restoPol p))
  where n = grado p
        a = coefLider p

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Un polinomio de enteros se dirá impar si su término
-- independiente es impar y el resto de sus coeficientes (si los
-- hubiera) son pares.
--
-- Definir la función
--    imparPol :: Integral a => Polinomio a -> Bool
-- tal que (imparPol p) se verifica si p es un polinomio impar de
-- acuerdo con la definicion anterior. Por ejemplo,
--    Main> imparPol (consPol 5 2 (consPol 3 6 (consPol 0 3 polCero)))
--    True
```

```haskell
--    Main> imparPol (consPol 5 2 (consPol 3 6 (consPol 0 4 polCero)))
--    False
--    Main> imparPol (consPol 5 2 (consPol 3 1 (consPol 0 3 polCero)))
--    False
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

imparPol :: Integral a => Polinomio a -> Bool
imparPol p = odd (coeficiente 0 p) &&
             all even [coeficiente k p | k <- [1..grado p]]

coeficiente :: Integral a => Int -> Polinomio a -> a
coeficiente k p
  | esPolCero p = 0
  | k > n       = 0
  | k == n      = coefLider p
  | otherwise   = coeficiente k (restoPol p)
  where n = grado p

-- 2ª solución
-- ===========

imparPol2 :: Integral a => Polinomio a -> Bool
imparPol2 p =
  all even [coeficiente k (sumaPol p polUnidad) | k <- [0..grado p]]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Un conjunto V de nodos de un grafo no dirigido g forma
-- un club si dos elementos cualesquiera de V tienen una arista en g que
-- los conecta. Por ejemplo, en el grafo:
--       4 ---- 5
--       |      | \
--       |      |  1
--       |      | /
--       3 ---- 2
-- el conjunto de vértices {1,2,5} forma un club y el conjunto {2,3,4,5}
-- no.
--
```

```haskell
-- En Haskell se puede representar el grafo anterior por
--    g1 :: Grafo Int Int
--    g1 = creaGrafo ND
--                   (1,5)
--                   [(1,2,0),(1,5,0),(2,3,0),(3,4,0),(5,2,0),(4,5,0)]
--
-- Definir la función
--    esUnClub :: Grafo Int Int -> [Int] -> Bool
-- tal que (esUnClub g xs) se verifica si xs forma un club en g. Por
-- ejemplo,
--    esUnClub g1 [1,2,5]   == True
--    esUnClub g1 [2,3,4,5] == False
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 6. El siguiente triángulo se construye sabiendo
-- que se verifica la siguiente relación
--    A(n,m)=(n-m)A(n-1,m-1) + (m+1)A(n-1,m).
-- Sus primeros términos son
--    1
--    1 0
--    1 1   0
--    1 4   1     0
--    1 11  11    1     0
--    1 26  66    26    1      0
--    1 57  302   302   57     1     0
--    1 120 1191  2416  1191   120   1     0
--    1 247 4293  15619 15619  4293  247   1   0
--    1 502 14608 88234 156190 88234 14608 502 1 0
--
-- Definir las siguientes funciones:
```

```
--   num          :: Integer -> Integer -> Integer
--   filaTriangulo :: Integer -> [Integer]
--   triangulo     :: [[Integer]]
-- tales que
-- + (num n k) es el número A(n,k). Por ejemplo,
--      num 8 3  == 15619
--      num 20 6 == 21598596303099900
-- + (filaTriangulo n) es la n-sima fila del triángulo. Por ejemplo,
--      ghci> filaTriangulo 8
--      [1,247,4293,15619,15619,4293,247,1]
--      ghci> filaTriangulo 12
--      [1,4083,478271,10187685,66318474,162512286,162512286,66318474,
--       10187685,478271,4083,1]
-- + triangulo es la lista con las filas del triángulo
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

triangulo1 :: [[Integer]]
triangulo1 = iterate siguiente [1]

siguiente :: [Integer] -> [Integer]
siguiente xs = zipWith (+) us vs
  where n = genericLength xs
        us = [x*k | (x,k) <-zip (0:xs) [n+1,n..1]]
        vs = [x*k | (x,k) <-zip (xs++[0]) [1..n+1]]

-- Otra definición de siguiente es
siguiente' :: [Integer] -> [Integer]
siguiente' xs = zipWith (+) us vs
  where n = genericLength xs
        us = zipWith (*) (0:xs) [n+1,n..1]
        vs = zipWith (*) (xs++[0]) [1..n+1]

filaTriangulo1 :: Integer -> [Integer]
filaTriangulo1 n = triangulo1 `genericIndex` (n-1)

num1 :: Integer -> Integer -> Integer
num1 n k = (filaTriangulo1 n) `genericIndex` k
```

```haskell
-- 2ª solución

num :: Integer -> Integer -> Integer
num n k = (matrizE n k) ! (n,k)

matrizE :: Integer -> Integer -> Array (Integer,Integer) Integer
matrizE n m = q
  where q = array ((0,0),(n,m)) [((i,j), f i j) | i <-[0..n],j<-[0..m]]
        f i 0 = 1
        f i j
          | i == j    = 0
          | otherwise = (i-j) * q!(i-1,j-1) + (j+1)* q!(i-1,j)

filaTriangulo :: Integer -> [Integer]
filaTriangulo n = map (num n) [0..n]

triangulo :: [[Integer]]
triangulo = map filaTriangulo [0..]
```

4.6. Examen 6 (10 de junio de 2019)

El examen es común con el del grupo 1 (ver página 45).

4.7. Examen 7 (02 de julio de 2019)

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

4.8. Examen 8 (11 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 59).

5

Exámenes del grupo 5

Andrés Cordón y Miguel A. Martínez

5.1. Examen 1 (30 de octubre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 1º examen de evaluación continua (30 de octubre de 2018)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    numDivisores :: Int -> Int -> [Int]
-- tal que (numDivisores n k) es la lista de los números enteros
-- positivos menores que n que tengan, al menos, k divisores. Por
-- ejemplo,
--    numDivisores  100 11 == [60,72,84,90,96]
--    numDivisores  500 30 == []
--    numDivisores 1000 30 == [720,840]
-- --------------------------------------------------------------------

numDivisores :: Int -> Int -> [Int]
numDivisores n k = [x | x <- [1..n-1], length (divisores x) >= k]

-- (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 x = [y | y <- [1..x], rem x y == 0]

-- 2ª definición:
```

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

numDivisores2 :: Int -> Int -> [Int]
numDivisores2 n k = reverse (aux n)
  where aux 0 = []
        aux x | length (divisores x) >= k = x : aux (x-1)
              | otherwise                 = aux (x-1)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    posiciones :: [a] -> [Int] -> [a]
-- tal que (posiciones xs ns) es la lista de elementos de xs que
-- aparecen en las posiciones indicadas por la lista de enteros ns,
-- descartando valores de posiciones no válidas (negativos o más allá
-- del tamaño de la lista). Por ejemplo,
--    posiciones [2,7,9,31,5,0] [3,1,10,-2] == [31,7]
--    posiciones "zamora" [2,3,7,0,3]       == "mozo"
-- ---------------------------------------------------------------------

posiciones :: [a] -> [Int] -> [a]
posiciones xs ns = [xs!!k | k <- ns, 0 <= k, k < length xs]

-- ----------------------------------------------------------------------
-- Ejercicio 3.1. Un triángulo de lados (a,b,c) es válido si cualquiera
-- de sus lados es menor que la suma de los otros dos.
--
-- Definir la función
--    valido :: (Float,Float,Float) -> Bool
-- tal que (valido (a,b,c)) se verifica si el triángulo de lados (a,b,c)
-- es válido. Por ejemplo,
--    valido (3,5,4) == True
--    valido (1,4,7) == False
-- ---------------------------------------------------------------------

valido ::(Float,Float,Float) -> Bool
valido (a,b,c) = a < b+c && b < a+c && c < a+b

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. La fórmula de Herón establece que el área de un
-- triángulo de lados a,b,c, y semiperimetro s=(a+b+c)/2 es igual a la
```

```
-- raíz cuadrada de s(s-a)(s-b)(s-c).
--
-- Definir la función
--    area :: (Float,Float,Float) -> Float
-- tal que (area t) es el área del triángulo t (o bien -1 si el
-- triángulo no es válido). Por ejemplo,
--    area (4,5,7)  == 9.797959
--    area (4,5,10) == -1.0
-- ---------------------------------------------------------------------

area :: (Float,Float,Float) -> Float
area (a,b,c)
  | valido (a,b,c) = sqrt (s*(s-a)*(s-b)*(s-c))
  | otherwise      = -1
  where s = (a+b+c)/2

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Una variante del clásico juego "piedra, papel,
-- tijeras" se obtiene añadiendo al juego "lagarto y Spock". A
-- continuación, se asigna a cada objeto del juego un número y se
-- especifica cuándo gana cada uno.
--   tijeras = 0 -- Tijeras cortan papel y decapitan lagarto
--   papel   = 1 -- Papel tapa a piedra y desautoriza a Spock
--   piedra  = 2 -- Aplasta a lagarto y a tijeras
--   lagarto = 3 -- Envenena a Spock y devora el papel
--   spock   = 4 -- Rompe tijeras y vaporiza piedra
--
-- Definir la función
--    tirada :: Int -> Int -> Int
-- tal que (tirada j1 j2) es el jugador (1 ó 2) gana la partida del
-- juego piedra, papel, tijeras, lagarto, Spock o 0, en caso de
-- empate. Por ejemplo,
--    tirada tijeras lagarto == 1
--    tirada piedra papel    == 2
--    tirada lagarto spock   == 1
--    tirada piedra piedra   == 0
-- ---------------------------------------------------------------------

tijeras, papel, piedra, lagarto, spock :: Int
tijeras = 0
```

```haskell
papel   = 1
piedra  = 2
lagarto = 3
spock   = 4

-- 1ª definición
-- =============

tirada :: Int -> Int -> Int
tirada j1 j2
  | j1 == j2                                             = 0
  | j1 == tijeras && (j2 == papel   || j2 == lagarto) = 1
  | j1 == papel   && (j2 == piedra  || j2 == spock)   = 1
  | j1 == piedra  && (j2 == lagarto || j2 == tijeras) = 1
  | j1 == lagarto && (j2 == spock   || j2 == papel)   = 1
  | j1 == spock   && (j2 == tijeras || j2 == piedra)  = 1
  | otherwise                                            = 2

-- 2ª definición
-- =============

tirada2 :: Int -> Int -> Int
tirada2 j1 j2
  | j1 == j2     = 0
  | gana1 j1 j2 = 1
  | otherwise    = 2

gana1 :: Int -> Int -> Bool
gana1 j1 j2 =
  (j1,j2) `elem` [ (tijeras, papel)
                 , (tijeras, lagarto)
                 , (papel,   piedra)
                 , (papel,   spock)
                 , (piedra,  lagarto)
                 , (piedra,  tijeras)
                 , (lagarto, spock)
                 , (lagarto, papel)
                 , (spock,   tijeras)
                 , (spock,   piedra)]
```

```haskell
-- ---------------------------------------------------------------
-- Ejercicio 4.2. Una partida es una lista de pares (donde el primer y
-- el segundo elemento del par son las jugadas del primer y segundo
-- jugador, respectivamente).
--
-- Definir la función
--    ganador :: [(Int,Int)] -> Int
-- tal que (ganador js) es el jugador que gana la mayoría de las jugadas
-- de la lista js (o bien 0 si hay un empate). Por ejemplo, para la
-- partida definida por
--    partida :: [(Int,Int)]
--    partida = [(tijeras,papel),(papel,tijeras),(piedra,spock),
--               (lagarto,papel),(piedra,papel)]
-- se tiene
--    ganador partida        = 2
--    ganador (init partida) = 0
--    ganador (take 1 partida)  ==  1
-- ---------------------------------------------------------------------

partida :: [(Int,Int)]
partida = [(tijeras,papel),(papel,tijeras),(piedra,spock),
           (lagarto,papel),(piedra,papel)]

ganador :: [(Int,Int)] -> Int
ganador js
  | ganadasPor 1 > ganadasPor 2 = 1
  | ganadasPor 1 < ganadasPor 2 = 2
  | otherwise                   = 0
  where ganadasPor j = length [(a,b) | (a,b) <- js, tirada a b == j]
```

5.2. Examen 2 (13 de diciembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 2º examen de evaluación continua (13 de diciembre de 2018)
-- ------------------------------------------------------------------

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

```haskell
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La sucesión de Hasler se define como sigue. Para cada
-- n >= 0, el término n-ésimo de la sucesión se calcula restando la suma
-- de los dígitos de n y el número de dígitos de n. Por ejemplo,
--    s(0)   = 0 - 1 = -1
--    s(10)  = 1 - 2 = -1
--    s(711) = 9 - 3 = 6
--
-- Definir la función
--    sucHasler :: [Int]
-- tal que (sucHasler) es la lista infinita que representa dicha
-- sucesión. Por ejemplo,
--    λ> take 30 sucHasler
--    [-1,0,1,2,3,4,5,6,7,8,-1,0,1,2,3,4,5,6,7,8,0,1,2,3,4,5,6,7,8,9]
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

sucHasler :: [Int]
sucHasler =
  [sum (digitos x) - length (digitos x) | x <- [0..]]

-- (digitos x) es la lista de los dígitos de x. Por ejemplo,
--    digitos 235  ==  [2,3,5]
digitos :: Int -> [Int]
digitos x = [read [c] | c <- show x]

-- 2ª definición
-- =============

sucHasler2 :: [Int]
sucHasler2 =
  map (\x -> sum (digitos x) - length (digitos x)) [0..]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    posicionHasler :: Int -> Int
```

```haskell
-- tal que (posicionHasler x) es la posición (empezando a contar por
-- cero) de la primera ocurrencia del elemento x en la sucesión de
-- Hasler. Por ejemplo,
--    posicionHasler 3    == 4
--    posicionHasler (-2) == 100
--    posicionHasler (-3) == 1000
-- ---------------------------------------------------------------------

-- 1ª definición
posicionHasler :: Int -> Int
posicionHasler x = head [b | (a,b) <- zip sucHasler [0..], a == x]

-- 2ª definición
posicionHasler2 :: Int -> Int
posicionHasler2 x = length (takeWhile (/= x) sucHasler)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    separa :: (a -> Bool) -> [a] -> ([a],[a])
-- tal que (separa p xs) es el par cuya primera componente son los
-- elementos de xs que cumplen la propiedad p y la segunda es la de los
-- que no la cumplen. Por ejemplo,
--    separa even [1,2,5,4,7] == ([2,4],[1,5,7])
--    separa (>3) [1..7]      == ([4,5,6,7],[1,2,3])
-- ---------------------------------------------------------

-- 1ª definición
separa :: (a -> Bool) -> [a] -> ([a],[a])
separa = partition

-- 2ª definición
separa2 :: (a -> Bool) -> [a] -> ([a],[a])
separa2 _ []                = ([],[])
separa2 p (x:xs) | p x       = (x:ys,zs)
                 | otherwise = (ys,x:zs)
  where (ys,zs) = separa2 p xs

-- 3ª definición
separa3 :: (a -> Bool) -> [a] -> ([a],[a])
separa3 p xs = (filter p xs, filter (not . p) xs)
```

```haskell
-- 4ª definición
separa4 :: (a -> Bool) -> [a] -> ([a],[a])
separa4 p = foldr (aux p) ([],[])
  where aux p x (ys,zs) | p x       = (x:ys,zs)
                        | otherwise = (ys,x:zs)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una lista de listas xss se dirá encadenada si
-- + todos sus elementos son listas de dos o más elementos, y
-- + los dos últimos elementos de cada lista de xss coinciden con los
--   dos primeros elementos de la siguiente.
--
-- Definir la función
--    encadenada :: Ord a => [[a]] -> Bool
-- tal que (encadenadaC xss) se verifica si xss es encadenada. Por
-- ejemplo,
--    encadenada [[1,2,3],[2,3,1,0],[1,0,1],[0,1,7]] == True
--    encadenada [[1,2,3],[2,3,3,3],[3,5,6]]         == False
--    encadenada [[1,2,3],[2,3,5,7],[7]]             == False
-- ---------------------------------------------------------------------

-- 1ª definición
encadenada :: Eq a => [[a]] -> Bool
encadenada xss =
  and [length xs >= 2 | xs <- xss] &&
  and [drop (length xs - 2) xs == take 2 ys
      | (xs,ys) <- zip xss (tail xss)]

-- 2ª definición
encadenada2 :: Eq a => [[a]] -> Bool
encadenada2 []   = True
encadenada2 [xs] = length xs >= 2
encadenada2 (xs:ys:zss) =
  length xs >= 2 &&
  length ys >= 2 &&
  drop (length xs - 2) xs == take 2 ys &&
  encadenada2 (ys:zss)

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

```haskell
-- Ejercicio 4. Una cadena de texto se dirá normalizada si:
-- + se han eliminado los espacios  iniciales,
-- + se han eliminado los espacios finales, y
-- + cada palabra está separada de la siguiente por un único espacio.
--
-- Definir la función
--    normaliza :: String -> String
-- tal que (normaliza xs) devuelve es la normalización de la cadena
-- xs. Por ejemplo,
--    normaliza "   Hoy    es    jueves    " == "Hoy es jueves"
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

normaliza :: String -> String
normaliza = unwords . words

-- 2ª definición
-- =============

normaliza2 :: String -> String
normaliza2 = frase . palabras

-- (palabras xs) es la lista de las palabras de cs. Por ejemplo,
--    λ> palabras "   Hoy    es    jueves    "
--    ["Hoy","es","jueves"]
palabras :: String -> [String]
palabras [] = []
palabras cs
  | null cs'  = []
  | otherwise = p : palabras cs''
  where cs'       = dropWhile (== ' ') cs
        (p,cs'') = break (== ' ') cs'

-- (frase ps) es la frase formada por las palabras de ps. Por ejemplo,
--    λ> frase ["Hoy","es","jueves"]
--    "Hoy es jueves"
frase :: [String] -> String
frase []     = ""
```

```haskell
frase (p:ps) = p ++ aux ps
  where aux []       = ""
        aux (p':ps') = ' ' : (p' ++ aux ps')
```

5.3. Examen 3 (22 de enero de 2019)

El examen es común con el del grupo 1 (ver página 133).

5.4. Examen 4 (21 de marzo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 4º examen de evaluación continua (21 de marzo de 2019)
-- ---------------------------------------------------------------------
-- ---------------------------------------------------------------------
-- § Librerías                                                       --
-- ---------------------------------------------------------------------

import Data.Numbers.Primes
import Data.List
import Data.Array
import Data.Maybe
import I1M.Pila

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Dos números primos a y b se dirán relacionados si
-- tienen el mismo número de cifras y se diferencian en, exactamente,
-- una de ellas. Por ejemplo, 3169 y 3119 están relacionados.
--
-- Definir la lista
--    primosRelacionados :: [(Integer,Integer)]
-- cuyos elementos son los pares (a,b), con 2 <= b < a, tales que a y b
--    λ> take 17 primosRelacionados
--    [(3,2),(5,2),(5,3),(7,2),(7,3),(7,5),(13,11),(17,11),(17,13),
--     (19,11),(19,13),(19,17),(23,13),(29,19),(29,23),(31,11),(37,17)]
-- ---------------------------------------------------------------------

primosRelacionados :: [(Integer,Integer)]
primosRelacionados =
  [(a,b) | a <- primes
```

```
         , b <- takeWhile (<a) primes
         , relacionados a b]

relacionados :: Integer -> Integer -> Bool
relacionados a b =
  length as == length bs &&
  length [(x,y) | (x,y) <- zip as bs, x /= y] == 1
  where as = show a
        bs = show b

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcular en qué posición aparece el par (3169,3119) en
-- la lista infinta primosRelacionados.
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> length (takeWhile(/=(3169,3119)) primosRelacionados)
--    1475
--
-- Otra forma de calcularlo es
--    λ> fromJust (elemIndex (3169,3119) primosRelacionados)
--    1475

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los árboles binarios mediante el tipo de
-- dato
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving Show
--
-- Define la función
--    ramaIgual :: Eq a => Arbol a -> Bool
-- tal que (ramaIgual x) se verifica si x tiene alguna rama con todos
-- los elementos de la rama iguales entre sí. Por ejemplo,
--    ramaIgual (N 2 (H 7) (N 2 (N 2 (H 0) (H 2)) (H 5)))  ==  True
--    ramaIgual (N 2 (H 7) (N 2 (N 3 (H 0) (H 2)) (H 5)))  == False
-- ---------------------------------------------------------------------

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

```
  deriving Show

ramaIgual :: Eq a => Arbol a -> Bool
ramaIgual  = any todosIguales . ramas

-- (ramas a) es la lista de las ramas del árbol a. Por ejemplo,
--    λ> ramas (N 2 (H 7) (N 2 (N 2 (H 0) (H 2)) (H 5)))
--    [[2,7],[2,2,2,0],[2,2,2,2],[2,2,5]]
ramas :: Arbol a -> [[a]]
ramas (H x)     = [[x]]
ramas (N x i d) = map (x:) (ramas i ++ ramas d)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Representamos la matrices mediante el tipo de dato
--    type Matriz a = Array (Int,Int) a
--
-- Un elemento de una matriz se dirá un mínimo local si es menor
-- estricto que todos sus vecinos. Por ejemplo, la matriz
--    ( 2,5,1,0 )
--    ( 1,7,4,8 )
--    ( 3,3,2,5 )
-- tiene tres mínimos locales que se encuentran en las posiciones
-- (1,4), (2,1) y (3,3).
--
-- Definir la función
--    posicionesMinimos :: Ord a => Matriz a -> [(Int,Int)]
-- tal que (posicionesMinimos p) es la lista de las posiciones de la
-- matriz p en la que p tiene un mínimo local. Por ejemplo,
--    λ> posicionesMinimos (listArray ((1,1),(3,4)) [2,5,1,0,1,7,4,8,3,3,2,5])
--    [(1,4),(2,1),(3,3)]
-- ---------------------------------------------------------------------
```

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

posicionesMinimos :: Ord a => Matriz a -> [(Int,Int)]
posicionesMinimos p =
    [(i,j) | (i,j) <- indices p,
             and [p!(i,j) < p!(a,b) | (a,b) <- vecinos (i,j)]]
    where (_,(m,n)) = bounds p
          vecinos (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                                   b <- [max 1 (j-1)..min n (j+1)],
                                   (a,b) /= (i,j)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    mayorSeg :: Eq a => Pila a -> [a]
-- tal que (mayorSeg q) es el mayor segmento inicial de la pila q que no
-- contiene ningún elemento repetido. Por ejemplo,
--    mayorSeg (foldr apila vacia [2,3,5,5,6])  ==  [2,3,5]
--    mayorSeg (foldr apila vacia [2,3,2,5,6])  ==  [2,3]
--    mayorSeg (foldr apila vacia [2,3,4,5,6])  ==  [2,3,4,5,6]
-- ---------------------------------------------------------------------

mayorSeg :: Eq a => Pila a -> [a]
mayorSeg = aux []
  where aux xs p
          | esVacia p   = reverse xs
          | elem cp xs = reverse xs
          | otherwise   = aux (cp:xs) dp
          where cp = cima p
                dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Una sucesión tipo Fibonacci de parámetros una lista de
-- números naturales ns y un número natural k >= 2, se define como
-- sigue:
--   a) para i entre 1 y k, a(i) es el i-ésimo elemento de ns
--      (completando con ceros si fuese necesario);
--   b) para cada i > k, a(i) el la suma de los k anteriores términos de
--      la sucesión.
--
-- Definir la función
```

```
--    fibTipo :: [Integer] -> Integer -> Integer -> Integer
-- tal que (fibTipo ns k x) devuelve el x-ésimo termino de la sucesión
-- tipo Fibonacci de parámetros ns y k. Por ejemplo:
--    fibTipo [1,1] 2 10 == 55
--    fibTipo [1] 4 10   == 15
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

fibTipo1 :: [Integer] -> Integer -> Integer -> Integer
fibTipo1 ns k x
  | x <= k    = (ns ++ repeat 0) `genericIndex` (x-1)
  | otherwise = sum [fibTipo1 ns k y | y <- [x-k..x-1]]

-- 2ª definición
-- =============

fibTipo2 :: [Integer] -> Integer -> Integer -> Integer
fibTipo2 ns k x = fibTipoVector ns k x ! x

fibTipoVector :: [Integer] -> Integer -> Integer -> Array Integer Integer
fibTipoVector ns k x = v
  where
    v = array (1,x) [(i,f i) | i <- [1..x]]
    f i | i <= k    = (ns ++ repeat 0) `genericIndex` (i-1)
        | otherwise = sum [v!j | j <- [i-k..i-1]]

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

--    λ> fibTipo1 [2,1] 3 26
--    1455549
--    (2.74 secs, 2,239,559,112 bytes)
--    λ> fibTipo2 [2,1] 3 26
--    1455549
--    (0.01 secs, 173,056 bytes)

-- En lo que sigue usaremos la 2ª definición
fibTipo :: [Integer] -> Integer -> Integer -> Integer
```

```haskell
fibTipo = fibTipo2

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Calcular el valor de (fibTipo [2,1] 3 99).
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> fibTipo [2,1] 3 99
--    3036939952156607693998O432
--    (0.01 secs, 320,768 bytes)
```

5.5. Examen 5 (16 de mayo de 2019)

```haskell
-- Informática (1º del Grado en Matemáticas), Grupo 5
-- 5º examen de evaluación continua (16 de mayo de 2019)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- § Librerías                                                        --
-- ---------------------------------------------------------------------

import Data.List
import Data.Maybe
import Data.Array
import I1M.Pol

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--     maxConUno :: [Integer] -> Maybe Integer
-- tal que (maxConUno xs) es (Just x) si x es el mayor elemento de la
-- lista xs que empieza por uno, o bien Nothing si xs no contiene ningún
-- elemento que empiece por uno. Por ejemplo,
--    maxConUno [23,1134,107,56,1117] == Just 1134
--    maxConUno [23,2311,99]          == Nothing
-- ---------------------------------------------------------------------

maxConUno :: [Integer] -> Maybe Integer
maxConUno xs
  | null ys   = Nothing
  | otherwise = Just (maximum ys)
```

```haskell
  where ys = filter (\x -> head (show x) == '1') xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los árboles binarios mediante el tipo
-- de dato
--    data Arbol a = H a | N a (Arbol a) (Arbol a)
--                   deriving Show
--
-- Definir la función
--    todasOninguno :: (a -> Bool) -> Arbol a -> Bool
-- tal que (todasOninguno p t) se verifica si todas las hojas del árbol t
-- satisfacen la propiedad p o ningún nodo interno del árbol t satisface
-- p. Por ejemplo,
--    todasOninguno even (N 1 (H 4) (N 2 (H 0) (H 8)))  ==  True
--    todasOninguno (>0) (N 1 (H 4) (N 2 (H 0) (H 8)))  ==  False
--    todasOninguno (>4) (N 1 (H 4) (N 2 (H 0) (H 8)))  ==  True
-- ---------------------------------------------------------------------

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

todasOninguno :: (a -> Bool) ->  Arbol a -> Bool
todasOninguno p a = todas p a || ninguno p a


-- (todas p t) se verifica si todas las hojas del árbol t satisfacen la
-- propiedad p. Por ejemplo,
--    todas even (N 1 (H 4) (N 2 (H 0) (H 8)))  ==  True
--    todas even (N 1 (H 4) (N 2 (H 1) (H 8)))  ==  False
todas :: (a -> Bool) ->  Arbol a -> Bool
todas p (H x)     = p x
todas p (N x i d) = todas p i && todas p d

-- (ninguno p t) se verifica si ningún nodo interno del árbol t
-- satisface p. Por ejemplo,
--    ninguno (>2) (N 1 (H 4) (N 2 (H 1) (H 8)))  ==  True
--    ninguno (>0) (N 1 (H 4) (N 2 (H 1) (H 8)))  ==  False
ninguno :: (a -> Bool) ->  Arbol a -> Bool
ninguno p (H x)     = True
ninguno p (N x i d) = (not . p) x && ninguno p i && ninguno p d
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3. Define la función
--    calculadora :: IO ()
-- que represente mediante entrada y salida la funcionalidad de una
-- calculadora con las siguientes operaciones: suma (+), resta (-),
-- división (/), multiplicación (*) y terminar (q). Un ejemplo de
-- interacción es el siguiente:
--   λ> calculadora
--    Introduzca una operación (+,-,/,*,q): sumar
--    Operación incorrecta
--    Introduzca una operación (+,-,/,*,q): +
--    Introduzca el primer operando: 2.3
--    Introduzca el segundo operando: 2.7
--    El resultado es 5.0
--    Introduzca una operación (+,-,/,*,q): q
--    Fin.
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

calculadora :: IO ()
calculadora = do
  putStr "Introduzca una operación (+,-,/,*,^,q): "
  os <- getLine
  if os == "q" then
    putStrLn "Fin"
  else if os `notElem` ["+","-","/","*","^"] then do
    putStrLn "Operación incorrecta"
    calculadora
  else do
    putStr "Introduzca el primer operando: "
    os1 <- getLine
    let op1 = read os1 :: Float
    putStr "Introduzca el segundo operando: "
    os2 <- getLine
    let op2 = read os2 :: Float
    let res = operacion os op1 op2
    putStrLn ("El resultado es " ++ show res)
```

```haskell
    calculadora

operacion :: String -> Float -> Float -> Float
operacion os op1 op2
   | os == "+" = op1 + op2
   | os == "-" = op1 - op2
   | os == "/" = op1 / op2
   | os == "*" = op1 * op2

-- 2ª solución
-- ===========

calculadora2 :: IO ()
calculadora2 = do
  putStr "Introduzca una operación (+,-,/,*,^,q): "
  os <- getLine
  if os == "q" then
    putStrLn "Fin"
  else if os `notElem` ["+","-","/","*","^"] then do
    putStrLn "Operación incorrecta"
    calculadora
  else do
    putStr "Introduzca el primer operando: "
    op1 <- read <$> getLine :: IO Float
    putStr "Introduzca el segundo operando: "
    op2 <- read <$> getLine :: IO Float
    putStrLn ("El resultado es " ++ show (operacion2 os op1 op2))
    calculadora

operacion2 :: String -> Float -> Float -> Float
operacion2 os =
  case os of
    "+" -> (+)
    "-" -> (-)
    "/" -> (/)
    "*" -> (*)

-- ---------------------------------------------------------------------
-- Ejercicio 4.  Representamos los polinomios mediante el TAD de los
-- Polinomios (I1M.Pol).
```

```haskell
--
-- Definir la función
--   combina :: (Ord a,Num a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (combina p q) es el polinomio formado a partir de p y q como
-- sigue:
--    1) Si para un grado m hay términos de grado m en ambos polinomios
--       p y q, entonces en la combinación se elige el coeficiente
--       menor en valor absoluto (o, en caso de coincidencia en valor
--       absoluto, el coeficiente positivo);
--    2) Si para un grado m solo hay términos de grado m en uno de los
--       dos polinomios, dicho término aparece tal cual en la
--       combinación.
-- Por ejemplo, si pol1 y pol2 son, respectivamente, los siguientes
-- polinomios
--    3*x^5 + -2*x^4 + 5*x + 11
--    7*x^4 + x^2 + -5*x + -3
-- entonces
--    λ> combina pol1 pol2
--    3*x^5 + -2*x^4 + x^2 + 5*x + -3
-- ---------------------------------------------------------------------

pol1, pol2 :: Polinomio Int
pol1 = consPol 5 3 (consPol 4 (-2) (consPol 1 5 (consPol 0 11 polCero)))
pol2 = consPol 4 7 (consPol 2 1 (consPol 1 (-5) (consPol 0 (-3) polCero)))

combina :: (Ord a,Num a) => Polinomio a -> Polinomio a -> Polinomio a
combina p1 p2
  | esPolCero p1 = p2
  | esPolCero p2 = p1
  | n1 == n2     = consPol n1 (combinaCoef b1 b2) (combina r1 r2)
  | n1 > n2      = consPol n1 b1 (combina r1 p2)
  | n1 < n2      = consPol n2 b2 (combina p1 r2)
  where n1 = grado p1
        n2 = grado p2
        b1 = coefLider p1
        b2 = coefLider p2
        r1 = restoPol p1
        r2 = restoPol p2
        combinaCoef a1 a2 | abs a1 == abs a2 = max a1 a2
                          | abs a1 <  abs a2 = a1
```

```haskell
                                    | abs a1 >  abs a2 = a2

-- ---------------------------------------------------------------------
-- Ejercicio 5. Representamos la matrices mediante el tipo de dato
--    type Matriz a = Array (Int,Int) a
--
-- Una matriz de ceros y unos se puede representar mediante una lista
-- de triples como sigue:
--   el triple (i,v,ps) aparece en la representación si v es el valor
--   minoritario de la fila i-ésima de la matriz (en caso de empate,
--   eligiremos el cero como valor minoritario); y ps es la lista
--   de las posiciones de v en la fila i-ésima.
-- Por ejemplo, la matriz
--      (1 0 1 0)
--      (0 0 1 0)
--      (1 1 1 1)
-- se representa por [(1,0,[2,4]),(2,1,[3]),(3,0,[])]
--
-- Definir la función
--    triples :: Matriz Int -> [(Int,Int,[Int])]
-- tal que (triples p) es la lista de los triples correspondientes a la
-- matriz p según lo arriba descrito. Por ejemplo,
--    λ> triples (listArray ((1,1),(3,4)) [1,0,1,0,0,0,0,1,1,1,1,1])
--    [(1,1,[1,3]),(2,1,[4]),(3,0,[])]
--    λ> triples (listArray ((1,1),(4,4)) [1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1])
--    [(1,1,[1]),(2,1,[2]),(3,1,[3]),(4,1,[4])]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

triples :: Matriz Int -> [(Int,Int,[Int])]
triples p = [(i,minelem i, minpos i) | i <- [1..m]]
  where
    (_,(m,n)) = bounds p
    minelem i | 2 * sum [p!(i,j) | j <- [1..n]] > n = 0
              | otherwise                            = 1
    minpos i  = [j | j <- [1..n], p!(i,j) == minelem i]
```

5.6. Examen 6 (10 de junio de 2019)

El examen es común con el del grupo 3 (ver página 121).

5.7. Examen 7 (02 de julio de 2019)

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

5.8. Examen 8 (11 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 59).

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 ŷ` 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.