Exámenes de "Programación funcional con Haskell"

Vol. 9 (Curso 2017–18)

José A. Alonso Jiménez

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

Sevilla, 20 de diciembre de 2018

Índice general

Introducción **5**

1 Exámenes del grupo 1 **7**

José A. Alonso

1.1 Examen 1 (27 de octubre de 2017) 7
1.2 Examen 2 (29 de noviembre de 2017) 10
1.3 Examen 3 (30 de enero de 2018) 16
1.4 Examen 4 (14 de marzo de 2018) 22
1.5 Examen 5 (2 de mayo de 2018) 31
1.6 Examen 6 (12 de junio de 2018) 37
1.7 Examen 7 (27 de junio de 2018) 45
1.8 Examen 8 (10 de septiembre de 2018) 58

2 Exámenes del grupo 2 **65**

Antonia M. Chávez

2.1 Examen 1 (30 de octubre de 2017) 65
2.2 Examen 2 (27 de noviembre de 2017) 68
2.3 Examen 3 (30 de enero de 2018) 71
2.4 Examen 4 (12 de marzo de 2018) 71
2.5 Examen 5 (30 de abril de 2018) 76
2.6 Examen 6 (12 de junio de 2018) 80
2.7 Examen 7 (27 de junio de 2018) 81
2.8 Examen 8 (10 de septiembre de 2018) 81

3 Exámenes del grupo 3 **83**

Francisco J. Martín

3.1 Examen 1 (2 de noviembre de 2017) 83
3.2 Examen 2 (30 de noviembre de 2017) 86
3.3 Examen 3 (30 de enero de 2018) 91
3.4 Examen 4 (8 de marzo de 2018) 91

3.5 Examen 5 (26 de abril de 2018) 95
3.6 Examen 6 (12 de junio de 2018) 101
3.7 Examen 7 (27 de junio de 2018) 101
3.8 Examen 8 (10 de septiembre de 2018) 101

4 Exámenes del grupo 4 **103**
María J. Hidalgo

4.1 Examen 1 (2 de noviembre de 2017) 103
4.2 Examen 2 (5 de diciembre de 2017) 107
4.3 Examen 3 (30 de enero de 2018) 113
4.4 Examen 4 (15 de marzo de 2018) 121
4.5 Examen 5 (3 de mayo de 2018) 128
4.6 Examen 6 (12 de junio de 2018) 137
4.7 Examen 7 (27 de junio de 2018) 144
4.8 Examen 8 (10 de septiembre de 2018) 144

5 Exámenes del grupo 5 **145**
Andrés Cordón y Miguel A. Martínez

5.1 Examen 1 (25 de octubre de 2017) 145
5.2 Examen 2 (18 de diciembre de 2017) 149
5.3 Examen 3 (30 de enero de 2018) 153
5.4 Examen 4 (21 de marzo de 2018) 153
5.5 Examen 5 (7 de mayo de 2018) 158
5.6 Examen 6 (12 de junio de 2018) 162
5.7 Examen 7 (27 de junio de 2018) 162
5.8 Examen 8 (10 de septiembre de 2018) 162

A Resumen de funciones predefinidas de Haskell **163**
A.1 Resumen de funciones sobre TAD en Haskell 165

B Método de Pólya para la resolución de problemas **169**
B.1 Método de Pólya para la resolución de problemas matemáticos . . 169
B.2 Método de Pólya para resolver problemas de programación 170

Bibliografía **173**

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 2017–18) 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 2017–18) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2017–18) [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 9º 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-17/temas/2017-18-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-17/ejercicios/ejercicios-I1M-2017.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol8
[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. 7 (Curso 2016–17) [12]

José A. Alonso
Sevilla, 20 de diciembre de 2018

[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

1

Exámenes del grupo 1

José A. Alonso

1.1. Examen 1 (27 de octubre de 2017)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (27 de octubre de 2017)
-- ---------------------------------------------------------------------

-- Nota: La puntuación de cada ejercicio es 2.5 puntos.

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La suma de la serie
--    1/3 + 1/15 + 1/35 + 1/63 + ...+ 1/(4*x^2-1) + ...
-- es 1/2.
--
-- Definir la función
--    sumaSerie:: Double -> Double
-- tal que (sumaSerie n) es la aproximación de 1/2 obtenida mediante n
-- términos de la serie. Por ejemplo,
--    sumaSerie 2   == 0.39999999999999997
--    sumaSerie 10  == 0.4761904761904761
--    sumaSerie 100 == 0.49751243781094495
```

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

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

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

-- La propiedad es
prop_SumaSerie :: Double -> Bool
prop_SumaSerie n = sumaSerie n < 0.5

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. La sombra de un número x es el que se obtiene borrando
-- las cifras de x que ocupan lugares impares (empezando a contar en
-- 0). Por ejemplo, la sombra de 123 es 13 ya que borrando el 2, que
-- ocupa la posición 1, se obtiene  el 13.
--
-- Definir la función
--    sombra :: Int -> Int
-- tal que (sombra x) es la sombra de x. Por ejemplo,
--    sombra 93245368790412345  ==  925670135
--    sombra 4736               ==  43
--    sombra 473                ==  43
--    sombra 47                 ==  4
--    sombra 4                  ==  4
-- ---------------------------------------------------------------------

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

sombra :: Int -> Int
sombra n = read [x | (x,n) <- zip (show n) [0..], even n]
```

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

sombra2 :: Int -> Int
sombra2 n = read (elementosEnPares (show n))

-- (elementosEnPares xs) es la lita de los elementos de xs en posiciones
-- pares. Por ejemplo,
--    elementosEnPares [4,7,3,6]  ==  [4,3]
--    elementosEnPares [4,7,3]    ==  [4,3]
--    elementosEnPares [4,7]      ==  [4]
--    elementosEnPares [4]        ==  [4]
--    elementosEnPares []         ==  []
elementosEnPares :: [a] -> [a]
elementosEnPares (x:y:zs) = x : elementosEnPares zs
elementosEnPares xs       = xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    cerosDelFactorial :: Integer -> Integer
-- tal que (cerosDelFactorial n) es el número de ceros en que termina el
-- factorial de n. Por ejemplo,
--    cerosDelFactorial 24 ==  4
--    cerosDelFactorial 25 ==  6
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

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

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

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
```

```haskell
ceros :: Integer -> Integer
ceros n | rem n 10 /= 0 = 0
        | otherwise     = 1 + ceros (div n 10)

-- 2ª definición
-- =============

cerosDelFactorial2 :: Integer -> Integer
cerosDelFactorial2 n | n < 5     = 0
                     | otherwise = m + cerosDelFactorial2 m
  where m = n `div` 5

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    todosDistintos :: Eq a => [a] -> Bool
-- tal que (todosDistintos xs) se verifica si todos los elementos de xs
-- son distintos. Por ejemplo,
--    todosDistintos [2,3,5,7,9]    ==  True
--    todosDistintos [2,3,5,7,9,3]  ==  False
--    todosDistintos "Betis"        ==  True
--    todosDistintos "Sevilla"      ==  False
--    ---------------------------------------------------------------------

todosDistintos :: Eq a => [a] -> Bool
todosDistintos []     = True
todosDistintos (x:xs) = x `notElem` xs && todosDistintos xs
```

1.2. Examen 2 (29 de noviembre de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (29 de noviembre de 2017)
-- ---------------------------------------------------------------------
```

```haskell
-- Nota: La puntuación de cada ejercicio es 2.5 puntos.

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

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    biparticiones :: Integer -> [(Integer,Integer)]
-- tal que (biparticiones n) es la lista de pares de números formados
-- por las primeras cifras de n y las restantes. Por ejemplo,
--    biparticiones  2025  ==  [(202,5),(20,25),(2,25)]
--    biparticiones 10000  ==  [(1000,0),(100,0),(10,0),(1,0)]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

biparticiones1 :: Integer -> [(Integer,Integer)]
biparticiones1 x = [(read y, read z) | (y,z) <- biparticionesL1 xs]
  where xs = show x

-- (biparticionesL1 xs) es la lista de los pares formados por los
-- prefijos no vacío de xs y su resto. Por ejemplo,
--    biparticionesL1 "2025" == [("2","025"),("20","25"),("202","5")]
biparticionesL1 :: [a] -> [([a],[a])]
biparticionesL1 xs = [splitAt k xs | k <- [1..length xs - 1]]

-- 2ª solución
-- ===========

biparticiones2 :: Integer -> [(Integer,Integer)]
biparticiones2 x = [(read y, read z) | (y,z) <- biparticionesL2 xs]
  where xs = show x

-- (biparticionesL2 xs) es la lista de los pares formados por los
-- prefijos no vacío de xs y su resto. Por ejemplo,
```

```haskell
--    biparticionesL2 "2025" == [("2","025"),("20","25"),("202","5")]
biparticionesL2 :: [a] -> [([a],[a])]
biparticionesL2 xs =
  takeWhile (not . null . snd) [splitAt n xs | n <- [1..]]

-- 3ª solución
-- ===========

biparticiones3 :: Integer -> [(Integer,Integer)]
biparticiones3 a =
  takeWhile ((>0) . fst) [divMod a (10^n) | n <- [1..]]

-- 4ª solución
-- ===========

biparticiones4 :: Integer -> [(Integer,Integer)]
biparticiones4 n =
  [quotRem n (10^x) | x <- [1..length (show n) -1]]

-- 5ª solución
-- ===========

biparticiones5 :: Integer -> [(Integer,Integer)]
biparticiones5 n =
  takeWhile (/= (0,n)) [divMod n (10^x) | x <- [1..]]

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

--    ghci> numero n = (read (replicate n '2')) :: Integer
--    (0.00 secs, 0 bytes)
--    ghci> length (biparticiones1 (numero 10000))
--    9999
--    (0.03 secs, 10,753,192 bytes)
--    ghci> length (biparticiones2 (numero 10000))
--    9999
--    (1.89 secs, 6,410,513,136 bytes)
--    ghci> length (biparticiones3 (numero 10000))
--    9999
--    (0.54 secs, 152,777,680 bytes)
```

```
--    ghci> length (biparticiones4 (numero 10000))
--    9999
--    (0.01 secs, 7,382,816 bytes)
--    ghci> length (biparticiones5 (numero 10000))
--    9999
--    (2.11 secs, 152,131,136 bytes)
--
--    ghci> length (biparticiones1 (numero (10^7)))
--    9999999
--    (14.23 secs, 10,401,100,848 bytes)
--    ghci> length (biparticiones4 (numero (10^7)))
--    9999999
--    (11.43 secs, 7,361,097,856 bytes)

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

-- 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
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios con valores enteros se pueden
-- representar con el tipo de dato algebraico
--    data Arbol = H
--               | N a Arbol Arbol
-- Por ejemplo, los árboles
--        3                7
--       / \              / \
--      2   4            5   8
--     / \   \          / \   \
--    1   3   5        6   4   10
--                        /   /
--                       9   1
-- se representan por
--    ej1, ej2 :: Arbol
--    ej1 = N 3 (N 2 (N 1 H H) (N 3 H H)) (N 4 H (N 5 H H))
--    ej2 = N 7 (N 5 (N 6 H H) (N 4 (N 9 H H) H)) (N 8 H (N 10 (N 1 H H) H))
--
-- Definir la función
--    suma :: Arbol -> Int
-- tal que (suma a) es la suma de todos los nodos a una distancia par
-- de la raíz del árbol a menos la suma de todos los nodos a una
-- distancia impar de la raíz. Por ejemplo,
--    suma ej1  ==  6
--    suma ej2  ==  4
-- ya que
--     (3 + 1+3+5) - (2+4)        = 6
--     (7 + 6+4+10) - (5+8 + 9+1) = 4
-- ---------------------------------------------------------------------

data Arbol = H
           | N Int Arbol Arbol

ej1, ej2 :: Arbol
ej1 = N 3 (N 2 (N 1 H H) (N 3 H H)) (N 4 H (N 5 H H))
```

```haskell
ej2 = N 7 (N 5 (N 6 H H) (N 4 (N 9 H H) H)) (N 8 H (N 10 (N 1 H H) H))

suma :: Arbol -> Int
suma H         = 0
suma (N x i d) = x - suma i - suma d

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    cercano :: (a -> Bool) -> Int -> [a] -> Maybe a
-- tal que (cercano p n xs) es el elemento de xs más cercano a n que
-- verifica la propiedad p. La búsqueda comienza en n y los elementos se
-- analizan en el siguiente orden: n, n+1, n-1, n+2, n-2,... Por ejemplo,
--    cercano (`elem` "aeiou") 6 "Sevilla"     ==  Just 'a'
--    cercano (`elem` "aeiou") 1 "Sevilla"     ==  Just 'e'
--    cercano (`elem` "aeiou") 2 "Sevilla"     ==  Just 'i'
--    cercano (`elem` "aeiou") 5 "Sevilla"     ==  Just 'a'
--    cercano (`elem` "aeiou") 9 "Sevilla"     ==  Just 'a'
--    cercano (`elem` "aeiou") (-3) "Sevilla"  ==  Just 'e'
--    cercano (`elem` "obcd")  1 "Sevilla"     ==  Nothing
--    cercano (>100) 4 [200,1,150,2,4]         ==  Just 150
--    cercano even 5 [1,3..99]                 ==  Nothing
--    cercano even 2 [1,4,6,8,0]               ==  Just 6
--    cercano even 2 [1,4,7,8,0]               ==  Just 8
--    cercano even 2 [1,4,7,5,0]               ==  Just 4
--    cercano even 2 [1,3,7,5,0]               ==  Just 0
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

cercano :: (a -> Bool) -> Int -> [a] -> Maybe a
cercano p n xs | null ys   = Nothing
               | otherwise = Just (head ys)
    where ys = filter p (ordenaPorCercanos xs n)

-- (ordenaPorCercanos xs n) es la lista de los elementos de xs que
-- ocupan las posiciones n, n+1, n-1, n+2, n-2... Por ejemplo,
--    ordenaPorCercanos [0..9] 4     ==  [4,5,3,6,2,7,1,8,0,9]
--    ordenaPorCercanos [0..9] 7     ==  [7,8,6,9,5,4,3,2,1,0]
--    ordenaPorCercanos [0..9] 2     ==  [2,3,1,4,0,5,6,7,8,9]
```

```haskell
--    ordenaPorCercanos [0..9] (-3)  ==  [0,1,2,3,4,5,6,7,8,9]
--    ordenaPorCercanos [0..9] 20    ==  [9,8,7,6,5,4,3,2,1,0]
ordenaPorCercanos :: [a] -> Int -> [a]
ordenaPorCercanos xs n
    | n < 0          = xs
    | n >= length xs = reverse xs
    | otherwise      = z : intercala zs (reverse ys)
    where (ys,(z:zs)) = splitAt n xs

-- (intercala xs ys) es la lista obtenida intercalando los elementos de
-- las lista xs e ys. Por ejemplo,
--    intercala [1..4] [5..10]   ==  [1,5,2,6,3,7,4,8,9,10]
--    intercala [5..10] [1..4]   ==  [5,1,6,2,7,3,8,4,9,10]
intercala :: [a] -> [a] -> [a]
intercala [] ys         = ys
intercala xs []         = xs
intercala (x:xs) (y:ys) = x : y : intercala xs ys

-- 2ª solución (usando find)
-- =========================

cercano2 :: (a -> Bool) -> Int -> [a] -> Maybe a
cercano2 p n xs = find p (ordenaPorCercanos xs n)
```

1.3. Examen 3 (30 de enero de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 4 y 5)
-- 3º examen de evaluación continua (30 de enero de 2018)
-- ---------------------------------------------------------------------

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

import Data.Char
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    alternadas :: String -> (String,String)
```

```haskell
-- tal que (alternadas cs) es el par de cadenas (xs,ys) donde xs es la
-- cadena obtenida escribiendo alternativamente en mayúscula o minúscula
-- las letras de la palabra cs (que se supone que es una cadena de
-- letras minúsculas) e ys se obtiene análogamente pero empezando en
-- minúscula. Por ejemplo,
--    λ> alternadas "salamandra"
--    ("SaLaMaNdRa","sAlAmAnDrA")
--    λ> alternadas "solosequenosenada"
--    ("SoLoSeQuEnOsEnAdA","sOlOsEqUeNoSeNaDa")
--    λ> alternadas (replicate 30 'a')
--    ("AaAaAaAaAaAaAaAaAaAaAaAaAaAa","aAaAaAaAaAaAaAaAaAaAaAaAaAaAaA")
-- ----------------------------------------------------------------------

-- 1ª solución
alternadas :: String -> (String,String)
alternadas []     = ([],[])
alternadas (x:xs) = (toUpper x : zs, x : ys)
  where (ys,zs) = alternadas xs

-- 2ª solución
alternadas2 :: String -> (String,String)
alternadas2 xs =
  ( [f x | (f,x) <- zip (cycle [toUpper,id]) xs]
  , [f x | (f,x) <- zip (cycle [id,toUpper]) xs]
  )

-- 3ª solución
alternadas3 :: String -> (String,String)
alternadas3 xs =
  ( zipWith ($) (cycle [toUpper,id]) xs
  , zipWith ($) (cycle [id,toUpper]) xs
  )

-- ----------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    biparticiones :: Integer -> [(Integer,Integer)]
-- tal que (biparticiones n) es la lista de pares de números formados
-- por las primeras cifras de n y las restantes. Por ejemplo,
--    biparticiones  2025  ==  [(202,5),(20,25),(2,25)]
--    biparticiones 10000  ==  [(1000,0),(100,0),(10,0),(1,0)]
```

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

-- 1ª solución
-- ===========

biparticiones :: Integer -> [(Integer,Integer)]
biparticiones x = [(read y, read z) | (y,z) <- particionesL1 xs]
  where xs = show x

-- (particionesL1 xs) es la lista de los pares formados por los
-- prefijos no vacío de xs y su resto. Por ejemplo,
--    particionesL1 "2025" == [("2","025"),("20","25"),("202","5")]
particionesL1 :: [a] -> [([a],[a])]
particionesL1 xs = [splitAt k xs | k <- [1..length xs - 1]]

-- 2ª solución
-- ===========

biparticiones2 :: Integer -> [(Integer,Integer)]
biparticiones2 x = [(read y, read z) | (y,z) <- particionesL2 xs]
  where xs = show x

-- (particionesL2 xs) es la lista de los pares formados por los
-- prefijos no vacío de xs y su resto. Por ejemplo,
--    particionesL2 "2025" == [("2","025"),("20","25"),("202","5")]
particionesL2 :: [a] -> [([a],[a])]
particionesL2 xs =
  takeWhile (not . null . snd) [splitAt n xs | n <- [1..]]

-- 3ª solución
-- ===========

biparticiones3 :: Integer -> [(Integer,Integer)]
biparticiones3 a =
  takeWhile ((>0) . fst) [divMod a (10^n) | n <- [1..]]

-- 4ª solución
-- ===========

biparticiones4 :: Integer -> [(Integer,Integer)]
```

```haskell
biparticiones4 n =
  [quotRem n (10^x) | x <- [1..length (show n) -1]]

-- 5ª solución
-- ===========

biparticiones5 :: Integer -> [(Integer,Integer)]
biparticiones5 n =
  takeWhile (/= (0,n)) [divMod n (10^x) | x <- [1..]]

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

--    λ> numero n = (read (replicate n '2')) :: Integer
--    (0.00 secs, 0 bytes)
--    λ> length (biparticiones (numero 10000))
--    9999
--    (0.03 secs, 10,753,192 bytes)
--    λ> length (biparticiones2 (numero 10000))
--    9999
--    (1.89 secs, 6,410,513,136 bytes)
--    λ> length (biparticiones3 (numero 10000))
--    9999
--    (0.54 secs, 152,777,680 bytes)
--    λ> length (biparticiones4 (numero 10000))
--    9999
--    (0.01 secs, 7,382,816 bytes)
--    λ> length (biparticiones5 (numero 10000))
--    9999
--    (2.11 secs, 152,131,136 bytes)
--
--    λ> length (biparticiones1 (numero (10^7)))
--    9999999
--    (14.23 secs, 10,401,100,848 bytes)
--    λ> length (biparticiones4 (numero (10^7)))
--    9999999
--    (11.43 secs, 7,361,097,856 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Un número x es construible a partir de a y b si se
```

```haskell
-- puede escribir como una suma cuyos sumandos son a o b (donde se
-- supone que a y b son números enteros mayores que 0). Por ejemplo, 7 y
-- 9 son construibles a partir de 2 y 3 ya que 7 = 2+2+3 y 9 = 3+3+3.
--
-- Definir la función
--    construibles  :: Integer -> Integer -> [Integer]
-- tal que (construibles a b) es la lista de los números construibles a
-- partir de a y b. Por ejemplo,
--    take 5 (construibles 2 9)  ==  [2,4,6,8,9]
--    take 5 (construibles 6 4)  ==  [4,6,8,10,12]
--    take 5 (construibles 9 7)  ==  [7,9,14,16,18]
-- ---------------------------------------------------------------------

-- 1ª definición
construibles :: Integer -> Integer -> [Integer]
construibles a b = tail aux
  where aux = 0 : mezcla [a + x | x <- aux]
                         [b + x | x <- aux]

mezcla :: [Integer] -> [Integer] -> [Integer]
mezcla p@(x:xs) q@(y:ys) | x < y     = x : mezcla xs q
                         | x > y     = y : mezcla p  ys
                         | otherwise = x : mezcla xs ys
mezcla []       ys                   = ys
mezcla xs       []                   = xs

-- 2ª definición
construibles2 :: Integer -> Integer -> [Integer]
construibles2 a b = filter (esConstruible2 a b) [1..]

-- Comparación de eficiencia
--    λ> construibles 2 9 !! 2000
--    2005
--    (0.02 secs, 1,133,464 bytes)
--    λ> construibles2 2 9 !! 2000
--    2005
--    (3.70 secs, 639,138,544 bytes)

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

```haskell
--    esConstruible :: Integer -> Integer -> Integer -> Bool
-- tal que (esConstruible a b x) se verifica si x es construible a
-- partir de a y b. Por ejemplo,
--    esConstruible 2 3 7   ==  True
--    esConstruible 9 7 15  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición
esConstruible :: Integer -> Integer -> Integer -> Bool
esConstruible a b x = x == y
  where (y:_) = dropWhile (<x) (construibles a b)

-- 2ª definición
esConstruible2 :: Integer -> Integer -> Integer -> Bool
esConstruible2 a b = aux
  where aux x
          | x < a && x < b  = False
          | otherwise       = x == a || x == b || aux (x-a) || aux (x-b)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Eq, Show)
--
-- Por ejemplo, el árbol
--          10
--         /  \
--        /    \
--       8      2
--      / \    / \
--     3   5  2   0
-- se pueden representar por
--    ejArbol :: Arbol Int
--    ejArbol = N 10 (N 8 (H 3) (H 5))
--                   (N 2 (H 2) (H 0))
--
-- Un árbol cumple la propiedad de la suma si el valor de cada nodo es
-- igual a la suma de los valores de sus hijos. Por ejemplo, el árbol
```

```haskell
-- anterior cumple la propiedad de la suma.
--
-- Definir la función
--    propSuma :: Arbol Int -> Bool
-- tal que (propSuma a) se verifica si el árbol a cumple la propiedad de
-- la suma. Por ejemplo,
--    λ> propSuma (N 10 (N 8 (H 3) (H 5)) (N 2 (H 2) (H 0)))
--    True
--    λ> propSuma (N 10 (N 8 (H 4) (H 5)) (N 2 (H 2) (H 0)))
--    False
--    λ> propSuma (N 10 (N 8 (H 3) (H 5)) (N 2 (H 2) (H 1)))
--    False
-- ---------------------------------------------------------------------

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

ejArbol :: Arbol Int
ejArbol = N 10 (N 8 (H 3) (H 5))
               (N 2 (H 2) (H 0))

propSuma :: Arbol Int -> Bool
propSuma (H _)     = True
propSuma (N x i d) = x == raiz i + raiz d && propSuma i && propSuma d

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

1.4. Examen 4 (14 de marzo de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 4º examen de evaluación continua (14 de marzo de 2018)
-- ---------------------------------------------------------------------

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

```haskell
import Data.List
import Data.Matrix
import Test.QuickCheck
import Graphics.Gnuplot.Simple
import Data.Function

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    menorPotencia :: Integer -> (Integer,Integer)
-- tal que (menorPotencia n) es el par (k,m) donde m es la menor
-- potencia de 2 que empieza por n y k es su exponentes (es decir,
-- 2^k = m). Por ejemplo,
--    menorPotencia 3             ==  (5,32)
--    menorPotencia 7             ==  (46,70368744177664)
--    fst (menorPotencia 982)     ==  3973
--    fst (menorPotencia 32627)   ==  28557
--    fst (menorPotencia 158426)  ==  40000
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

menorPotencia :: Integer -> (Integer,Integer)
menorPotencia n =
  head [(k,m) | (k,m) <- zip [0..] potenciasDe2
              , cs `isPrefixOf` show m]
  where cs = show n

-- potenciasDe 2 es la lista de las potencias de dos. Por ejemplo,
--    take 12 potenciasDe2  ==  [1,2,4,8,16,32,64,128,256,512,1024,2048]
potenciasDe2 :: [Integer]
potenciasDe2 = iterate (*2) 1

-- 2ª definición
-- =============

menorPotencia2 :: Integer -> (Integer,Integer)
menorPotencia2 n = aux (0,1)
  where aux (k,m) | cs `isPrefixOf` show m = (k,m)
```

```
                  | otherwise              = aux (k+1,2*m)
        cs = show n

-- 3ª definición
-- =============

menorPotencia3 :: Integer -> (Integer,Integer)
menorPotencia3 n =
  until (isPrefixOf n1 . show . snd) (\(x,y) -> (x+1,2*y)) (0,1)
  where n1 = show n

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

--    λ> maximum [fst (menorPotencia n) | n <- [1..1000]]
--    3973
--    (3.69 secs, 1,094,923,696 bytes)
--    λ> maximum [fst (menorPotencia2 n) | n <- [1..1000]]
--    3973
--    (5.13 secs, 1,326,382,872 bytes)
--    λ> maximum [fst (menorPotencia3 n) | n <- [1..1000]]
--    3973
--    (4.71 secs, 1,240,498,128 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    graficaMenoresExponentes :: Integer -> IO ()
-- tal que (graficaMenoresExponentes n) dibuja la gráfica de los
-- exponentes de 2 en las menores potencias de los n primeros números
-- enteros positivos.
-- ---------------------------------------------------------------------

graficaMenoresExponentes :: Integer -> IO ()
graficaMenoresExponentes n =
  plotList [ Key Nothing
           , PNG "Menor_potencia_de_2_que_comienza_por_n.png"
           ]
           (map (fst . menorPotencia) [1..n])

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

```haskell
-- Ejercicio 2. Definir la función
--    raizEnt :: Integer -> Integer -> Integer
-- tal que (raizEnt x n) es la raíz entera n-ésima de x; es decir, el
-- mayor número entero y tal que y^n <= x. Por ejemplo,
--    raizEnt  8 3      ==  2
--    raizEnt  9 3      ==  2
--    raizEnt 26 3      ==  2
--    raizEnt 27 3      ==  3
--    raizEnt (10^50) 2 ==  100000000000000000000000000
--
-- Comprobar con QuickCheck que para todo número natural n,
--     raizEnt (10^(2*n)) 2 == 10^n
-- ---------------------------------------------------------------------

-- 1ª definición
raizEnt1 :: Integer -> Integer -> Integer
raizEnt1 x n =
  last (takeWhile (\y -> y^n <= x) [0..])

-- 2ª definición
raizEnt2 :: Integer -> Integer -> Integer
raizEnt2 x n =
  floor ((fromIntegral x)**(1 / fromIntegral n))

-- Nota. La definición anterior falla para números grandes. Por ejemplo,
--    λ> raizEnt2 (10^50) 2 == 10^25
--    False

-- 3ª definición
raizEnt3 :: Integer -> Integer -> Integer
raizEnt3 x n = aux (1,x)
  where aux (a,b) | d == x    = c
                  | c == a    = c
                  | d < x     = aux (c,b)
                  | otherwise = aux (a,c)
          where c = (a+b) `div` 2
                d = c^n

-- Comparación de eficiencia
--    λ> raizEnt1 (10^14) 2
```

```
--    10000000
--    (6.15 secs, 6,539,367,976 bytes)
--    λ> raizEnt2 (10^14) 2
--    10000000
--    (0.00 secs, 0 bytes)
--    λ> raizEnt3 (10^14) 2
--    10000000
--    (0.00 secs, 25,871,944 bytes)
--
--    λ> raizEnt2 (10^50) 2
--    99999999999999998758486016
--    (0.00 secs, 0 bytes)
--    λ> raizEnt3 (10^50) 2
--    100000000000000000000000000
--    (0.00 secs, 0 bytes)

-- La propiedad es
prop_raizEnt :: (Positive Integer) -> Bool
prop_raizEnt (Positive n) =
  raizEnt3 (10^(2*n)) 2 == 10^n

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles se pueden representar mediante el siguiente
-- tipo de datos
--    data Arbol a = N a [Arbol a]
--                 deriving Show
-- Por ejemplo, los árboles
--      1               3
--     / \             /|\
--    2   3           / | \
--        |          5  4  7
--        4          |    /|\
--                   6   2 8 6
-- se representan por
--    ejArbol1, ejArbol2 :: Arbol Int
--    ejArbol1 = N 1 [N 2 [], N 3 [N 4 []]]
```

```
--    ejArbol2 = N 3 [N 5 [N 6 []],
--                    N 4 [],
--                    N 7 [N 2 [], N 8 [], N 6 []]]
--
-- Definir la función
--    nodosSumaMaxima :: (Num t, Ord t) => Arbol t -> [t]
-- tal que (nodosSumaMaxima a) es la lista de los nodos del
-- árbol a cuyos hijos tienen máxima suma. Por ejemplo,
--    nodosSumaMaxima ejArbol1  ==  [1]
--    nodosSumaMaxima ejArbol2  ==  [7,3]
-- ---------------------------------------------------------------------

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

ejArbol1, ejArbol2 :: Arbol Int
ejArbol1 = N 1 [N 2 [], N 3 [N 4 []]]
ejArbol2 = N 3 [N 5 [N 6 []],
                N 4 [],
                N 7 [N 2 [], N 8 [], N 6 []]]

-- 1ª solución
-- ===========

nodosSumaMaxima :: (Num t, Ord t) => Arbol t -> [t]
nodosSumaMaxima a =
  [x | (s,x) <- ns, s == m]
  where ns = reverse (sort (nodosSumas a))
        m  = fst (head ns)

-- (nodosSumas x) es la lista de los pares (s,n) donde n es un nodo del
-- árbol x y s es la suma de sus hijos. Por ejemplo,
--    λ> nodosSumas ejArbol1
--    [(5,1),(0,2),(4,3),(0,4)]
--    λ> nodosSumas ejArbol2
--    [(16,3),(6,5),(0,6),(0,4),(16,7),(0,2),(0,8),(0,6)]
nodosSumas :: Num t => Arbol t -> [(t,t)]
nodosSumas (N x []) = [(0,x)]
nodosSumas (N x as) = (sum (raices as),x) : concatMap nodosSumas as
```

```haskell
-- (raices b) es la lista de las raíces del bosque b. Por ejemplo,
--    raices [ejArbol1,ejArbol2]  ==  [1,3]
raices :: [Arbol t] -> [t]
raices = map raiz

-- (raiz a) es la raíz del árbol a. Por ejemplo,
--    raiz ejArbol1  ==  1
--    raiz ejArbol2  ==  3
raiz :: Arbol t -> t
raiz (N x _) = x

-- 2ª solución
-- ===========

nodosSumaMaxima2 :: (Num t, Ord t) => Arbol t -> [t]
nodosSumaMaxima2 a =
  [x | (s,x) <- ns, s == m]
  where ns = sort (nodosOpSumas a)
        m  = fst (head ns)

-- (nodosOpSumas x) es la lista de los pares (s,n) donde n es un nodo del
-- árbol x y s es el opuesto de la suma de sus hijos. Por ejemplo,
--    λ> nodosOpSumas ejArbol1
--    [(-5,1),(0,2),(-4,3),(0,4)]
--    λ> nodosOpSumas ejArbol2
--    [(-16,3),(-6,5),(0,6),(0,4),(-16,7),(0,2),(0,8),(0,6)]
nodosOpSumas :: Num t => Arbol t -> [(t,t)]
nodosOpSumas (N x []) = [(0,x)]
nodosOpSumas (N x as) = (-sum (raices as),x) : concatMap nodosOpSumas as


-- 3ª solución
-- ===========

nodosSumaMaxima3 :: (Num t, Ord t) => Arbol t -> [t]
nodosSumaMaxima3 a =
  [x | (s,x) <- ns, s == m]
  where ns = sort (nodosOpSumas a)
        m  = fst (head ns)
```

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

nodosSumaMaxima4 :: (Num t, Ord t) => Arbol t -> [t]
nodosSumaMaxima4 a =
  map snd (head (groupBy (\p q -> fst p == fst q)
                         (sort (nodosOpSumas a))))

-- 5ª solución
-- ===========

nodosSumaMaxima5 :: (Num t, Ord t) => Arbol t -> [t]
nodosSumaMaxima5 a =
  map snd (head (groupBy ((==) `on` fst)
                         (sort (nodosOpSumas a))))

-- 6ª solución
-- ===========

nodosSumaMaxima6 :: (Num t, Ord t) => Arbol t -> [t]
nodosSumaMaxima6 =
  map snd
  . head
  . groupBy ((==) `on` fst)
  . sort
  . nodosOpSumas

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    ampliaMatriz :: Matrix a -> Int -> Int -> Matrix a
-- tal que (ampliaMatriz p f c) es la matriz obtenida a partir de p
-- repitiendo cada fila f veces y cada columna c veces. Por ejemplo, si
-- ejMatriz es la matriz definida por
--    ejMatriz :: Matrix Char
--    ejMatriz = fromLists [" x ",
--                          "x x",
--                          " x "]
-- entonces
--    λ> ampliaMatriz ejMatriz 1 2
--    ( ' ' ' ' 'x' 'x' ' ' ' ' )
```

```
--    ( 'x' 'x' ' ' ' ' ' 'x' 'x' )
--    ( ' ' ' ' 'x' 'x' ' ' ' ' ' )
--
--    λ> (putStr . unlines . toLists) (ampliaMatriz ejMatriz 1 2)
--      xx
--    xx  xx
--      xx
--    λ> (putStr . unlines . toLists) (ampliaMatriz ejMatriz 2 1)
--     x
--     x
--    x x
--    x x
--     x
--     x
--    λ> (putStr . unlines . toLists) (ampliaMatriz ejMatriz 2 2)
--      xx
--      xx
--    xx  xx
--    xx  xx
--      xx
--      xx
--    λ> (putStr . unlines . toLists) (ampliaMatriz ejMatriz 2 3)
--       xxx
--       xxx
--    xxx   xxx
--    xxx   xxx
--       xxx
--       xxx
-- ----------------------------------------------------------------------

ejMatriz :: Matrix Char
ejMatriz = fromLists [" x ",
                      "x x",
                      " x "]

-- 1ª definición
-- =============

ampliaMatriz :: Matrix a -> Int -> Int -> Matrix a
ampliaMatriz p f c =
```

```
  ampliaColumnas (ampliaFilas p f) c

ampliaFilas :: Matrix a -> Int -> Matrix a
ampliaFilas p f =
  matrix (f*m) n (\(i,j) -> p!(1 + (i-1) `div` f, j))
  where m = nrows p
        n = ncols p

ampliaColumnas :: Matrix a -> Int -> Matrix a
ampliaColumnas p c =
  matrix m (c*n) (\(i,j) -> p!(i,1 + (j-1) `div` c))
  where m = nrows p
        n = ncols p

-- 2ª definición
-- =============

ampliaMatriz2 :: Matrix a -> Int -> Int -> Matrix a
ampliaMatriz2 p f c =
  ( fromLists
  . concatMap (map (concatMap (replicate c)) . replicate f)
  . toLists) p

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

ejemplo :: Int -> Matrix Int
ejemplo n = fromList n n [1..]

--    λ> maximum (ampliaMatriz (ejemplo 10) 100 200)
--    100
--    (6.44 secs, 1,012,985,584 bytes)
--    λ> maximum (ampliaMatriz2 (ejemplo 10) 100 200)
--    100
--    (2.38 secs, 618,096,904 bytes)
```

1.5. Examen 5 (2 de mayo de 2018)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 5º examen de evaluación continua (2 de mayo de 2018)
```

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

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

import Data.List
import Data.Numbers.Primes
import Data.Array
import I1M.PolOperaciones
import qualified Data.Map as M

-- ---------------------------------------------------------------------
-- Ejercicio 1. Se considera una enumeración de los números primos:
--     p(1) = 2, p(2) = 3, p(3) = 5, p(4) = 7, p(5) = 11, ...
--
-- Dado un entero x > 1, su altura prima es el mayor i tal que el
-- primo p(i) aparece en la factorización de x en números primos. Por
-- ejemplo, la altura prima de 3500 es 4, pues 3500=2^2*5^3*7^1 y la de
-- 34 tiene es 7, pues 34 = 2*17. Ademaś, la altura prima de 1 es 0.
--
-- Definir la función
--    alturasPrimas      :: Integer -> [Integer]
-- tal que (alturasPrimas n) es la lista de las alturas primas de los
-- primeros n números enteros positivos. Por ejemplo,
--    alturasPrimas 15  ==  [0,1,2,1,3,2,4,1,2,3,5,2,6,4,3]
--    maximum (alturasPrimas 10000)  ==  1229
--    maximum (alturasPrimas 20000)  ==  2262
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

alturasPrimas :: Integer -> [Integer]
alturasPrimas n = map alturaPrima [1..n]

-- (alturaPrima x) es la altura prima de x. Por ejemplo,
--    alturaPrima 3500  ==  4
--    alturaPrima 34    ==  7
alturaPrima :: Integer -> Integer
```

```haskell
alturaPrima 1 = 0
alturaPrima n = indice (mayorFactorPrimo n)

-- (mayorFactorPrimo n) es el mayor factor primo de n. Por ejemplo,
--    mayorFactorPrimo 3500  ==  7
--    mayorFactorPrimo 34    ==  17
mayorFactorPrimo :: Integer -> Integer
mayorFactorPrimo = last . primeFactors

-- (indice p) es el índice de p en la sucesión de los números
-- primos. Por ejemplo,
--    indice 7   ==  4
--    indice 17  ==  7
indice :: Integer -> Integer
indice p = genericLength (takeWhile (<=p) primes)

-- 2ª definición
-- =============

alturasPrimas2 :: Integer -> [Integer]
alturasPrimas2 n = map alturaPrima2 [1..n]

alturaPrima2 :: Integer -> Integer
alturaPrima2 n = v ! n
  where v = array (1,n) [(i,f i) | i <- [1..n]]
        f 1 = 0
        f k | isPrime k = indice2 k
            | otherwise = v ! k `div` head (primeFactors k)

indice2 :: Integer -> Integer
indice2 p = head [n | (x,n) <- indicesPrimos, x == p]

-- indicesPrimos es la sucesión formada por los números primos y sus
-- índices. Por ejemplo,
--    λ> take 10 indicesPrimos
--    [(2,1),(3,2),(5,3),(7,4),(11,5),(13,6),(17,7),(19,8),(23,9),(29,10)]
indicesPrimos :: [(Integer,Integer)]
indicesPrimos = zip primes [1..]

-- 3ª definición
```

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

alturasPrimas3 :: Integer -> [Integer]
alturasPrimas3 n = elems v
  where v = array (1,n) [(i,f i) | i <- [1..n]]
        f 1 = 0
        f k | isPrime k = indice2 k
            | otherwise = v ! k `div` head (primeFactors k)

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

--    λ> maximum (alturasPrimas 5000)
--    669
--    (2.30 secs, 1,136,533,912 bytes)
--    λ> maximum (alturasPrimas2 5000)
--    669
--    (12.51 secs, 3,595,318,584 bytes)
--    λ> maximum (alturasPrimas3 5000)
--    669
--    (0.27 secs, 75,110,896 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una partición prima de un número natural n es un
-- conjunto de primos cuya suma es n. Por ejemplo, el número 7 tiene 7
-- particiones primas ya que
--    7 = 7 = 5 + 2 = 3 + 2 + 2
--
-- Definir la función
--    particiones :: Int -> [[Int]]
-- tal que (particiones n) es el conjunto de las particiones primas de
-- n. Por ejemplo,
--    particiones 7             ==  [[7],[5,2],[3,2,2]]
--    particiones 8             ==  [[5,3],[3,3,2],[2,2,2,2]]
--    particiones 9             ==  [[7,2],[5,2,2],[3,3,3],[3,2,2,2]]
--    length (particiones 90)   ==  20636
--    length (particiones 100)  ==  40899
-- ---------------------------------------------------------------------

-- 1ª solución
```

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

particiones1 :: Int -> [[Int]]
particiones1 0 = [[]]
particiones1 n = [x:y | x <- xs,
                        y <- particiones1 (n-x),
                        [x] >= take 1 y]
  where xs = reverse (takeWhile (<= n) primes)

-- 2ª solución (con programación dinámica)
-- =======================================

particiones2 :: Int -> [[Int]]
particiones2 n = vectorParticiones n ! n

-- (vectorParticiones n) es el vector con índices de 0 a n tal que el
-- valor del índice k es la lista de las particiones primas de k. Por
-- ejemplo,
--    λ> mapM_ print (elems (vectorParticiones 9))
--    [[]]
--    []
--    [[2]]
--    [[3]]
--    [[2,2]]
--    [[5],[3,2]]
--    [[3,3],[2,2,2]]
--    [[7],[5,2],[3,2,2]]
--    [[5,3],[3,3,2],[2,2,2,2]]
--    [[7,2],[5,2,2],[3,3,3],[3,2,2,2]]
--    λ> elems (vectorParticiones 9) == map particiones1 [0..9]
--    True
vectorParticiones :: Int -> Array Int [[Int]]
vectorParticiones n = v where
  v = array (0,n) [(i,f i) | i <- [0..n]]
    where f 0 = [[]]
          f m = [x:y | x <- xs,
                       y <- v ! (m-x),
                       [x] >= take 1 y]
            where xs = reverse (takeWhile (<= m) primes)
```

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

--    λ> length (particiones1 35)
--    175
--    (5.88 secs, 2,264,266,040 bytes)
--    λ> length (particiones2 35)
--    175
--    (0.02 secs, 1,521,560 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    polDiagonal :: Array (Int,Int) Int  -> Polinomio Int
-- tal que (polDiagonal p) es el polinomio cuyas raíces son los
-- elementos de la diagonal de la matriz cuadrada p. Por ejemplo,
--    λ> polDiagonal (listArray ((1,1),(2,2)) [1..])
--    x^2 + -5*x + 4
-- ya que los elementos de la diagonal son 1 y 4 y
--    (x - 1) * (x - 4) = x^2 + -5*x + 4
-- Otros ejemplos
--    λ> polDiagonal (listArray ((1,1),(3,4)) [-12,-11..1])
--    x^3 + 21*x^2 + 122*x + 168
--    λ> polDiagonal (listArray ((1,1),(4,3)) [-12,-11..1])
--    x^3 + 24*x^2 + 176*x + 384
-- ---------------------------------------------------------------------

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

-- (diagonal p) es la lista de los elementos de la diagonal de la matriz
-- p. Por ejemplo,
--    diagonal (listArray ((1,1),(3,3)) [1..])  ==  [1,5,9]
diagonal :: Num a => Array (Int,Int) a -> [a]
diagonal p = [p ! (i,i) | i <- [1..min m n]]
    where (_,(m,n)) = bounds p

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

```haskell
multListaPol (p:ps) = multPol p (multListaPol ps)

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. El inverso de un diccionario d es el diccionario que a
-- cada valor x le asigna la lista de claves cuyo valor en d es x. Por
-- ejemplo, el inverso de
--    [("a",3),("b",2),("c",3),("d",2),("e",1)])
-- es
--    [(1,["e"]),(2,["d","b"]),(3,["c","a"])]
--
-- Definir la función
--    inverso :: (Ord k, Ord v) => M.Map k v -> M.Map v [k]
-- tal que (inverso d) es el inverso del diccionario d. Por ejemplo,
--    λ> inverso (M.fromList [("a",3),("b",2),("c",3),("d",2),("e",1)])
--    fromList [(1,["e"]),(2,["d","b"]),(3,["c","a"])]
--    λ> inverso (M.fromList [(x,x^2) | x <- [-3,-2..3]])
--    fromList [(0,[0]),(1,[1,-1]),(4,[2,-2]),(9,[3,-3])]
-- ---------------------------------------------------------------------

-- 1ª definición
inverso :: (Ord k, Ord v) => M.Map k v -> M.Map v [k]
inverso d = M.fromListWith (++) [(y,[x]) | (x,y) <- M.assocs d]

-- 2ª definición
inverso2 :: (Ord k, Ord v) => M.Map k v -> M.Map v [k]
inverso2 d
  | M.null d  = M.empty
  | otherwise = M.insertWith (++) y [x] (inverso2 e)
  where ((x,y),e) = M.deleteFindMin d
```

1.6. Examen 6 (12 de junio de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 4 y 5)
-- 6º examen de evaluación continua (12 de junio de 2018)
-- ---------------------------------------------------------------------
```

```haskell
-- Nota: La puntuación de cada ejercicio es 2.5 puntos.

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

import Data.List
import Data.Matrix
import I1M.PolOperaciones
import Test.QuickCheck
import qualified Data.Set as S

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La sucesión de polinomios de Fibonacci se define por
--    p(0) = 0
--    p(1) = 1
--    p(n) = x*p(n-1) + p(n-2)
-- Los primeros términos de la sucesión son
--    p(2) = x
--    p(3) = x^2 + 1
--    p(4) = x^3 + 2*x
--    p(5) = x^4 + 3*x^2 + 1
--
-- Definir la lista
--    sucPolFib :: [Polinomio Integer]
-- tal que sus elementos son los polinomios de Fibonacci. Por ejemplo,
--    λ> take 7 sucPolFib
--    [0,1,1*x,x^2 + 1,x^3 + 2*x,x^4 + 3*x^2 + 1,x^5 + 4*x^3 + 3*x]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

sucPolFib :: [Polinomio Integer]
sucPolFib = [polFibR n | n <- [0..]]

polFibR :: Integer -> Polinomio Integer
polFibR 0 = polCero
polFibR 1 = polUnidad
polFibR n =
```

```
  sumaPol (multPol (consPol 1 1 polCero) (polFibR (n-1)))
          (polFibR (n-2))

-- 2ª definición (dinámica)
-- ========================

sucPolFib2 :: [Polinomio Integer]
sucPolFib2 =
  polCero : polUnidad : zipWith f (tail sucPolFib2) sucPolFib2
  where f p = sumaPol (multPol (consPol 1 1 polCero) p)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que el valor del n-ésimo
-- término de sucPolFib para x=1 es el n-ésimo término de la sucesión de
-- Fibonacci 0, 1, 1, 2, 3, 5, 8, ...
--
-- Nota. Limitar la búsqueda a ejemplos pequeños usando
--    quickCheckWith (stdArgs {maxSize=5}) prop_polFib
-- ---------------------------------------------------------------------

prop_polFib :: Integer -> Property
prop_polFib n =
  n >= 0 ==> valor (polFib n) 1 == fib n
  where polFib n = sucPolFib2 `genericIndex` n
        fib n    = fibs `genericIndex` n

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

-- La comprobación es
--    ghci> quickCheckWith (stdArgs {maxSize=5}) prop_polFib
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las expresiones aritméticas pueden representarse usando
-- el siguiente tipo de datos
--    data Expr = N Int | S Expr Expr | P Expr Expr
--      deriving (Eq, Ord, Show)
-- Por ejemplo, la expresión 2*(3+7) se representa por
--    P (N 2) (S (N 3) (N 7))
```

```haskell
--
-- Definir la función
--    subexpresiones :: Expr -> S.Set Expr
-- tal que (subexpresiones e) es el conjunto de las subexpresiones de
-- e. Por ejemplo,
--    λ> subexpresiones (S (N 2) (N 3))
--    fromList [N 2,N 3,S (N 2) (N 3)]
--    λ> subexpresiones (P (S (N 2) (N 2)) (N 7))
--    fromList [N 2,N 7,S (N 2) (N 2),P (S (N 2) (N 2)) (N 7)]
-- ---------------------------------------------------------------------

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

subexpresiones :: Expr -> S.Set Expr
subexpresiones (N x)   = S.singleton (N x)
subexpresiones (S i d) =
  S i d `S.insert` (subexpresiones i `S.union` subexpresiones d)
subexpresiones (P i d) =
  P i d `S.insert` (subexpresiones i `S.union` subexpresiones d)

-- ---------------------------------------------------------------------
-- Ejercicio 3. El triángulo de Pascal es un triángulo de números
--          1
--         1 1
--        1 2 1
--       1  3 3  1
--      1 4  6  4 1
--     1 5 10 10 5 1
--    ...............
-- construido de la siguiente forma
-- + la primera fila está formada por el número 1;
-- + las filas siguientes se construyen sumando los números adyacentes
--   de la fila superior y añadiendo un 1 al principio y al final de la
--   fila.
--
-- La matriz de Pascal es la matriz cuyas filas son los elementos de la
-- correspondiente fila del triángulo de Pascal completadas con
-- ceros. Por ejemplo, la matriz de Pascal de orden 6 es
--    |1 0  0  0 0 0|
```

```
--    |1 1  0  0 0 0|
--    |1 2  1  0 0 0|
--    |1 3  3  1 0 0|
--    |1 4  6  4 1 0|
--    |1 5 10 10 5 1|
--
-- Definir la función
--    matrizPascal :: Int -> Matriz Int
-- tal que (matrizPascal n) es la matriz de Pascal de orden n. Por
-- ejemplo,
--    λ> matrizPascal 6
--    (  1  0  0  0  0  0 )
--    (  1  1  0  0  0  0 )
--    (  1  2  1  0  0  0 )
--    (  1  3  3  1  0  0 )
--    (  1  4  6  4  1  0 )
--    (  1  5 10 10  5  1 )
-- ----------------------------------------------------------------------

-- 1ª solución
-- ===========

matrizPascal :: Int -> Matrix Integer
matrizPascal 1 = fromList 1 1 [1]
matrizPascal n = matrix n n f
  where f (i,j) | i < n && j <  n  = p!(i,j)
                | i < n && j == n  = 0
                | j == 1 || j == n = 1
                | otherwise        = p!(i-1,j-1) + p!(i-1,j)
        p = matrizPascal (n-1)

-- 2ª solución
-- ===========

matrizPascal2 :: Int -> Matrix Integer
matrizPascal2 n = fromLists xss
  where yss = take n pascal
        xss = map (take n) (map (++ repeat 0) yss)

pascal :: [[Integer]]
```

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

-- 3ª solución
-- ===========

matrizPascal3 :: Int -> Matrix Integer
matrizPascal3 n =  matrix n n f
  where f (i,j) | i >=  j   = comb (i-1) (j-1)
                | otherwise = 0

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

-- 4ª solución
-- ===========

matrizPascal4 :: Int -> Matrix Integer
matrizPascal4 n = p
  where p = matrix n n (\(i,j) -> f i j)
        f i 1 = 1
        f i j
          | j >  i    = 0
          | i == j    = 1
          | otherwise = p!(i-1,j) + p!(i-1,j-1)

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

--    λ> maximum (matrizPascal 150)
--    46413034868354394849492907436302560970058760
--    (2.58 secs, 394,030,504 bytes)
--    λ> maximum (matrizPascal2 150)
--    46413034868354394849492907436302560970058760
--    (0.03 secs, 8,326,784 bytes)
--    λ> maximum (matrizPascal3 150)
```

```
--    46413034868354394849492907436302560970058760
--    (0.38 secs, 250,072,360 bytes)
--    λ> maximum (matrizPascal4 150)
--    46413034868354394849492907436302560970058760
--    (0.10 secs, 13,356,360 bytes)
--
--    λ> length (show (maximum (matrizPascal2 300)))
--    89
--    (0.06 secs, 27,286,296 bytes)
--    λ> length (show (maximum (matrizPascal3 300)))
--    89
--    (2.74 secs, 2,367,037,536 bytes)
--    λ> length (show (maximum (matrizPascal4 300)))
--    89
--    (0.36 secs, 53,934,792 bytes)
--
--    λ> length (show (maximum (matrizPascal2 700)))
--    209
--    (0.83 secs, 207,241,080 bytes)
--    λ> length (show (maximum (matrizPascal4 700)))
--    209
--    (2.22 secs, 311,413,008 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    sumas  :: Int -> [[Int]]
-- tal que (sumas n) es la lista de las descomposiciones de n como sumas
-- cuyos sumandos son 1 ó 2. Por ejemplo,
--    sumas 1            ==  [[1]]
--    sumas 2            ==  [[1,1],[2]]
--    sumas 3            ==  [[1,1,1],[1,2],[2,1]]
--    sumas 4            ==  [[1,1,1,1],[1,1,2],[1,2,1],[2,1,1],[2,2]]
-- ---------------------------------------------------------------------

-- 1ª definición
sumas1 :: Int -> [[Int]]
sumas1 0 = [[]]
sumas1 1 = [[1]]
sumas1 n = [1:xs | xs <- sumas1 (n-1)] ++ [2:xs | xs <- sumas1 (n-2)]
```

```haskell
-- 2ª definición
sumas2 :: Int -> [[Int]]
sumas2 n = aux !! n
    where aux     = [[]] : [[1]] : zipWith f (tail aux) aux
          f xs ys = map (1:) xs ++ map (2:) ys

-- Comparación de las definiciones de sumas
--    ghci> length (sumas 25)
--    121393
--    (1.84 secs, 378307888 bytes)
--    ghci> length (sumas 26)
--    196418
--    (3.09 secs, 623707712 bytes)
--    ghci> length (sumas2 25)
--    121393
--    (0.11 secs, 39984864 bytes)
--    ghci> length (sumas2 26)
--    196418
--    (0.17 secs, 63880032 bytes)

-- La segunda definición es más eficiente y es la que usaremos en lo
-- sucesivo:
sumas :: Int -> [[Int]]
sumas = sumas2

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    nSumas :: Int -> Integer
-- tal que (nSumas n) es el número de descomposiciones de n como sumas
-- cuyos sumandos son 1 ó 2. Por ejemplo,
--    nSumas 4  ==  5
--    nSumas 7  ==  21
-- ---------------------------------------------------------------------

-- 1ª definición
nSumas1 :: Int -> Integer
nSumas1 = genericLength . sumas2

-- 2ª definición
nSumas2 :: Int -> Integer
```

```haskell
nSumas2 0 = 1
nSumas2 1 = 1
nSumas2 n = nSumas2 (n-1) + nSumas2 (n-2)

-- 3ª definición
nSumas3 :: Int -> Integer
nSumas3 n = aux `genericIndex` n
    where aux = 1 : 1 : zipWith (+) aux (tail aux)

-- Comparación de las definiciones de nSumas
--    ghci> nSumas1 33
--    5702887
--    (4.33 secs, 1831610456 bytes)
--    ghci> nSumas2 33
--    5702887
--    (12.33 secs, 1871308192 bytes)
--    ghci> nSumas3 33
--    5702887
--    (0.01 secs, 998704 bytes)

-- Nota. El valor de (nSumas n) es el n-ésimo término de la sucesión de
-- Fibonacci 1, 1, 2, 3, 5, 8, ...
```

1.7. Examen 7 (27 de junio de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 1ª convocatoria (27 de junio de 2018)
-- ---------------------------------------------------------------------

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

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. 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]
--    reducido (take (10^7) (cycle [N,E,O,S])) ==  []
--
-- Nótese que en el penúltimo 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]
```

```haskell
-- 1ª solución (por recursión)
-- ===========================

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 (por plegado)
-- =========================

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'
```

```haskell
    where ds' = reducido3 ds

-- 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
-- =========================

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un capicúa es un número que es igual leído de izquierda
-- a derecha que de derecha a izquierda.
--
-- Definir la función
--    mayorCapicuaP :: Integer -> Integer
-- tal que (mayorCapicuaP n) es el mayor capicúa que es el producto de
-- dos números de n cifras. Por ejemplo,
--    mayorCapicuaP 2  ==  9009
--    mayorCapicuaP 3  ==  906609
--    mayorCapicuaP 4  ==  99000099
--    mayorCapicuaP 5  ==  9966006699
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

mayorCapicuaP :: Integer -> Integer
mayorCapicuaP n = head (capicuasP n)

-- (capicuasP n) es la lista de las capicúas de 2*n cifras que
-- pueden escribirse como productos de dos números de n cifras. Por
-- ejemplo, Por ejemplo,
--    λ> capicuasP 2
```

```haskell
--    [9009,8448,8118,8008,7227,7007,6776,6336,6006,5775,5445,5335,
--     5225,5115,5005,4884,4774,4664,4554,4224,4004,3773,3663,3003,
--     2992,2772,2552,2442,2332,2112,2002,1881,1771,1551,1221,1001]
capicuasP n = [x | x <- capicuas n,
                   not (null (productosDosNumerosCifras n x))]

-- (capicuas n) es la lista de las capicúas de 2*n cifras de mayor a
-- menor. Por ejemplo,
--    capicuas 1           ==  [99,88,77,66,55,44,33,22,11]
--    take 7 (capicuas 2)  ==  [9999,9889,9779,9669,9559,9449,9339]
capicuas :: Integer -> [Integer]
capicuas n = [capicua x | x <- numerosCifras n]

-- (numerosCifras n) es la lista de los números de n cifras de mayor a
-- menor. Por ejemplo,
--    numerosCifras 1           ==  [9,8,7,6,5,4,3,2,1]
--    take 7 (numerosCifras 2)  ==  [99,98,97,96,95,94,93]
--    take 7 (numerosCifras 3)  ==  [999,998,997,996,995,994,993]
numerosCifras :: Integer -> [Integer]
numerosCifras n = [a,a-1..b]
  where a = 10^n-1
        b = 10^(n-1)

-- (capicua n) es la capicúa formada añadiendo el inverso de n a
--  continuación de n. Por ejemplo,
--    capicua 93  ==  9339
capicua :: Integer -> Integer
capicua n = read (xs ++ reverse xs)
  where xs = show n

-- (productosDosNumerosCifras n x) es la lista de los números y de n
-- cifras tales que existe un z de n cifras y x es el producto de y por
-- z. Por ejemplo,
--    productosDosNumerosCifras 2 9009  ==  [99,91]
productosDosNumerosCifras n x = [y | y <- numeros,
                                     mod x y == 0,
                                     div x y `elem` numeros]
  where numeros = numerosCifras n

-- 2ª solución
```

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

mayorCapicuaP2 :: Integer -> Integer
mayorCapicuaP2 n = maximum [x*y | x <- [a,a-1..b],
                                  y <- [a,a-1..b],
                                  esCapicua (x*y)]
  where a = 10^n-1
        b = 10^(n-1)

-- (esCapicua x) se verifica si x es capicúa. Por ejemplo,
--    esCapicua 353  ==  True
--    esCapicua 357  ==  False
esCapicua :: Integer -> Bool
esCapicua n = xs == reverse xs
  where xs = show n

-- 3ª solución
-- ===========

mayorCapicuaP3 :: Integer -> Integer
mayorCapicuaP3 n = maximum [x*y | (x,y) <- pares a b,
                                  esCapicua (x*y)]
  where a = 10^n-1
        b = 10^(n-1)

-- (pares a b) es la lista de los pares de números entre a y b de forma
-- que su suma es decreciente. Por ejemplo,
--    pares 9 7  ==  [(9,9),(8,9),(8,8),(7,9),(7,8),(7,7)]
pares a b = [(x,z-x) | z <- [a1,a1-1..b1],
                       x <- [a,a-1..b],
                       x <= z-x, z-x <= a]
  where a1 = 2*a
        b1 = 2*b

-- 4ª solución
-- ===========

mayorCapicuaP4 :: Integer -> Integer
mayorCapicuaP4 n = maximum [x | y <- [a..b],
                                z <- [y..b],
```

```
                                  let x = y * z,
                                  let s = show x,
                                  s == reverse s]
  where a = 10^(n-1)
        b = 10^n-1

-- 5ª solución
-- ===========

mayorCapicuaP5 :: Integer -> Integer
mayorCapicuaP5 n = maximum [x*y | (x,y) <- pares2 b a, esCapicua (x*y)]
  where a = 10^(n-1)
        b = 10^n-1

-- (pares2 a b) es la lista de los pares de números entre a y b de forma
-- que su suma es decreciente. Por ejemplo,
--    pares2 9 7  ==  [(9,9),(8,9),(8,8),(7,9),(7,8),(7,7)]
pares2 a b = [(x,y) | x <- [a,a-1..b], y <- [a,a-1..x]]

-- 6ª solución
-- ===========

mayorCapicuaP6 :: Integer -> Integer
mayorCapicuaP6 n = maximum [x*y | x <- [a..b],
                                  y <- [x..b] ,
                                  esCapicua (x*y)]
  where a = 10^(n-1)
        b = 10^n-1

-- (cifras n) es la lista de las cifras de n en orden inverso. Por
-- ejemplo,
--    cifras 325  == [5,2,3]
cifras :: Integer -> [Integer]
cifras n
  | n < 10    = [n]
  | otherwise = ultima n : cifras (quitarUltima n)

-- (ultima n) es la última cifra de n. Por ejemplo,
--    ultima 325  ==  5
ultima  :: Integer -> Integer
```

```haskell
ultima n =  n - (n `div` 10)*10

-- (quitarUltima n) es el número obtenido al quitarle a n su última
-- cifra. Por ejemplo,
--    quitarUltima 325  =>  32
quitarUltima :: Integer -> Integer
quitarUltima n = (n - ultima n) `div` 10

-- 7ª solución
-- ===========

mayorCapicuaP7 :: Integer -> Integer
mayorCapicuaP7 n = head [x | x <- capicuas n, esFactorizable x n]

-- (esFactorizable x n) se verifica si x se puede escribir como producto
-- de dos números de n dígitos. Por ejemplo,
--    esFactorizable 1219 2  ==  True
--    esFactorizable 1217 2  ==  False
esFactorizable x n = aux i x
  where b = 10^n-1
        i = floor (sqrt (fromIntegral x))
        aux i x | i > b          = False
                | x `mod` i == 0 = x `div` i < b
                | otherwise      = aux (i+1) x

-- Comparación de soluciones                                          --
-- =========================

-- El tiempo de cálculo de (mayorCapicuaP n) para las 6 definiciones es
--    +------+------+------+------+
--    | Def. | 2    | 3    | 4    |
--    |------+------+------+------|
--    |    1 | 0.01 | 0.13 | 1.39 |
--    |    2 | 0.03 | 2.07 |      |
--    |    3 | 0.05 | 3.86 |      |
--    |    4 | 0.01 | 0.89 |      |
--    |    5 | 0.03 | 1.23 |      |
--    |    6 | 0.02 | 1.03 |      |
--    |    7 | 0.01 | 0.02 | 0.02 |
--    +------+------+------+------+
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol a = H a
--                 | Nd a (Arbol a) (Arbol a)
--      deriving (Eq, Show)
--
-- Por ejemplo, el árbol
--          1
--        /   \
--       2     3
--      / \   / \
--     4   5 6   7
--        / \
--       8   9
-- se pueden representar por
--    ejArbol :: Arbol Int
--    ejArbol = Nd 1 (Nd 2 (H 4)
--                         (Nd 5 (H 8) (H 9)))
--                   (Nd 3 (H 6) (H 7))
--
-- Definir la función
--    recorrido :: Arbol t -> [t]
-- tal que (recorrido a) es el recorrido del árbol a por niveles desde
-- la raíz a las hojas y de izquierda a derecha. Por ejemplo,
--    λ> recorrido (Nd 1 (Nd 2 (H 4) (Nd 5 (H 8) (H 9))) (Nd 3 (H 6) (H 7)))
--    [1,2,3,4,5,6,7,8,9]
--    λ> recorrido (Nd 1 (Nd 3 (H 6) (H 7)) (Nd 2 (H 4) (Nd 5 (H 8) (H 9))))
--    [1,3,2,6,7,4,5,8,9]
--    λ> recorrido (Nd 1 (Nd 3 (H 6) (H 7)) (Nd 2 (H 4) (H 5)))
--    [1,3,2,6,7,4,5]
--    λ> recorrido (Nd 1 (Nd 2 (H 4) (H 5)) (Nd 3 (H 6) (H 7)))
--    [1,2,3,4,5,6,7]
--    λ> recorrido (Nd 1 (Nd 2 (H 4) (H 5)) (H 3))
--    [1,2,3,4,5]
--    λ> recorrido (Nd 1 (H 4) (H 3))
--    [1,4,3]
--    λ> recorrido (H 3)
--    [3]
```

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

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

ejArbol :: Arbol Int
ejArbol = Nd 1 (Nd 2 (H 4)
                     (Nd 5 (H 8) (H 9)))
               (Nd 3 (H 6) (H 7))

-- 1ª definición
-- =============

recorrido :: Arbol t -> [t]
recorrido = concat . niveles

-- (niveles a) es la lista de los niveles del árbol a. Por ejemplo,
--    λ> niveles (Nd 1 (Nd 2 (H 4) (Nd 5 (H 8) (H 9))) (Nd 3 (H 6) (H 7)))
--    [[1],[2,3],[4,5,6,7],[8,9]]
niveles :: Arbol t -> [[t]]
niveles (H x)      = [[x]]
niveles (Nd x i d) = [x] : mezcla2 (niveles i) (niveles d)

-- (mezcla2 xss yss) es la lista obtenida concatenando los
-- correspondientes elementos de xss e yss. Por ejemplo,
--    λ> mezcla2 [[1,3],[2,5,7]] [[4],[1,9],[0,14]]
--    [[1,3,4],[2,5,7,1,9],[0,14]]
--    λ> mezcla2 [[1,3],[2,5,7]] [[4]]
--    [[1,3,4],[2,5,7]]
mezcla2 :: [[a]] -> [[a]] -> [[a]]
mezcla2 [] yss           = yss
mezcla2 xss []           = xss
mezcla2 (xs:xss) (ys:yss) = (xs ++ ys) : mezcla2 xss yss

-- 2ª definición
-- =============

recorrido2 :: Arbol t -> [t]
recorrido2 = concat . niveles2
```

```haskell
-- (niveles2 a) es la lista de los niveles del árbol a. Por ejemplo,
--    λ> niveles2 (Nd 1 (Nd 2 (H 4) (Nd 5 (H 8) (H 9))) (Nd 3 (H 6) (H 7)))
--    [[1],[2,3],[4,5,6,7],[8,9]]
niveles2 :: Arbol t -> [[t]]
niveles2 a = takeWhile (not . null) [nivel n a | n <- [0..]]

-- (nivel n a) es el nivel iésimo del árbol a. Por ejemplo,
--    λ> nivel 2 (Nd 1 (Nd 2 (H 4) (Nd 5 (H 8) (H 9))) (Nd 3 (H 6) (H 7)))
--    [4,5,6,7]
--    λ> nivel 5 (Nd 1 (Nd 2 (H 4) (Nd 5 (H 8) (H 9))) (Nd 3 (H 6) (H 7)))
--    []
nivel :: Int -> Arbol t -> [t]
nivel 0 (H x)      = [x]
nivel n (H _)      = []
nivel 0 (Nd x _ _) = [x]
nivel n (Nd x i d) = nivel (n-1) i ++ nivel (n-1) d

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    diagonalesDescendentes :: Int -> [[(Int,Int)]]
-- tal que (diagonalesDescendentes) es la lista de las posiciones de las
-- diagonales secundarias de una matriz cuadrada de orden n desde la
-- posición superior izquierda hasta la inferior derecha, recorriendo
-- las diagonales de arriba hacia abajo. Por ejemplo,
--    λ> diagonalesDescendentes 4
--    [[(1,1)],
--     [(1,2),(2,1)],
--     [(1,3),(2,2),(3,1)],
--     [(1,4),(2,3),(3,2),(4,1)],
--     [(2,4),(3,3),(4,2)],
--     [(3,4),(4,3)],
--     [(4,4)]]
---------------------------------------------------------------------

diagonalesDescendentes :: Int -> [[(Int,Int)]]
diagonalesDescendentes n =
  [[(i,m-i) | i <- [max 1 (m-n)..min n (m-1)]] | m <- [2..2*n]]

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

```haskell
-- Ejercicio 4.2. Definir la función
--    diagonalesZigZag :: Int -> [[(Int,Int)]]
-- tal que (diagonalesZigZag n) es la lista de las posiciones de las
-- diagonales secundarias de una matriz cuadrada de orden n desde la
-- posición superior izquierda hasta la inferior derecha, recorriendo
-- las diagonales en zig zag; es decir, altenativamente de arriba hacia
-- abajo y de abajo hacia arriba. Por ejemplo,
--    λ> diagonalesZigZag 4
--    [[(1,1)],
--     [(2,1),(1,2)],
--     [(1,3),(2,2),(3,1)],
--     [(4,1),(3,2),(2,3),(1,4)],
--     [(2,4),(3,3),(4,2)],
--     [(4,3),(3,4)],
--     [(4,4)]]
-- ---------------------------------------------------------------------

diagonalesZigZag :: Int -> [[(Int,Int)]]
diagonalesZigZag n =
  [f d | (f,d) <- zip (cycle [id,reverse]) (diagonalesDescendentes n)]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    numeracion :: (Int -> [[(Int,Int)]]) -> Int -> Matrix Int
-- tal que (numeracion r n) es la matriz cuadrada de orden n obtenida
-- numerando sus elementos con los números del 0 al n^2-1 según su
-- posición en el recorrido r. Por ejemplo,
--    λ> numeracion diagonalesDescendentes 4
--    (  0  1  3  6 )
--    (  2  4  7 10 )
--    (  5  8 11 13 )
--    (  9 12 14 15 )
--
--    λ> numeracion diagonalesZigZag 4
--    (  0  2  3  9 )
--    (  1  4  8 10 )
--    (  5  7 11 14 )
--    (  6 12 13 15 )
-- ---------------------------------------------------------------------
```

```haskell
numeracion :: (Int -> [[(Int,Int)]]) -> Int -> Matrix Int
numeracion r n =
  fromList n n (elems (numeracionAux r n))

numeracionAux :: (Int -> [[(Int,Int)]]) -> Int -> Array (Int,Int) Int
numeracionAux r n =
  array ((1,1),(n,n)) (zip (concat (r n)) [0..])

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Comprobar con QuickCheck que para para cualquier
-- número natural n se verifica que son iguales las diagonales
-- principales de (numeracion r n) donde r es cualquiera de los dos
-- recorridos definidos (es decir, diagonalesDescendentes y
-- diagonalesZigZag).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_numeracion :: Int -> Property
prop_numeracion n =
  n >= 0 ==>
   getDiag (numeracion diagonalesDescendentes n) ==
   getDiag (numeracion diagonalesZigZag n)

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

1.8. Examen 8 (10 de septiembre de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la 2ª convocatoria (10 de septiembre de 2018)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Matrix as M
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número de Hilbert es un entero positivo de la forma
-- 4n+1. Los primeros números de Hilbert son 1, 5, 9, 13, 17, 21, 25,
-- 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, ...
--
-- Un primo de Hilbert es un número de Hilbert n que no es divisible
-- por ningún número de Hilbert menor que n (salvo el 1). Los primeros
-- primos de Hilbert son 5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57,
-- 61, 69, 73, 77, 89, 93, 97, 101, 109, 113, 121, 129, 133, 137, ...
--
-- Definir la función
--    factorizacionesH :: Integer -> [[Integer]]
-- tal que (factorizacionesH n) es la lista de listas de primos de
-- Hilbert tales que el producto de los elementos de cada una de las
-- listas es el número de Hilbert n. Por ejemplo,
--   factorizacionesH  25  ==  [[5,5]]
--   factorizacionesH  45  ==  [[5,9]]
--   factorizacionesH 441  ==  [[9,49],[21,21]]
--
-- Comprobar con QuickCheck que todos los números de Hilbert son
-- factorizables como producto de primos de Hilbert (aunque la
-- factorización, como para el 441, puede no ser única).
-- ---------------------------------------------------------------------

factorizacionesH :: Integer -> [[Integer]]
factorizacionesH n = aux n primosH
  where aux n (x:xs)
          | x == n         = [[n]]
          | x > n          = []
          | n `mod` x == 0 = map (x:) (aux (n `div` x) (x:xs))
                             ++ aux n xs
          | otherwise      = aux n xs

-- primosH es la sucesión de primos de Hilbert. Por ejemplo,
--    take 15 primosH  ==  [5,9,13,17,21,29,33,37,41,49,53,57,61,69,73]
primosH :: [Integer]
primosH = filter esPrimoH (tail numerosH)
```

```haskell
  where esPrimoH n = all noDivideAn [5,9..m]
        where noDivideAn x = n `mod` x /= 0
              m            = ceiling (sqrt (fromIntegral n))

-- numerosH es la sucesión de los números de Hilbert. Por ejemplo,
--    take 15 numerosH  ==  [1,5,9,13,17,21,25,29,33,37,41,45,49,53,57]
numerosH :: [Integer]
numerosH = [1,5..]

-- La propiedad es
prop_factorizable :: Integer -> Property
prop_factorizable n =
  n > 0 ==> not (null (factorizacionesH (1 + 4 * n)))

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se llaman potencias de Fermi-Dirac los números de la
-- forma p^(2^k), donde p es un número primo y k es un número natural.
--
-- Definir la sucesión
--    potencias :: [Integer]
-- cuyos términos sean las potencias de Fermi-Dirac ordenadas de menor a
-- mayor. Por ejemplo,
--    take 14 potencias    ==  [2,3,4,5,7,9,11,13,16,17,19,23,25,29]
--    potencias !! 60      ==  241
--    potencias !! (10^6)  ==  15476303
-- ---------------------------------------------------------------------

potencias :: [Integer]
potencias = 2 : mezcla (tail primes) (map (^2) potencias)

-- (mezcla xs ys) es la lista obtenida mezclando las dos listas xs e ys,
-- que se suponen ordenadas y disjuntas. Por ejemplo,
--    ghci> take 15 (mezcla [2^n | n <- [1..]] [3^n | n <- [1..]])
--    [2,3,4,8,9,16,27,32,64,81,128,243,256,512,729]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla (x:xs) (y:ys) | x < y = x : mezcla xs (y:ys)
```

```
                    | x > y = y : mezcla (x:xs) ys

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Eq, Show)
--
-- Por ejemplo, el árbol
--          10
--         /  \
--        /    \
--       8      1
--      / \    / \
--     3   9  2   6
-- se pueden representar por
--    ejArbol :: Arbol Int
--    ejArbol = N 10 (N 8 (H 3) (H 9))
--                   (N 1 (H 2) (H 6))
--
-- La distancia entre un padre y un hijo en el árbol es el valor
-- absoluto de la diferencia de sus valores. Por ejemplo, la distancia
-- de 10 a 8 es 2 y de 1 a 6 es 5.
--
-- Definir la función
--    maximaDistancia :: (Num a, Ord a) => Arbol a -> a
-- tal que (maximaDistancia a) es la máxima distancia entre un padre y
-- un hijo del árbol a. Por ejemplo,
--    maximaDistancia ejArbol                                    ==  9
--    maximaDistancia (N 1 (N 8 (H 3) (H 9)) (N 1  (H 2) (H 6)))  ==  7
--    maximaDistancia (N 8 (N 8 (H 3) (H 9)) (N 10 (H 2) (H 6)))  ==  8
-- ---------------------------------------------------------------------

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

ejArbol :: Arbol Int
ejArbol = N 10 (N 8 (H 3) (H 9))
```

```
              (N 1 (H 2) (H 6))

maximaDistancia :: (Num a, Ord a) => Arbol a -> a
maximaDistancia (H _)     = 0
maximaDistancia (N x i d) = maximum [abs (x - raiz i)
                                    , maximaDistancia i
                                    , abs (x - raiz d)
                                    , maximaDistancia d]

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    mayor :: (Num a, Ord a) => Int -> Matrix a -> [[a]]
-- tal que (mayor n p) es la lista de los segmentos formados por n
-- elementos adyacentes en la misma fila, columna, diagonal principal o
-- diagonal secundaria de la matriz p cuyo productos son máximo. Por
-- ejemplo,
--    mayor 3 (fromList 3 4 [1..12])                     == [[10,11,12]]
--    mayor 3 (fromLists [[1,3,4,5],[0,7,2,1],[3,9,2,1]]) == [[3,7,9]]
--    mayor 2 (fromLists [[1,3,4],[0,3,2]])              == [[3,4],[4,3]]
--    mayor 2 (fromLists [[1,2,1],[3,0,3]])              == [[2,3],[2,3]]
--    mayor 2 (fromLists [[1,2,1],[3,4,3]])              == [[3,4],[4,3]]
--    mayor 2 (fromLists [[1,5,1],[3,4,3]])              == [[5,4]]
--    mayor 3 (fromLists [[1,3,4,5],[0,7,2,1],[3,9,2,1]]) == [[3,7,9]]
-- ---------------------------------------------------------------------

mayor :: (Num a, Ord a) => Int -> Matrix a -> [[a]]
mayor n p =
  [xs | xs <- segmentos, product xs == m]
  where segmentos = adyacentes n p
        m         = maximum (map product segmentos)

-- (adyacentes n p) es la lista de los segmentos de longitud n de las
-- líneas de la matriz p. Por ejemplo,
--    ghci> adyacentes 3 (fromList 3 4 [1..12])
--    [[1,2,3],[2,3,4],[5,6,7],[6,7,8],[9,10,11],[10,11,12],[1,5,9],
--     [2,6,10],[3,7,11],[4,8,12],[1,6,11],[2,7,12],[3,6,9],[4,7,10]]
```

```haskell
adyacentes :: Num a => Int -> Matrix a -> [[a]]
adyacentes n p = concatMap (segmentos n) (lineas p)

-- (lineas p) es la lista de las líneas de la matriz p. Por ejemplo,
--    ghci> lineas (fromList 3 4 [1..12])
--    [[1,2,3,4],[5,6,7,8],[9,10,11,12],
--     [1,5,9],[2,6,10],[3,7,11],[4,8,12],
--     [9],[5,10],[1,6,11],[2,7,12],[3,8],[4],
--     [1],[2,5],[3,6,9],[4,7,10],[8,11],[12]]
lineas :: Num a => Matrix a -> [[a]]
lineas p = filas p ++
           columnas p ++
           diagonalesPrincipales p ++
           diagonalesSecundarias p

-- (filas p) es la lista de las filas de la matriz p. Por ejemplo,
--    ghci> filas (fromList 3 4 [1..12])
--    [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
filas :: Num a => Matrix a -> [[a]]
filas = toLists

-- (columnas p) es la lista de las columnas de la matriz p. Por ejemplo,
--    ghci> columnas (fromList 3 4 [1..12])
--    [[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
columnas :: Num a => Matrix a -> [[a]]
columnas = toLists . M.transpose

-- (diagonalesPrincipales p) es la lista de las diagonales principales
-- de p. Por ejemplo,
--    ghci> diagonalesPrincipales (fromList 3 4 [1..12])
--    [[9],[5,10],[1,6,11],[2,7,12],[3,8],[4]]
diagonalesPrincipales :: Matrix a -> [[a]]
diagonalesPrincipales p =
  [[p!ij1 | ij1 <- extension ij] | ij <- iniciales]
  where m = nrows p
        n = ncols p
        iniciales = [(i,1) | i <- [m,m-1..2]] ++ [(1,j) | j <- [1..n]]
        extension (i,j) = [(i+k,j+k) | k <- [0..min (m-i) (n-j)]]

-- (diagonalesSecundarias p) es la lista de las diagonales secundarias
```

```haskell
-- de p. Por ejemplo,
--    ghci> diagonalesSecundarias (fromList 3 4 [1..12])
--    [[1],[2,5],[3,6,9],[4,7,10],[8,11],[12]]
diagonalesSecundarias p =
  [[p!ij1 | ij1 <- extension ij] | ij <- iniciales]
  where m = nrows p
        n = ncols p
        iniciales = [(1,j) | j <- [1..n]] ++ [(i,n) | i <- [2..m]]
        extension (i,j) = [(i+k,j-k) | k <- [0..min (j-1) (m-i)]]

-- (segmentos n xs) es la lista de los segmentos de longitud n de la
-- lista xs. Por ejemplo,
--    segmentos 2 [3,5,4,6]  ==  [[3,5],[5,4],[4,6]]
segmentos :: Int -> [a] -> [[a]]
segmentos n xs = take (length xs - n + 1) (map (take n) (tails xs))

-- Se puede definir por recursión
segmentos2 :: Int -> [a] -> [[a]]
segmentos2 n xs
  | length xs < n = []
  | otherwise     = take n xs : segmentos2 n (tail xs)
```

2

Exámenes del grupo 2

Antonia M. Chávez

2.1. Examen 1 (30 de octubre de 2017)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 1º examen de evaluación continua (30 de octubre de 2017)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número es defectivo si es mayor que la suma de sus
-- divisores propios positivos. Por ejemplo, 15 tiene 3 divisores
-- propios (1, 3 y 5) cuya suma es 9 que es menor que 15. Por tanto, el
-- número 15 es defectivo.
--
-- Definir la función
--    esDefectivo :: Int -> Bool
-- tal que (esDefectivo x) se verifica si x es defectivo. Por ejemplo,
--    esDefectivo 15 == True
--    esDefectivo 17 == True
--    esDefectivo 18 == False
-- ---------------------------------------------------------------------
```

```haskell
esDefectivo :: Int -> Bool
esDefectivo x = x > sum (divisoresPropios x)

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    defectivosMenores :: Int -> [Int]
-- tal que (defectivosMenores n) es la lista de los números defectivos
-- menores que n. Por ejemplo,
--    defectivosMenores 20 == [1,2,3,4,5,7,8,9,10,11,13,14,15,16,17,19]
-- ---------------------------------------------------------------------

-- 1ª definición
defectivosMenores :: Int -> [Int]
defectivosMenores n = [x | x <- [1 ..n-1], esDefectivo x]

-- 2ª definición
defectivosMenores2 :: Int -> [Int]
defectivosMenores2 n = filter esDefectivo [1..n-1]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    defectivosConCifra :: Int -> Int -> [Int]
-- tal que (defectivosConCifra n x) es la lista de los n primeros
-- números defectivos que contienen la cifra x. Por ejemplo,
--    defectivosConCifra 10 3 == [3,13,23,31,32,33,34,35,37,38]
--    defectivosConCifra 10 0 == [10,50,101,103,105,106,107,109,110,130]
-- ---------------------------------------------------------------------

defectivosConCifra :: Int -> Int -> [Int]
defectivosConCifra n x =
  take n [y | y <- [1..]
            , esDefectivo y
            , c `elem` show y]
  where c = head (show x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Comprobar con QuickCheck que los números primos son
```

```haskell
-- defectivos.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_primosSonDefectivos :: Int  -> Property
prop_primosSonDefectivos x =
  esPrimo x ==> esDefectivo x

esPrimo :: Int -> Bool
esPrimo x = divisoresPropios x == [1]

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por comprension, la función
--    lugarPar :: [a] -> [a]
-- tal que (lugarPar xs) es la lista de los elementos de xs que ocupan
-- las posiciones pares. Por ejemplo,
--    lugarPar [0,1,2,3,4] == [0,2,4]
--    lugarPar "ahora si"  == "aoas"
-- ---------------------------------------------------------------------

-- 1ª definición
lugarPar :: [a] -> [a]
lugarPar xs =
  [xs!!n | n <-[0,2 .. length xs - 1]]

-- 2ª definición
lugarPar2 :: [a] -> [a]
lugarPar2 xs =
  [x | (x,n) <- zip xs [0..], even n]

-- Comparación de eficiencia
--    ghci> maximum (lugarPar [1..10^5])
--    99999
--    (5.80 secs, 22,651,424 bytes)
--    ghci> maximum (lugarPar2 [1..10^5])
--    99999
```

```
--    (0.06 secs, 48,253,456 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    paresOimpares :: [a] -> [a]
-- tal que (paresOimpares xs) es la lista de los elementos que ocupan
-- las posiciones pares en xs, si la longitud de xs es par y, en caso
-- contrario, es la lista de los elementos que ocupan  posiciones
-- impares en xs. Por ejemplo,
--    paresOimpares [1,2,3,4,5]        == [2,4]
--    paresOimpares [1,2,3,4]          == [1,3]
--    paresOimpares "zorro y la loba"  == "or  alb"
--    paresOimpares "zorro y la lob"   == "zroyl o"
-- ---------------------------------------------------------------------

paresOimpares :: [a] -> [a]
paresOimpares xs
  | even (length xs) = lugarPar2 xs
  | otherwise        = lugarImpar xs

lugarImpar :: [a] -> [a]
lugarImpar xs =
  [x | (x,n) <- zip xs [0..], odd n]
```

2.2. Examen 2 (27 de noviembre de 2017)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 2º examen de evaluación continua (27 de noviembre de 2017)
-- ---------------------------------------------------------------------

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

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número es un escalón en una lista si todos los que le
-- siguen en la lista son mayores que él. Por ejemplo, 3 es un escalón
```

```
-- en la lista [4,3,6,5,8] porque 6, 5 y 8 son mayores, pero 4 no es un
-- escalón ya que el 3 le sigue y no es mayor que 4.
--
-- Definir la función
--    escalones :: [Int] -> [Int]
-- tal que (escalones xs) es la lista de los escalones de la lista
-- xs. Por ejemplo,
--    escalones [5,3,6,5,8] == [3,5,8]
--    escalones [4,6,0]     == [0]
-- ---------------------------------------------------------------------

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

escalones1 :: [Int] -> [Int]
escalones1 xs =
  [y | (y:ys) <- init (tails xs)
     , all (>y) ys]

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

escalones2 :: [Int] -> [Int]
escalones2 [] = []
escalones2 (x:xs) | all (>x) xs = x : escalones2 xs
                  | otherwise   = escalones2 xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumaNum :: [Int] -> [Int] -> Int
-- tal (sumaNum xs ns) es la suma de los elementos de la lista xs que
-- ocupan las posiciones que se indican en la lista de números no
-- negativos ns. Por ejemplo,
-- + sumaNum [1,2,4,3,6] [0,1,4] es 1+2+6, ya que 1, 2 y 6 son los
--   números de la lista que están en las posiciones 0,1 y 4.
-- + sumaNum [1,2,5,3,9] [4,6,2] es 9+0+5 ya que el 9 y el 5 están en
--   las posiciones 4 y 2 pero en la posicion 6 no aparece ningún
--   elemento.
-- Ejemplos:
--    sumaNum [1,2,4,3,6] [2,4,6]  ==  10
```

```haskell
--    sumaNum [1,2,4,3,6] [0,1,4]  ==  9
--    sumaNum [1,2,5,3,9] [4,6,2]  ==  14
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
sumaNum1 :: [Int] -> [Int] -> Int
sumaNum1 xs ns = sum [xs!!n | n <- ns, n < length xs]

-- 2ª definición (por recursión)
sumaNum2 :: [Int] -> [Int] -> Int
sumaNum2 _ [] = 0
sumaNum2 xs (n:ns) | n < length xs = xs!!n + sumaNum2 xs ns
                   | otherwise     = sumaNum2 xs ns

-- 3ª definición (por recursión final)
sumaNum3  :: [Int] -> [Int] -> Int
sumaNum3 xs ns = aux ns 0
    where aux [] ac = ac
          aux (y:ys) ac | y < length xs = aux ys (ac + (xs!!y))
                        | otherwise     = aux ys ac

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideremos la sucesión construida a partir de un
-- número n, sumando los factoriales de los dígitos de n, y repitiendo
-- sobre el resultado dicha operación. Por ejemplo, si comenzamos en 69,
-- obtendremos:
--    69
-- 363600  (porque 6! + 9! = 363600)
--   1454  (porque 3! + 6! + 3! + 6! + 0! + 0! = 1454)
--    169  (porque 1! + 4! + 5! + 4! = 169)
-- 363601  (porque 1! + 6! + 9! = 363601)
--   1454  (porque 3! + 6! + 3! + 6! + 0! + 1! = 1454)
-- ......
--
-- La cadena correspondiente a un número n son los términos de la
-- sucesión que empieza en n hasta la primera repetición de un elemento
-- en la sucesión. Por ejemplo, la cadena de 69 es
--   [69,363600,1454,169,363601]
-- ya que el siguiente número sería 1454, que ya está en la lista.
--
```

```haskell
-- Definir la función cadena
--   cadena  :: Int -> [Int]
-- tal que (cadena n) es la cadena correspondiente al número n. Por
-- ejemplo,
--
-- cadena 69    ==  [69,363600,1454,169,363601]
-- cadena 145   ==  [145]
-- cadena 78    ==  [78,45360,871,45361]
-- cadena 569   ==  [569,363720,5775,10320,11,2]
-- cadena 3888  ==  [3888,120966,364324,782,45362,872]
-- length (cadena 1479) ==  60
-- ---------------------------------------------------------------------

cadena :: Int -> [Int]
cadena x = aux x [x]
  where aux y ac | elem (f y) ac = ac
                 | otherwise     = aux (f y) (ac ++[f y])
        f x       = sum [fac x | x <- digitos x]
        digitos x = [read [y] | y <- show x]
        fac n     = product [1..n]
```

2.3. Examen 3 (30 de enero de 2018)

El examen es común con el del grupo 4 (ver página 121).

2.4. Examen 4 (12 de marzo de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 4º examen de evaluación continua (12 de marzo de 2018)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Array
-- import Data.Matrix as M
-- import Data.Vector
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Hay 3 números (el 2, 3 y 4) cuyos factoriales son
-- divisibles por 2 pero no por 5. Análogamente, hay números 5 (el 5, 6,
-- 7, 8, 9) cuyos factoriales son divisibles por 15 pero no por 25.
--
-- Definir la función
--    nNumerosConFactorialesDivisibles :: Integer -> Integer -> Integer
-- tal que (nNumerosConFactorialesDivisibles x y) es la cantidad de
-- números cuyo factorial es divisible por x pero no  por y. Por ejemplo,
--   nNumerosConFactorialesDivisibles 2  5      ==  3
--   nNumerosConFactorialesDivisibles 15 25     ==  5
--   nNumerosConFactorialesDivisibles 100 2000  ==  5
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

nNumerosConFactorialesDivisibles :: Integer -> Integer -> Integer
nNumerosConFactorialesDivisibles x y =
  genericLength (numerosConFactorialesDivisibles x y)

-- (numerosConFactorialesDivisibles x y) es la lista de números
-- divisibles por el factorial de x pero no divisibles por el
-- factorial de y. Por ejemplo,
--   numerosConFactorialesDivisibles 2  5   ==  [2,3,4]
--   numerosConFactorialesDivisibles 15 25  ==  [5,6,7,8,9]
numerosConFactorialesDivisibles :: Integer -> Integer -> [Integer]
numerosConFactorialesDivisibles x y =
  [z | z <- [0..y-1]
     , factorial z `mod` x == 0
     , factorial z `mod` y /= 0]

-- (factorial n) es el factorial de n. Por ejemplo,
--   factorial 4  ==  24
factorial :: Integer -> Integer
factorial n = product [1..n]

-- 2ª solución (usando la función de Smarandache)
-- ==============================================
```

```haskell
nNumerosConFactorialesDivisibles2 :: Integer -> Integer -> Integer
nNumerosConFactorialesDivisibles2 x y =
  max 0 (smarandache y - smarandache x)

--(smarandache n) es el menor número cuyo factorial es divisible por
-- n. Por ejemplo,
--    smarandache 8   ==  4
--    smarandache 10  ==  5
--    smarandache 16  ==  6
smarandache :: Integer -> Integer
smarandache x =
  head [n | (n,y) <- zip [0..] factoriales
          , y `mod` x == 0]

-- factoriales es la lista de los factoriales. Por ejemplo,
--    λ> take 12 factoriales
--    [1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800]
factoriales :: [Integer]
factoriales = 1 : scanl1 (*) [1..]

-- Comparación de ediciencia
--    λ> nNumerosConFactorialesDivisibles 100 2000
--    5
--    (2.70 secs, 3,933,938,648 bytes)
--    λ> nNumerosConFactorialesDivisibles2 100 2000
--    5
--    (0.01 secs, 148,200 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una matriz centro simétrica es una matriz cuadrada que
-- es simétrica respecto de su centro. Por ejemplo, de las siguientes
-- matrices, las dos primeras son simétricas y las otras no lo son
--    (1 2)   (1 2 3)   (1 2 3)   (1 2 3)
--    (2 1)   (4 5 4)   (4 5 4)   (4 5 4)
--            (3 2 1)   (3 2 2)
--
-- Definir la función
--    esCentroSimetrica :: Eq t => Array (Int,Int) t -> Bool
-- tal que (esCentroSimetrica a) se verifica si la matriz a es centro
```

```
-- simétrica. Por ejemplo,
--    λ> esCentroSimetrica (listArray ((1,1),(2,2)) [1,2, 2,1])
--    True
--    λ> esCentroSimetrica (listArray ((1,1),(3,3)) [1,2,3, 4,5,4, 3,2,1])
--    True
--    λ> esCentroSimetrica (listArray ((1,1),(3,3)) [1,2,3, 4,5,4, 3,2,2])
--    False
--    λ> esCentroSimetrica (listArray ((1,1),(2,3)) [1,2,3, 4,5,4])
--    False
-- ---------------------------------------------------------------------

esCentroSimetrica :: Eq t => Array (Int,Int) t -> Bool
esCentroSimetrica a =
  n == m && and [a!(i,j) == a!(n-i+1,n-j+1) | i <- [1..n], j <- [i..n]]
  where (_,(n,m)) = bounds a

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios con valores en las hojas y en los
-- nodos se definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Eq, Show)
--
-- Por ejemplo, el árbol
--          10
--         /  \
--        /    \
--       8      2
--      / \    / \
--     3   5  2   0
-- se pueden representar por
--    ejArbol :: Arbol Int
--    ejArbol = N 10 (N 8 (H 3) (H 5))
--                   (N 2 (H 2) (H 0))
--
-- Un árbol cumple la propiedad de la suma si el valor de cada nodo es
-- igual a la suma de los valores de sus hijos. Por ejemplo, el árbol
-- anterior cumple la propiedad de la suma.
--
-- Definir la función
```

```
--    propSuma :: Arbol Int -> Bool
-- tal que (propSuma a) se verifica si el árbol a cumple la propiedad de
-- la suma. Por ejemplo,
--    λ> propSumaG (NG 10 [NG 8 [NG 3 [], NG 5 [] ], NG 2 [NG 2 [], NG 0 []]])
--    True
--    λ> propSumaG (NG 10 [NG 8 [NG 4 [], NG 5 [] ], NG 2 [NG 2 [], NG 0 []]])
--    False
--    λ> propSumaG (NG 10 [NG 8 [NG 3 [], NG 5 [] ], NG 2 [NG 2 [], NG 1 []]])
--    False
-- ---------------------------------------------------------------------

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

propSuma :: Arbol Int -> Bool
propSuma (H _)     = True
propSuma (N x i d) = x == raiz i + raiz d && propSuma i && propSuma d

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles generales se pueden definir mediante el
-- siguiente tipo de datos
--    data ArbolG a = NG a [ArbolG a] deriving Show
-- Por ejemplo, el árbol
--          10
--         /  \
--        /    \
--       8      2
--      / \    / \
--     3   5  2   0
-- se puede representar por
--    ejArbolG :: ArbolG Int
--    ejArbolG = NG 10 [NG 8 [NG 3 [], NG 5 [] ], NG 2 [NG 2 [], NG 0 []]]
--
-- Definir la función
--    propSumaG :: ArbolG Int -> Bool
```

```haskell
-- tal que (propSumaG a) se verifica si el árbol general a cumple la
-- propiedad de la suma. Por ejemplo,
--    λ> propSuma (N 10 (N 8 (H 3) (H 5)) (N 2 (H 2) (H 0)))
--    True
--    λ> propSuma (N 10 (N 8 (H 4) (H 5)) (N 2 (H 2) (H 0)))
--    False
--    λ> propSuma (N 10 (N 8 (H 3) (H 5)) (N 2 (H 2) (H 1)))
--    False
-- ---------------------------------------------------------------------

data ArbolG a = NG a [ArbolG a] deriving Show

propSumaG :: ArbolG Int -> Bool
propSumaG (NG _ []) = True
propSumaG (NG x as) = x == sum [raizG a | a <- as]
                      && all propSumaG as

raizG :: ArbolG a -> a
raizG (NG x _) = x
```

2.5. Examen 5 (30 de abril de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas) Grupo 2
-- 5º examen de evaluación continua (30 de abril de 2018)
-- ---------------------------------------------------------------------

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

import Data.List
import I1M.Grafo
import I1M.Pila
import I1M.PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1. El número 545 es a la vez capicúa y suma de dos
-- cuadrados consecutivos: 545 = 16^2 + 17^2
--
-- Definir la lista
```

```
--    sucesion :: [Integer]
-- cuyos elementos son los números que son suma de cuadrados
-- consecutivos y capicúas. Por ejemplo,
--    λ> take 10 sucesion
--    [1,5,181,313,545,1690961,3162613,3187813,5258525,5824285]
-- ----------------------------------------------------------------------

sucesion :: [Integer]
sucesion = filter capicua sucSumaCuadConsec

-- sucSumaCuadConsec es la sucesión de los números que son
-- suma de los cuadrados de dos números consecutivos. Por ejemplo,
--    ghci> take 10 sucSumaCuadConsec
--    [1,5,13,25,41,61,85,113,145,181]
sucSumaCuadConsec :: [Integer]
sucSumaCuadConsec =
  [x^2 + (x+1)^2 | x <- [0..]]

-- (capicua n) se verifica si n es capicúa. Por ejemplo,
--    capicua 252    ==  True
--    capicua 2552   ==  True
--    capicua 25352  ==  True
--    capicua 25342  ==  False
capicua :: Integer -> Bool
capicua n = xs == reverse xs
  where xs = show n

-- ----------------------------------------------------------------------
-- Ejercicio 2. Consideremos las pilas ordenadas según el orden
-- lexicográfico. Es decir, la pila p1 es "menor" que p2 si la
-- cima de p1 es menor que la cima de p2 y, en caso de coincidir, la
-- pila que resulta de desapilar p1 es "menor" que la pila que resulta
-- de desapilar p2.
--
-- Definir la función
--    esPilaMenor :: Ord a => Pila a -> Pila a -> Bool
-- tal que (esPilaMenor p1 p2) se verifica si p1 es "menor" que p2. Por
-- ejemplo, para la pilas
--    p1 = foldr apila vacia [1..20]
--    p2 = foldr apila vacia [1..5]
```

```haskell
--    p3 = foldr apila vacia [3..10]
--    p4 = foldr apila vacia [4,-1,7,3,8,10,0,3,3,4]
-- se verifica que
--    esPilaMenor p1 p2    == False
--    esPilaMenor p2 p1    == True
--    esPilaMenor p3 p4    == True
--    esPilaMenor vacia p1 == True
--    esPilaMenor p1 vacia == False
-- ---------------------------------------------------------------------

p1, p2, p3, p4 :: Pila Int
p1 = foldr apila vacia [1..20]
p2 = foldr apila vacia [1..5]
p3 = foldr apila vacia [3..10]
p4 = foldr apila vacia [4,-1,7,3,8,10,0,3,3,4]

esPilaMenor :: Ord a => Pila a -> Pila a -> Bool
esPilaMenor p1 p2
  | esVacia p1 = True
  | esVacia p2 = False
  | a1 < a2    = True
  | a1 > a2    = False
  | otherwise  = esPilaMenor r1 r2
  where a1 = cima p1
        a2 = cima p2
        r1 = desapila p1
        r2 = desapila p2

-- ---------------------------------------------------------------------
-- Ejercicio 3. Dados dos polinomios P y Q, la suma en grados de P y Q
-- es el polinomio que resulta de conservar los términos de ambos
-- polinomios que coinciden en grado pero sumando sus coeficientes y
-- eliminar el resto. Por ejemplo, dados los polinomios
--    pol1  =  4*x^4 + 5*x^3 + 1
--    pol2  =  6*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
-- la suma en grados de pol1 y pol2 será: 9*x^4 + 9*x^3 + 2
--
-- Definir la función
--    sumaEnGrados :: Polinomio Int -> Polinomio Int -> Polinomio Int
-- tal que (sumaEnGrados p q) es la suma en grados de los polinomios
```

```haskell
-- p y q. Por ejemplo, dados los polinomios
--    pol1  =  4*x^4 + 5*x^3 + 1
--    pol2  =  6*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
--    pol3  =  -3*x^7 + 3*x^6 + -2*x^4 + 2*x^3 + -1*x + 1
--    pol4  =  -1*x^6 + 3*x^4 + -3*x^2
-- se tendrá:
--    sumaEnGrados pol1 pol1  =>  8*x^4 + 10*x^3 + 2
--    sumaEnGrados pol1 pol2  =>  9*x^4 + 9*x^3 + 2
--    sumaEnGrados pol1 pol3  =>  2*x^4 + 7*x^3 + 2
--    sumaEnGrados pol3 pol4  =>  2*x^6 + x^4
-- ---------------------------------------------------------------------

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un clique de un grafo no dirigido G es un conjunto de
-- vértices V tal que para todo par de vértices de V, existe una arista
```

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

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

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

2.6. Examen 6 (12 de junio de 2018)

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

2.7. Examen 7 (27 de junio de 2018)

El examen es común con el del grupo 1 (ver página 58).

2.8. Examen 8 (10 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 64).

3

Exámenes del grupo 3

Francisco J. Martín

3.1. Examen 1 (2 de noviembre de 2017)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Consideremos las regiones del plano delimitadas por
-- las rectas x = 1, x = -1, y = 1 e y = -1. Diremos que dos regiones
-- son vecinas si comparten una frontera distinta de un punto; es decir.
-- no son vecinas dos regiones con un único punto de contacto.
--
-- En este ejercicio se pretende contar el número de regiones vecinas de
-- aquella en la que está un punto dado. Por ejemplo, el punto (0,0)
-- está en la region central, que tiene 4 regiones vecinas; el punto
-- (2,2) está en la región superior derecha, que tiene 2 regiones
-- vecinas; y el punto (2,0) está en la región media derecha, que tiene
-- 3 regiones vecinas. Para cualquier punto que se encuentre en las
-- rectas x = 1, x = -1, y = 1 e y = -1, el resultado debe ser 0.
--
-- Definir la función
--    numeroRegionesVecinas :: (Float,Float) -> Int
-- tal que (numeroRegionesVecinas (x,y)) es el número de regiones
-- vecinas de aquella en la que está contenido el punto (x,y). Por
-- ejemplo,
--    numeroRegionesVecinas (0,0)  ==  4
```

```haskell
--    numeroRegionesVecinas (2,2)  ==  2
--    numeroRegionesVecinas (2,0)  ==  3
--    numeroRegionesVecinas (1,0)  ==  0
-- ---------------------------------------------------------------------

numeroRegionesVecinas :: (Float,Float) -> Int
numeroRegionesVecinas (x,y)
  | x' <  1 && y' <  1 = 4
  | x' == 1 || y' == 1 = 0
  | x' >  1 && y' >  1 = 2
  | otherwise          = 3
  where x' = abs x
        y' = abs y

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una secuencia de números es de signo alternado si en
-- ella se alternan los números positivos y negativos. Se pueden dar dos
-- casos de secuencias de signo alternado:
-- + El primer término es positivo, el segundo es negativo, el tercero
--   es positivo, el cuarto es negativo, y así sucesivamente. Por
--   ejemplo, la secuencia
--      1, -1, 2, -2, 3, -3
-- + El primer término es negativo, el segundo es positivo, el tercero
--   es negativo, el cuarto es positivo, y así sucesivamente. Por
--   ejemplo, la secuencia
--      -1, 1, -2, 2, -3, 3
-- Las secuencias que tengan un 0 entre sus elementos nunca son de signo
-- alternado.
--
-- Definir la función
--    signosAlternados :: [Int] -> Bool
-- tal que (signosAlternados xs) se verifica si la secuencia de números
-- enteros xs es de signo alternado. Por ejemplo,
--    signosAlternados [1,-1,2,-2,3,-3]  ==  True
--    signosAlternados [-1,1,-2,2,-3,3]  ==  True
--    signosAlternados [0,-1,1,-1,0]     ==  False
--    signosAlternados [1,2,-1,1,-5]     ==  False
--    signosAlternados [1,-2,1,-1,-5]    ==  False
-- ---------------------------------------------------------------------
```

```haskell
signosAlternados :: [Int] -> Bool
signosAlternados xs =
  and [x*y < 0 | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. 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!
--
-- Es decir, la serie cuyo término general n-ésimo es el cociente entre
-- (n+1) yel doble del factorial de n:
--
--                    n+1
--           s(n) =  ------
--                    2*n!
--
-- Definir por comprensión la función:
--    aproximaEC :: Double -> Double
-- tal que (aproximaEC n) es la aproximación del número e calculada con
-- la serie anterior hasta el término n-ésimo. Por ejemplo,
--    aproximaEC 10  ==  2.718281663359788
--    aproximaEC 15  ==  2.718281828458612
--    aproximaEC 20  ==  2.718281828459045
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir por recursión la función:
--    aproximaER :: Double -> Double
-- tal que (aproximaER n) es la aproximación del número e calculada con
-- la serie anterior hasta el término n-ésimo. Por ejemplo,
--    aproximaER 10  ==  2.718281663359788
--    aproximaER 15  ==  2.718281828458612
--    aproximaER 20  ==  2.718281828459045
```

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

aproximaER :: Double -> Double
aproximaER 0 = 1/2
aproximaER n =
  (n+1)/(2*product [1..n]) + aproximaER (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definición por recursión la función:
--    restaCifrasDe2en2 :: Integer -> Integer
-- tal que (restaCifrasDe2en2 n) es el número obtenido a partir del
-- número n, considerando sus cifras de 2 en 2 y tomando el valor
-- absoluto de sus diferencias. Por ejemplo
--    restaCifrasDe2en2 3       ==  3
--    restaCifrasDe2en2 83      ==  5
--    restaCifrasDe2en2 283     ==  25
--    restaCifrasDe2en2 5283    ==  35
--    restaCifrasDe2en2 2538    ==  35
--    restaCifrasDe2en2 102583  ==  135
-- ---------------------------------------------------------------------

restaCifrasDe2en2 :: Integer -> Integer
restaCifrasDe2en2 n
  | n < 10    = n
  | otherwise = 10 * restaCifrasDe2en2 (n `div` 100) + abs (x-y)
  where x = n `mod` 10
        y = (n `mod` 100) `div` 10
```

3.2. Examen 2 (30 de noviembre de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 2º examen de evaluación continua (30 de noviembre de 2017)
-- =====================================================================

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

import Data.List
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Dados un número x, una relación r y una lista ys, el
-- índice de relación de x en ys con respecto a r, es el número de
-- elementos de ys tales que x está relacionado con y con respecto a
-- r. Por ejemplo, el índice de relación del número 2 en la lista
-- [1,2,3,5,1,4] con respecto a la relación < es 3, pues en la lista hay
-- 3 elementos (el 3, el 5 y el 4) tales que 2 < 3, 2 < 5 y 2 < 4.
--
-- Definir la función
--    listaIndicesRelacion :: [Int] -> (Int -> Int -> Bool) -> [Int]
--                            -> [Int]
-- tal que (listaIndicesRelacion xs r ys) es la lista de los índices de
-- relación de los elementos de la lista xs en la lista ys con respecto
-- a la relación r. Por ejemplo,
--    ghci> listaIndicesRelacion [2] (<) [1,2,3,5,1,4]
--    [3]
--    ghci> listaIndicesRelacion [4,2,5] (<) [1,6,3,6,2]
--    [2,3,2]
--    ghci> listaIndicesRelacion [4,2,5] (/=) [1,6,3,6,2]
--    [5,4,5]
--    ghci> listaIndicesRelacion [4,2,5] (\ x y -> odd (x+y)) [1,6,3,6,2]
--    [2,2,3]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

listaIndicesRelacion :: [Int] -> (Int -> Int -> Bool) -> [Int] -> [Int]
listaIndicesRelacion xs r ys =
  [indice x r ys | x <- xs]

indice :: Int -> (Int -> Int -> Bool) -> [Int] -> Int
indice x r ys =
  length [y | y <- ys, r x y]

-- 2ª solución
-- ===========

listaIndicesRelacion2 :: [Int] -> (Int -> Int -> Bool) -> [Int] -> [Int]
listaIndicesRelacion2 xs r ys =
```

```haskell
  map (\x -> length (filter (r x) ys)) xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Decimos que dos listas xs e ys encajan, si hay un trozo
-- no nulo al final de la lista xs que aparece al comienzo de la lista
-- ys. Por ejemplo [1,2,3,4,5,6] y [5,6,7,8] encajan, pues el trozo con
-- los dos últimos elementos de la primera lista, [5,6], aparece al
-- comienzo de la segunda lista.
--
-- Consideramos la función
--   encajadas :: Eq a => [a] -> [a] -> [a]
-- tal que (encajadas xs ys) se verifica si las listas xs e ys encajan.
-- Por ejemplo,
--   encajadas [1,2,3,4,5,6] [5,6,7,8]  ==  True
--   encajadas [4,5,6] [6,7,8]          ==  True
--   encajadas [4,5,6] [4,3,6,8]        ==  False
--   encajadas [4,5,6] [7,8]            ==  False
-- ---------------------------------------------------------------------

-- 1ª solución
encajadas1 :: Eq a => [a] -> [a] -> Bool
encajadas1 [] ys = False
encajadas1 (x:xs) ys =
  (x:xs) == take (length (x:xs)) ys || encajadas1 xs ys

-- 2ª solución
encajadas2 :: Eq a => [a] -> [a] -> Bool
encajadas2 xs ys = aux xs ys [1..length ys]
  where aux xs ys []     = False
        aux xs ys (n:ns) = drop (length xs - n) xs == take n ys ||
                           aux xs ys ns

-- 3ª solución
encajadas3 :: Eq a => [a] -> [a] -> Bool
encajadas3 xs ys =
  foldr (\ n r -> drop (length xs - n) xs == take n ys || r)
        False [1..length ys]

-- 4ª solución
encajadas4 :: Eq a => [a] -> [a] -> Bool
```

```
encajadas4 xs ys =
  any (`isPrefixOf` ys) (init (tails xs))

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se considera la función
--   repeticionesConsecutivas :: [Int] -> [Int]
-- tal que (repeticionesConsecutivas xs) es la lista con el número de
-- veces que se repiten los elementos de la lista xs de forma
-- consecutiva. Por ejemplo,
--   repeticionesConsecutivas [1,1,1,3,3,2,2]    ==  [3,2,2]
--   repeticionesConsecutivas [1,2,2,2,3,3]      ==  [1,3,2]
--   repeticionesConsecutivas [1,1,3,3,3,1,1,1]  ==  [2,3,3]
--   repeticionesConsecutivas []                 ==  []
-- ---------------------------------------------------------------------

-- 1ª solución
repeticionesConsecutivas1 :: [Int] -> [Int]
repeticionesConsecutivas1 xs = [length ys | ys <- group xs]

-- 2ª solución
repeticionesConsecutivas2 :: [Int] -> [Int]
repeticionesConsecutivas2 = map length . group

-- 3ª solución
repeticionesConsecutivas3 :: [Int] -> [Int]
repeticionesConsecutivas3 [] = []
repeticionesConsecutivas3 (x:xs) =
  1 + length (takeWhile (==x) xs) :
  repeticionesConsecutivas3 (dropWhile (==x) xs)

-- 4ª solución
repeticionesConsecutivas4 :: [Int] -> [Int]
repeticionesConsecutivas4 []     = []
repeticionesConsecutivas4 (x:xs) =
  1 + length ys : repeticionesConsecutivas3 zs
  where (ys,zs) = span (==x) xs

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un número triangular es aquel que tiene la forma
-- n*(n+1)/2 para algún n. Un número pentagonal es aquel que tiene la
```

```haskell
-- forma n*(3*n-1)/2 para algún n.
--
-- Definir la constante
--   numerosTriangulares :: [Integer]
-- cuyo valor es la lista infinita de todos los números  triangulares.
-- Por ejemplo,
--   take 10 numerosTriangulares  ==  [1,3,6,10,15,21,28,36,45,55]
--   numerosTriangulares !! 1000  ==  501501
--
-- Definir la constante
--   numerosPentagonales :: [Integer]
-- cuyo valor es la lista infinita de todos los números pentagonales.
-- Por ejemplo,
--   take 10 numerosPentagonales  ==  [1,5,12,22,35,51,70,92,117,145]
--   numerosPentagonales !! 1000  ==  1502501
--
-- Calcular los 5 primeros números que son al mismo tiempo triangulares
-- y pentagonales.
-- ---------------------------------------------------------------------

numerosTriangulares :: [Integer]
numerosTriangulares =
  [n * (n + 1) `div` 2 | n <- [1..]]

numerosPentagonales :: [Integer]
numerosPentagonales =
  [n * (3 * n - 1) `div` 2 | n <- [1..]]

interseccionInfinitaCreciente :: [Integer] -> [Integer] -> [Integer]
interseccionInfinitaCreciente (x:xs) (y:ys)
  | x == y    = x : interseccionInfinitaCreciente xs ys
  | x < y     = interseccionInfinitaCreciente xs (y:ys)
  | otherwise = interseccionInfinitaCreciente (x:xs) ys

lista5primerosTriangularesPentagonales :: [Integer]
lista5primerosTriangularesPentagonales =
  take 5 (interseccionInfinitaCreciente numerosTriangulares
                                        numerosPentagonales)

-- Los 5 primeros números que son al mismo tiempo triangulares y
```

```
-- pentagonales son: [1,210,40755,7906276,1533776805]
```

3.3. Examen 3 (30 de enero de 2018)

El examen es común con el del grupo 4 (ver página 121).

3.4. Examen 4 (8 de marzo de 2018)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 4º examen de evaluación continua (13 de marzo de 2018)
-- ---------------------------------------------------------------------

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

import Data.Matrix
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una lista de números se puede describir indicando
-- cuantas veces se repite cada elemento. Por ejemplo la lista
-- [1,1,1,3,3,2,2] se puede describir indicando que hay 3 unos, 2 treses
-- y 2 doses. De esta forma, la descripción de una lista es otra lista
-- en la que se indica qué elementos hay y cuántas veces se repiten. Por
-- ejemplo, la descripción de la lista [1,1,1,3,3,2,2] es [3,1,2,3,2,2].
--
-- La secuencia de listas que comienza en [1] y en la que cada una de
-- los demás elementos es la descripción de la lista anterior se llama
-- 'Look And Say'. Sus primeros términos son
--   [1], [1,1], [2,1], [1,2,1,1], [1,1,1,2,2,1], [3,1,2,2,1,1], ...
--
-- Definir la constante
--   lookAndSay :: [[Int]]
-- cuyo valor es la secuencia infinita 'Look And Say'. Por ejemplo,
--   take 5 lookAndSay         == [[1],[1,1],[2,1],[1,2,1,1],[1,1,1,2,2,1]]
--   lookAndSay !! 8           == [3,1,1,3,1,2,1,1,1,3,1,2,2,1]
--   length (lookAndSay !! 20) == 408
--   sum (lookAndSay !! 20)    == 679
```

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

lookAndSay :: [[Int]]
lookAndSay =
  [1] : map describe lookAndSay

describe :: [Int] -> [Int]
describe xs =
  concatMap (\ g -> [length g,head g]) (group xs)

-- ---------------------------------------------------------------------
-- Ejercicio 2. La identidad
--
--                         x-1
--   (sqrt x) = 1 + --------------
--                  1 + (sqrt x)
--
-- se puede usar para aproximar la raíz cuadrada de un número de la
-- siguiente forma:
--   y(0)   = 1
--   y(n+1) = 1 + (x-1)/(1+y(n))
--
-- Definir la función
--   aproximaRaiz :: Double -> Double
-- tal que (aproximaRaiz x) es la aproximación con 12 cifras decimales
-- exactas de la raíz cuadrada del número positivo x, calculada usando
-- el método anterior. Por ejemplo,
--   aproximaRaiz 2   ==  1.4142135623728214
--   aproximaRaiz 3   ==  1.7320508075686347
--   aproximaRaiz 10  ==  3.1622776601685203
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

aproximaRaiz :: Double -> Double
aproximaRaiz x =
  aproximaRaizAux x 1

aproximaRaizAux :: Double -> Double -> Double
```

```
aproximaRaizAux x y
  | abs (x-y^2) < 10**(-12) = y
  | otherwise               = aproximaRaizAux x (1+(x-1)/(1+y))

-- 2ª solución
-- ===========

aproximaRaiz2 :: Double -> Double
aproximaRaiz2 x =
  until (\y -> abs (x-y^2) < 10**(-12))
        (\y -> 1+(x-1)/(1+y))
        1

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--   permutaDos :: [a] -> [[a]]
-- tal que (permutaDos xs) es la familia de todas las permutaciones de
-- la lista xs en las que sólo hay dos elementos intercambiados. Por
-- ejemplo,
--   permutaDos [1..3]           ==  [[1,3,2],[2,1,3],[3,2,1]]
--   permutaDos [1..4]           ==  [[1,2,4,3],[1,3,2,4],[1,4,3,2],
--                                    [2,1,3,4],[3,2,1,4],[4,2,3,1]]
--   length (permutaDos [1..10])  ==  45
--   length (permutaDos [1..20])  ==  190
-- ---------------------------------------------------------------------

permutaDos :: [a] -> [[a]]
permutaDos []     = []
permutaDos [x]    = []
permutaDos [x1,x2] = [[x2,x1]]
permutaDos (x:xs) =
  map (x:) (permutaDos xs) ++ intercambiaPrimero (x:xs)

-- (intercambiaPrimero xs) es la lista de las lista obtenidas
-- intercambiando el primer elemento de xs con cada uno de los
-- restantes. Por ejemplo,
--    λ> intercambiaPrimero [1..5]
--    [[2,1,3,4,5],[3,2,1,4,5],[4,2,3,1,5],[5,2,3,4,1]]
intercambiaPrimero :: [a] -> [[a]]
intercambiaPrimero [] = []
```

```haskell
intercambiaPrimero (x:ys) =
  [(ys!!i : take i ys) ++ (x : drop (i+1) ys)
  | i <- [0..length ys -1]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Consideremos el siguiente tipo de matriz:
--     ( V1  V2  V3  V4  .. )
--     ( V2  V3  V4  V5  .. )
--     ( V3  V4  V5  V6  .. )
--     ( ..  ..  ..  ..  .. )
-- donde todos los elementos de cualquier diagonal paralela a la diagonal
-- secundaria son iguales. Diremos que ésta es una matriz inclinada.
--
-- Definir la función:
--   matrizInclinada :: Int -> Int -> Int -> Matrix Int
-- tal que (matrizInclinada v p q) es la matriz inclinada de p filas
-- por q columnas en la que cada valor Vi es igual a i*v. Por ejemplo,
--    λ> matrizInclinada 1 3 5
--    ( 1 2 3 4 5 )
--    ( 2 3 4 5 6 )
--    ( 3 4 5 6 7 )
--
--    λ> matrizInclinada 2 4 2
--    (  2  4 )
--    (  4  6 )
--    (  6  8 )
--    (  8 10 )
--
--    λ> matrizInclinada (-2) 3 3
--    (  -2  -4  -6 )
--    (  -4  -6  -8 )
--    (  -6  -8 -10 )
-- ---------------------------------------------------------------------

matrizInclinada :: Int -> Int -> Int -> Matrix Int
matrizInclinada v p q =
  matrix p q (\ (i,j) -> v*(i+j-1))

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función:
```

```
--   esMatrizInclinada :: Matrix Int -> Bool
-- tal que (esMatrizInclinada a) se verifica si a es una matriz
-- inclinada.
--   esMatrizInclinada (matrizInclinada 1 3 5)  ==  True
--   esMatrizInclinada (identity 4)             ==  False
-- ---------------------------------------------------------------------

esMatrizInclinada :: Matrix Int -> Bool
esMatrizInclinada a =
  all todosIguales (diagonalesSecundarias a)

-- (diagonalesSecundarias a) es la lista de las diagonales secundarias
-- de la matriz a. Por ejemplo,
--    λ> diagonalesSecundarias ((matrizInclinada 1 3 5))
--    [[1],[2,2],[3,3,3],[4,4,4],[5,5,5],[6,6],[7]]
diagonalesSecundarias :: Matrix Int -> [[Int]]
diagonalesSecundarias a =
  [[a ! (i,k-i) | i <- [1..m], 1+i <= k, k <= n+i]
   | k <- [2..m+n]]
  where m = nrows a
        n = ncols a

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    todosIguales [3,3,3]  ==  True
--    todosIguales [3,4,3]  ==  False
todosIguales :: Eq a => [a] -> Bool
todosIguales []     = True
todosIguales (x:xs) = all (==x) xs
```

3.5. Examen 5 (26 de abril de 2018)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 5º examen de evaluación continua (26 de abril de 2018)
-- =====================================================================

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

```haskell
import Data.Array
import Data.List
import I1M.Pol
import I1M.Grafo
-- import GrafoConMatrizDeAdyacencia

-- ---------------------------------------------------------------------
-- Ejercicio 1. La secuencia de Thue-Morse es una secuencia binaria
-- (formada por ceros y unos) infinita tal que el término n-ésimo se
-- define de forma recursiva de la siguiente forma:
--    d(0)    = 0
--    d(2n)   = d(n)
--    d(2n+1) = 1-d(n)
-- Por tanto, está secuencia comienza de la siguiente forma:
--    "0110100110010110100101100110100110010110..."
--
-- Definir la constante
--   thueMorse :: String
-- cuyo valor es la secuencia infinita de Thue-Morse. Por ejemplo,
--   take 10 thueMorse  ==  "0110100110"
--   take 40 thueMorse  ==  "0110100110010110100101100110100110010110"
-- ---------------------------------------------------------------------

complementario :: Char -> Char
complementario '0' = '1'
complementario '1' = '0'

thueMorse :: String
thueMorse = map thueMorseAux [0..]

thueMorseAux :: Int -> Char
thueMorseAux 0 = '0'
thueMorseAux n
  | even n    = thueMorseAux (n `div` 2)
  | otherwise = complementario (thueMorseAux (n `div` 2))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles genéricos se pueden representar con el
-- tipo de dato algebraico
--    data ArbolG t = NG t [ArbolG t]
```

```
--                    deriving (Show, Eq)
-- en la que un árbol se representa con el constructor NG seguido del
-- valor que tiene el nodo raíz del árbol y de la lista de las
-- representaciones de sus árboles hijos.
--
-- Por ejemplo, los árboles
--       3              3
--      /|\            /|\
--     / | \          / | \
--    2  4  2        2  4  3
--         / \          /|\
--        3   1        3 5 5
-- se representan por
--    a1, a2 :: ArbolG Int
--    a1 = NG 3 [NG 2 [], NG 4 [], NG 2 [NG 3 [], NG 1 []]]
--    a2 = NG 3 [NG 2 [], NG 4 [NG 3 [], NG 5 [], NG 5 []], NG 3 []]
--
-- En particular, en esta representación una hoja es un árbol genérico
-- con una lista vacía de hijos: NG x [].
--
-- Un árbol genérico es de salto fijo si el valor absoluto de la
-- diferencia de los elementos adyacentes (es decir, entre cualquier
-- nodo y cualquiera de sus hijos) es siempre la misma. Por ejemplo, el
-- árbol a1 es de salto fijo ya que el valor absoluto de sus pares de
-- elementos adyacentes son
--    |3-2| = |3-4| = |3-2| = |2-3| = |2-1| = 1
-- En cambio, el árbol a2 no es de salto fijo ya que el nodo raíz tiene
-- dos hijos con los que la diferencia en valor absoluto no es la misma
--    |3-2| = 1 =/= 0 = |3-3|
--
-- Definir la función
--    esSaltoFijo :: (Num a, Eq a) => Arbol a -> Bool
-- tal que (esSaltoFijo a) se verifica si el árbol a es de salto fijo. Por
-- ejemplo,
--    esSaltoFijo a1  ==  True
--    esSaltoFijo a2  ==  False
-- ----------------------------------------------------------------------

data ArbolG t = NG t [ArbolG t]
  deriving (Show, Eq)
```

```haskell
a1, a2 :: ArbolG Int
a1 = NG 3 [NG 2 [], NG 4 [], NG 2 [NG 3 [], NG 1 []]]
a2 = NG 3 [NG 2 [], NG 4 [NG 3 [], NG 5 [], NG 5 []], NG 3 []]

esSaltoFijo :: (Num a, Eq a) => ArbolG a -> Bool
esSaltoFijo a =
  all (==1) (listaDiferencias a)

listaDiferencias :: (Num a, Eq a) => ArbolG a -> [a]
listaDiferencias (NG v []) = []
listaDiferencias (NG v ns) =
  map (\ (NG w _) -> abs (v-w)) ns ++
  concatMap listaDiferencias ns

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Decimos que un polinomio P es regresivo si está formado
-- por un único monomio o todo monomio no nulo de P es un múltiplo
-- entero (por un monomio de coefiente entero y grado positivo) del
-- monomio no nulo de grado inmediatamente inferior. Por ejemplo,
-- + El polinomio 3x^3 es regresivo.
-- + El polinomio 6x^2 + 3x + 1 es regresivo pues el monomio 6x² es un
--   múltiplo entero de 3x (6x^2 = 2x * 3x) y éste es un múltiplo entero
--   de 1 (3x = 3x * 1).
-- + El polinomio 6x^4 - 2x^2 - 2 es regresivo pues el monomio 6x^4 es
--   un múltiplo entero de -2x^2 (6x^4 = (-2x^2) * (-2x^2)) y éste es un
--   múltiplo entero de -2 (-2x^2 = x^2 * (-2)).
-- + El polinomio 6x^3 + 3x^2 + 2 no es regresivo pues el monomio 3x^2
--   no es un múltiplo entero de 2 (3x^2 = 3/2x^2 * 2).
--
-- Definir la función
--    polinomioRegresivo :: Polinomio Int -> Bool
-- tal que (polinomioRegresivo p) se cumple si el polinomio p es regresivo.
-- Por ejemplo:
--    polinomioRegresivo p1  ==  True
--    polinomioRegresivo p2  ==  True
--    polinomioRegresivo p3  ==  True
--    polinomioRegresivo p4  ==  False
-- ---------------------------------------------------------------------
```

```haskell
p1, p2, p3, p4 :: Polinomio Int
p1 = consPol 3 3 polCero
p2 = foldr (\ (g,c) p -> consPol g c p) polCero [(2,6),(1,3),(0,1)]
p3 = foldr (\ (g,c) p -> consPol g c p) polCero [(4,6),(2,-2),(0,-2)]
p4 = foldr (\ (g,c) p -> consPol g c p) polCero [(3,6),(2,3),(0,2)]

polinomioRegresivo :: Polinomio Int -> Bool
polinomioRegresivo p =
  esPolCero p
  || esPolCero q
  || coefLider p `mod` coefLider q == 0 && polinomioRegresivo q
  where q = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. La regresión de un polinomio es el polinomio que se
-- obtiene haciendo el cociente entre monomios no nulos consecutivos. Si
-- un polinomio de coeficientes enteros es regresivo, entonces su
-- regresión también será un polinomio de coeficientes enteros. La
-- regresión de un polinomio formado por un único monomio es el
-- polinomio nulo.
--
-- Definir la función
--    regresionPolinomio :: Polinomio Int -> Polinomio Int
-- tal que (regresionPolinomio p) es la regresión del polinomio regresivo
-- p. Por ejemplo,
--    regresionPolinomio p1  =>  0
--    regresionPolinomio p2  =>  5*x
--    regresionPolinomio p3  =>  - 2*x^2
-- ---------------------------------------------------------------------

regresionPolinomio :: Polinomio Int -> Polinomio Int
regresionPolinomio p
  | esPolCero p || esPolCero q = polCero
  | otherwise = consPol (grado p - grado q)
                        (coefLider p `div` coefLider q)
                        (regresionPolinomio q)
  where q = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 4. Dado un grafo dirigido G, su grafo moral es un grafo no
```

```haskell
-- dirigido construido con el mismo conjunto de nodos y cuyas aristas
-- son las del grafo original G (consideradas sin dirección), añadiendo
-- una arista entre cada dos nodos distintos que tengan un hijo en común
-- en G. Por ejemplo, si consideramos los siguientes grafos:
--    g1 = creaGrafo D (1,3) [(1,3,0),(2,3,0)]
--    g2 = creaGrafo D (1,4) [(1,2,0),(2,4,0),(1,3,0),(3,4,0)]
--    g3 = creaGrafo D (1,4) [(1,4,0),(2,4,0),(3,4,0)]
-- + El grafo moral de g1 tiene las aristas originales junto con una
--   arista nueva que une los nodos 1 y 2.
-- + El grafo moral de g2 tiene las aristas originales junto con una
--   arista nueva que une los nodos 2 y 3.
-- + El grafo moral de g3 tiene las aristas originales junto con aristas
--   nuevas que unen los nodos 1 y 2; 1 y 3; y 2 y 3.
--
-- Definir la función
--   grafoMoral :: (Ix v, Num p, Eq p) => Grafo v p -> Grafo v p
-- tal que (grafoMoral g) es el grafo moral del grafo g. Por ejemplo,
--    λ> grafoMoral g1
--    G ND (array (1,3) [(1,[(3,0),(2,0)]),
--                       (2,[(1,0),(3,0)]),
--                       (3,[(1,0),(2,0)])])
--    λ> [(x,adyacentes (grafoMoral g1) x) | x <- nodos g1]
--    [(1,[3,2]),(2,[1,3]),(3,[1,2])]
--    λ> [(x,adyacentes (grafoMoral g2) x) | x <- nodos g2]
--    [(1,[2,3]),(2,[1,4,3]),(3,[1,2,4]),(4,[2,3])]
--    λ> [(x,adyacentes (grafoMoral g3) x) | x <- nodos g3]
--    [(1,[4,2,3]),(2,[1,4,3]),(3,[1,2,4]),(4,[1,2,3])]
-- ----------------------------------------------------------------------

g1, g2, g3 :: Grafo Int Int
g1 = creaGrafo D (1,3) [(1,3,0),(2,3,0)]
g2 = creaGrafo D (1,4) [(1,2,0),(2,4,0),(1,3,0),(3,4,0)]
g3 = creaGrafo D (1,4) [(1,4,0),(2,4,0),(3,4,0)]

grafoMoral :: (Ix v, Num p, Eq p) => Grafo v p -> Grafo v p
grafoMoral g =
  creaGrafo ND
            (minimum xs,maximum xs)
            (sinAristasSimetricas (aristas g `union` aristasMorales g))
  where xs = nodos g
```

```haskell
aristasMorales :: (Ix v, Num p) => Grafo v p -> [(v,v,p)]
aristasMorales g = [(x,y,0) | x <- xs
                            , y <- xs
                            , x /= y
                            , hijoComun g x y]
  where xs = nodos g

hijoComun :: (Ix v, Num p) => Grafo v p -> v -> v -> Bool
hijoComun g x y =
  not (null (adyacentes g x `intersect` adyacentes g y))

-- (sinAristasSimetricas as) es la lista de aristas obtenida eliminando
-- las simétricas de as. Por ejemplo,
--    λ> sinAristasSimetricas [(1,3,0),(2,3,0),(1,2,0),(2,1,0)]
--    [(1,3,0),(2,3,0),(1,2,0)]
sinAristasSimetricas :: (Eq v, Eq p) => [(v,v,p)] -> [(v,v,p)]
sinAristasSimetricas [] = []
sinAristasSimetricas ((x,y,p):as) =
  (x,y,p) : sinAristasSimetricas (as \\ [(y,x,p)])
```

3.6. Examen 6 (12 de junio de 2018)

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

3.7. Examen 7 (27 de junio de 2018)

El examen es común con el del grupo 1 (ver página 58).

3.8. Examen 8 (10 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 64).

4

Exámenes del grupo 4

María J. Hidalgo

4.1. Examen 1 (2 de noviembre de 2017)

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

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La suma de la serie
--    1/1 - 1/2 + 1/3 - 1/4 + ...+ (-1)^n/(n+1) + ...
-- es log 2.
--
-- Definir la función
--    sumaSL :: Double -> Double
-- tal que (sumaSL n) es la aproximación de (log 2) obtenida mediante n
-- términos de la serie. Por ejemplo,
--    sumaSL 2   == 0.8333333333333333
--    sumaSL 10  == 0.7365440115440116
--    sumaSL 100 == 0.6931471805599453
```

```haskell
--
-- Indicaciones:
-- + en Haskell (log x) es el logaritmo neperiano de x; por ejemplo,
--      log (exp 1) == 1.0
-- + usar la función (**) para la potencia.
-- ---------------------------------------------------------------------

sumaSL :: Double -> Double
sumaSL n = sum [(-1)**k/(k+1) | k <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    errorSL :: Double -> Double
-- tal que (errorSL x) es el menor número de términos de la serie
-- anterior necesarios para obtener su límite con un error menor que
-- x. Por ejemplo,
--    errorSL 0.1    ==  4.0
--    errorSL 0.01   ==  49.0
--    errorSL 0.001  ==  499.0
-- ---------------------------------------------------------------------

errorSL :: Double -> Double
errorSL x = head [m | m <- [1..]
                    , abs (sumaSL m - log 2) < x]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se define la raizS de un número natural como sigue. Dado
-- un número natural N, sumamos todos sus dígitos, y repetimos este
-- procedimiento hasta que quede un solo dígito al cual llamamos raizS
-- de N. Por ejemplo para 9327: 3+2+7 = 21 y 2+1 = 3. Por lo tanto, la
-- raizS de 9327 es 3.

-- Definir la función
--    raizS :: Integer -> Integer
-- tal que (raizS n) es la raizS de n. Por ejemplo.
--   raizS 9327                 == 3
--   raizS 932778214521         == 6
--   raizS 93277821452189123561 == 5
-- ---------------------------------------------------------------------
```

```haskell
raizS :: Integer -> Integer
raizS n | n < 10    = n
        | otherwise = raizS (sum (digitos n))

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    alterna :: [a] -> [a] -> [a]
-- tal que (alterna xs ys) es la lista obtenida intercalando los
-- elementos de xs e ys. Por ejemplo,
--   alterna [5,1] [2,7,4,9]     == [5,2,1,7,4,9]
--   alterna [5,1,7] [2..10]     == [5,2,1,3,7,4,5,6,7,8,9,10]
--   alterna [2..10] [5,1,7]     == [2,5,3,1,4,7,5,6,7,8,9,10]
--   alterna [2,4..12] [1,5..30] == [2,1,4,5,6,9,8,13,10,17,12,21,25,29]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que el número de elementos de
-- (alterna xs ys) es la suma de los números de elementos de xs e ys.
-- ---------------------------------------------------------------------

propAlterna :: [a] -> [a] -> Bool
propAlterna xs ys = length (alterna xs ys) == length xs + length ys

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. La sucesión de los números triangulares se obtiene
-- sumando los números naturales. Así, el 7º número triangular es
--    1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.
--
```

```haskell
-- Los primeros 10 números triangulares son
--    1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
--
-- Los divisores de los primeros 7 números triangulares son:
--     1: 1
--     3: 1,3
--     6: 1,2,3,6
--    10: 1,2,5,10
--    15: 1,3,5,15
--    21: 1,3,7,21
--    28: 1,2,4,7,14,28
--
-- Como se puede observar, 28 es el menor número triangular con más de 5
-- divisores.
--
-- Definir la función
--    menorTND :: Int -> Integer
-- tal que (menorTND n) es el menor número triangular que tiene al menos
-- n divisores. Por ejemplo,
--    menorTND 5  == 28
--    menorTND 10 == 120
--    menorTND 50 == 25200
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

menorTND1 :: Int -> Integer
menorTND1 x = head $ filter ((> x) . numeroDiv1) triangulares1

triangulares1 :: [Integer]
triangulares1 = [sum [1..n] | n <- [1..]]

numeroDiv1 :: Integer -> Int
numeroDiv1 n = length [x | x <- [1..n], rem n x == 0]

-- 2ª solución
-- ===========

menorTND2 :: Int -> Integer
```

```haskell
menorTND2 x = head $ filter ((> x) . numeroDiv2) triangulares2

triangulares2 :: [Integer]
triangulares2 = [(n*(n+1)) `div` 2 | n <- [1..]]

numeroDiv2 :: Integer -> Int
numeroDiv2 n = 2 + length [x | x <- [2..n `div` 2], rem n x == 0]

-- 3ª solución
-- ===========

menorTND3 :: Int -> Integer
menorTND3 x = head $ filter ((> x) . numeroDiv3) triangulares3

triangulares3 :: [Integer]
triangulares3 = scanl (+) 1 [2..]

numeroDiv3 :: Integer -> Int
numeroDiv3 n = product $ map ((+1) . length) $ group $ primeFactors n

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

--    ghci> menorTND1 100
--    73920
--    (3.44 secs, 1,912,550,376 bytes)
--    ghci> menorTND2 100
--    73920
--    (1.72 secs, 951,901,856 bytes)
--    ghci> menorTND3 100
--    73920
--    (0.02 secs, 7,430,752 bytes)
```

4.2. Examen 2 (5 de diciembre de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (5 de diciembre de 2017)
-- ---------------------------------------------------------------------

-- Nota: La puntuación de cada ejercicio es 2.5 puntos.
```

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

import Data.List
import Data.Maybe
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercico 1. Definir la función
--    sublistasSuma :: (Num a, Eq a) => [a] -> a -> [[a]]
-- tal que (sublistasSuma xs m) es la lista de sublistas de xs cuya suma
-- es m. Por ejemplo,
--    sublistasSuma [1..5] 10 == [[2,3,5],[1,4,5],[1,2,3,4]]
--    sublistasSuma [1..5] 23 == []
--    length (sublistasSuma [1..20] 145) == 5337
-- ---------------------------------------------------------------------

-- 1ª solución (por orden superior):
sublistasSuma1 :: (Num a, Eq a) => [a] -> a -> [[a]]
sublistasSuma1 xs m = filter ((==m) . sum) (subsequences xs)

-- 2ª solución (por recursión:
sublistasSuma2 :: [Int] -> Int -> [[Int]]
sublistasSuma2 xs = aux (sort xs)
  where aux [] m = []
        aux (y:ys) m | y == m    = [[m]]
                     | y > m     = []
                     | otherwise = aux ys m ++ map (y:) (aux ys (m-y))

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. La sucesión de Padovan es la secuencia de números
-- enteros P(n) definida por los valores iniciales
--    P(0) = P(1) = P( 2) = 1,
-- y por la relación
--    P (n) = P(n–2) + P(n–3) .
--
-- Los primeros valores de P(n) son
--    1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, 21, 28, 37,...
```

```
--
-- Definir la funciones
--    padovan :: Int -> Integer
--    sucPadovan :: [Integer]
-- tales que
--    sucPadovan es la sucesión así construida, y
--    (padovan n) es el término n-simo de la sucesión
-- Por ejemplo,
--    padovan 10                    == 12
--    padovan 100                   == 1177482265857
--    length $ show $ padovan 1000 == 122
--    take 14 sucPadovan            == [1,1,1,2,2,3,4,5,7,9,12,16,21,28]
-- ----------------------------------------------------------------------

-- 1ª solución
-- ===========

padovan1 :: Int -> Integer
padovan1 n | n <= 2    = 1
           | otherwise = padovan1 (n-2) + padovan1 (n-3)

sucPadovan1 :: [Integer]
sucPadovan1 = map padovan1 [0..]

-- 2ª solución
-- ===========

sucPadovan2 :: [Integer]
sucPadovan2 = 1:1:1: zipWith (+) sucPadovan2 (tail sucPadovan2)

padovan2 :: Int -> Integer
padovan2 n = sucPadovan2 `genericIndex` n

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

--    ghci> sucPadovan1 !! 60
--    15346786
--    (12.53 secs, 6,752,729,800 bytes)
--    ghci> sucPadovan2 !! 60
```

```
--    15346786
--    (0.00 secs, 152,648 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck la siguiente propiedad:
--                 n
--                ---
--                \
--                /    P(j)   = P(n+4) - 2
--                ---
--                j=0
-- ---------------------------------------------------------------------

-- La propiedad es
propPadovan :: Int -> Bool
propPadovan n = sum (take m sucPadovan2) == padovan2 (m+4) - 2
  where m = abs n

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles con un número variable de sucesores 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
--     / \       / \           / \
--    8   3     8   3         8   3
--        |        /|\       /|\  |
--        4       4 5 6     4 5 6 7
-- se representan por
--    ej1, ej2, ej3 :: Arbol Int
--    ej1 = N 1 [N 8 [],N 3 [N 4 []]]
--    ej2 = N 1 [N 8 [], N 3 [N 4 [], N 5 [], N 6 []]]
--    ej3 = N 1 [N 8 [N 4 [], N 5 [], N 6 []], N 3 [N 7 []]]
--
-- Definir la función
```

```
--     caminosDesdeRaiz :: Arbol a -> [[a]]
-- tal que (caminosDesdeRaiz x) es la lista de todos los caminos desde
-- la raiz. Por ejemplo,
--     caminosDesdeRaiz ej1 == [[1,8],[1,3,4]]
--     caminosDesdeRaiz ej2 == [[1,8],[1,3,4],[1,3,5],[1,3,6]]
--     caminosDesdeRaiz ej3 == [[1,8,4],[1,8,5],[1,8,6],[1,3,7]]
-- ---------------------------------------------------------------------

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

ej1, ej2, ej3 :: Arbol Int
ej1 = N 1 [N 8 [],N 3 [N 4 []]]
ej2 = N 1 [N 8 [], N 3 [N 4 [], N 5 [], N 6 []]]
ej3 = N 1 [N 8 [N 4 [], N 5 [], N 6 []], N 3 [N 7 []]]

caminosDesdeRaiz :: Arbol a -> [[a]]
caminosDesdeRaiz (N r []) = [[r]]
caminosDesdeRaiz (N r as) = map (r:) (concatMap caminosDesdeRaiz as)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. La lista [7, 10, 12, 1, 2, 4] no está ordenada, pero si
-- consideramos las listas que se pueden formar cíclicamente a partir de
-- cada elemento, obtenemos:
--    [7, 10, 12, 1, 2, 4]
--    [10, 12, 1, 2, 4, 7]
--    [12, 1, 2, 4, 7, 10]
--    [1, 2, 4, 7, 10, 12]  ** ordenada **
--    [2, 4, 7, 10, 12, 1]
--    [4, 7, 10, 12, 1, 2]
-- Se observa que una de ellas está ordenada.
--
-- Se dice que una lista [x(0), ..., x(n)] es cíclicamente ordenable
-- si existe un índice i tal que la lista
--    [x(i), x(i+1), ..., x(n), x(0), ..., x(i-1)]
-- está ordenada.
--
-- Definir la función
--    ciclicamenteOrdenable :: Ord a => [a] -> Bool
-- tal que (ciclicamenteOrdenable xs) se verifica si xs es una lista
```

```haskell
-- cíclicamente ordenable. Por ejemplo,
--    ciclicamenteOrdenable [7,10,12,1,2,4]  ==  True
--    ciclicamenteOrdenable [7,20,12,1,2,4]  ==  False
-- ---------------------------------------------------------------------

ciclicamenteOrdenable :: Ord a => [a] -> Bool
ciclicamenteOrdenable xs =
  any ordenada (zipWith (++) (tails xs) (inits xs))

ordenada :: Ord a => [a] -> Bool
ordenada xs = and (zipWith (<=) xs (tail xs))

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    posicionOC :: Ord a => [a] -> Maybe Int
-- tal que (posicionOC xs) es (Just i) si i es el índice a partir del
-- cual la lista xs está ordenada cíclicamente; o bien Nothing, en caso
-- contrario. Por ejemplo,
--    posicionOC [7,10,12,1,2,4]           == Just 3
--    posicionOC [4,5,6,1,2,3]             == Just 3
--    posicionOC [4,5,6,1,2,3,7]           == Nothing
--    posicionOC ([10^3..10^4] ++ [1..999]) == Just 9001
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

posicionOC :: Ord a => [a] -> Maybe Int
posicionOC xs | null is   = Nothing
              | otherwise = Just (head is)
  where is = ciclosOrdenados xs

-- (cicloOrdenado xs i) se verifica si el ciclo i-ésimo de xs está
-- ordenado; es decir, si
--    [x(i), x(i+1), ..., x(n), x(0), ..., x(i-1)]
-- está ordenado. Por ejemplo,
--    cicloOrdenado [4,5,6,1,2,3] 3  ==  True
--    cicloOrdenado [4,5,6,1,2,3] 2  ==  False
--    cicloOrdenado [4,5,6,1,2,3] 4  ==  False
cicloOrdenado :: Ord a => [a] -> Int -> Bool
```

```haskell
cicloOrdenado xs i = ordenada (drop i xs ++ take i xs)

-- (ciclosOrdenados xs) es el conjunto de índices i tales que el ciclo
-- i-ésimo de xs está ordenado. Por ejemplo,
--    ciclosOrdenados [7,10,12,1,2,4]  ==  [3]
--    ciclosOrdenados [7,20,12,1,2,4]  ==  []
ciclosOrdenados :: Ord a => [a] -> [Int]
ciclosOrdenados xs =
  [i | i <- [0..length xs - 1]
     , cicloOrdenado xs i]

-- 2ª solución
-- ===========

posicionOC2 :: Ord a => [a] -> Maybe Int
posicionOC2 = listToMaybe . ciclosOrdenados
```

4.3. Examen 3 (30 de enero de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 1, 2 y 3)
-- 3º examen de evaluación continua (30 de enero de 2018)
-- ---------------------------------------------------------------------

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Una secuencia de números es estrictamente oscilante
-- si el orden relativo entre términos consecutivos se va alternando. Se
-- pueden dar dos casos de secuencias estrictamente oscilantes:
-- + El primer término es estrictamente menor que el segundo, el segundo
--   es estrictamente mayor que el tercero, el tercero es estrictamente
--   menor que el cuarto, el cuarto es estrictamente mayor que el
--   quinto, y así sucesivamente. Por ejemplo las secuencias
--      0, 1, -1, 2, -2, 3, -3, 4, -4
--      1, 10, 5, 28, 3, 12, 4
```

```
-- + El primer término es estrictamente mayor que el segundo, el segundo
--   es estrictamente menor que el tercero, el tercero es estrictamente
--   mayor que el cuarto, el cuarto es estrictamente menor que el
--   quinto, y así sucesivamente. Por ejemplo las secuencias
--      0, -1, 1, -2, 2, -3, 3, -4, 4
--      10, 5, 28, 3, 12, 4, 24
--
-- Definir la función
--   estrictamenteOscilante :: [Int] -> Bool
-- tal que (estrictamenteOscilante xs) se cumple si y sólo si la
-- secuencia de números enteros xs es estrictamente oscilante. Por
-- ejemplo,
--   estrictamenteOscilante [0,1,-1,2,-2,3,-3,4,-4]  ==  True
--   estrictamenteOscilante [1,10,5,28,3,12,4]       ==  True
--   estrictamenteOscilante [0,-1,1,-2,2,-3,3,-4,4]  ==  True
--   estrictamenteOscilante [10,5,28,3,12,4,24]      ==  True
--   estrictamenteOscilante [1,1,1,1,1]              ==  False
--   estrictamenteOscilante [1,2,3,4,5,6]            ==  False
-- ---------------------------------------------------------------------

-- 1ª solución
-- ============

estrictamenteOscilante :: [Int] -> Bool
estrictamenteOscilante xs =
  signosAlternados [x-y | (x,y) <- zip xs (tail xs)]

signosAlternados xs =
  and [x*y < 0 | (x,y) <- zip xs (tail xs)]

-- 2ª solución
-- ============

estrictamenteOscilante2 :: [Int] -> Bool
estrictamenteOscilante2 (x:y:z:xs) =
  signum (x-y) /= signum (y-z) && estrictamenteOscilante2 (y:z:xs)
estrictamenteOscilante2 _ = True

-- 3ª solución
-- ============
```

```haskell
estrictamenteOscilante3 :: [Int] -> Bool
estrictamenteOscilante3 xs =
  all (== (-1)) (signos (*) (signos (-) xs))

signos :: (Int -> Int -> Int) -> [Int] -> [Int]
signos f xs =
  [signum (f y x) | (x,y) <- zip xs (tail xs)]

-- 4ª solución
-- ===========

estrictamenteOscilante4 :: [Int] -> Bool
estrictamenteOscilante4 = all (-1 ==) . signos (*) . signos (-)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Cualquier número natural admite una representación en
-- base 3; es decir, se puede escribir como combinación lineal de
-- potencias de 3, con coeficientes 0, 1 ó 2. Por ejemplo,
--    46 = 1*3^0 + 0*3^1 + 2*3^2 + 1*3^3
--   111 = 0*3^0 + 1*3^1 + 0*3^2 + 1*3^3 +1*3^4
-- Esta representación se suele expresar como la lista de los
-- coeficientes de las potencias sucesivas de 3. Así, 46 es [1,0,2,1] y
-- 111 es [0,1,0,1,1].
--
-- De esta forma, los números que no tienen el 2 en su representación en
-- base 3 son:
--    0 [0], 1 [1],
--    3 [0,1], 4 [1,1],
--    9 [0,0,1], 10 [1,0,1], 12 [0,1,1], 13 [1,1,1], ...
--
-- Definir la lista infinita
--    sin2enBase3 :: [Integer]
-- cuyo valor es la lista formada por los números naturales que no tienen
-- el 2 en su representación en base 3. Por ejemplo,
--    λ> take 22 sin2enBase3
--    [0,1,3,4,9,10,12,13,27,28,30,31,36,37,39,40,81,82,84,85,90,91]
-- ---------------------------------------------------------------------

-- 1ª solución
```

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

sin2enBase3 :: [Integer]
sin2enBase3 = map base3Adecimal (filter (all (/=2)) enBase3)

-- enBase3 es la lista de los números enteros en base 3. Por ejemplo,
--    λ> take 10 enBase3
--    [[0],[1],[2],[0,1],[1,1],[2,1],[0,2],[1,2],[2,2],[0,0,1]]
enBase3 :: [[Integer]]
enBase3 = map decimalAbase3 [0..]

-- (decimalAbase3 n) es la representación de n en base 3. Por
-- ejemplo,
--    decimalAbase3 34  ==  [1,2,0,1]
-- ya que 1*3^0 + 2*3^1 + 0*3^2 + 1*3^3 = 34.
decimalAbase3 :: Integer -> [Integer]
decimalAbase3 n
  | n < 3     = [n]
  | otherwise = n `mod` 3 : decimalAbase3 (n `div` 3)

-- (base3Adecimal xs) es el número decimal cuya representación
-- en base 3b es xs. Por ejemplo,
--    base3Adecimal [1,2,0,1]  ==  34
-- ya que 1*3^0 + 2*3^1 + 0*3^2 + 1*3^3 = 34.
base3Adecimal :: [Integer] -> Integer
base3Adecimal xs = sum (zipWith (\x k -> x*3^k) xs [0..])

-- Otra definición de la función anterior es
base3Adecimal2 :: [Integer] -> Integer
base3Adecimal2 = foldr (\x a -> x+3*a) 0

-- 2ª solución
-- ===========

sin2enBase3b :: [Integer]
sin2enBase3b = map base3Adecimal (filter (notElem 2) enBase3)

-- 3ª solución
-- ===========
```

```haskell
sin2enBase3c :: [Integer]
sin2enBase3c =
  0 : aux 0 [0]
  where aux n ns = map ((3^n)+) ns ++ aux (n+1) (ns ++ map ((3^n)+) ns)

-- 4ª solución
-- ===========

sin2enBase3d :: [Integer]
sin2enBase3d =
  0 : 1 : concatMap (\n -> [3*n,3*n+1]) (tail sin2enBase3d)

-- 5ª solución
-- ===========

sin2enBase3e :: [Integer]
sin2enBase3e = 0 : aux
  where aux = 1 : concat [[3*n,3*n+1] | n <- aux]

-- 6ª solución
-- ===========

sin2enBase3f :: [Integer]
sin2enBase3f = map base3Adecimal enBase3sin2

-- enBase3sin2 es la lista de los números en base 3 que no tienen el
-- dígito 2. Por ejemplo,
--    λ> take 9 enBase3sin2
--    [[0],[1],[0,1],[1,1],[0,0,1],[1,0,1],[0,1,1],[1,1,1],[0,0,0,1]]
enBase3sin2 :: [[Integer]]
enBase3sin2 = [0] : aux
  where aux = [1] : concat [[0:xs,1:xs] | xs <- aux]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. ¿Cuál será el próximo año que no tenga 2 en su
-- representación en base 3?
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> head (dropWhile (<2018) sin2enBase3)
```

```
--    2187

-- ---------------------------------------------------------------------
-- Ejercicio 3.1 Definir la función
--    listasParciales :: [a] -> [[a]]
-- tal que (listasParciales xs) es la lista obtenida agrupando los
-- elementos de la lista infinita xs de forma que la primera tiene 0
-- elementos; la segunda, el primer elemento de xs; la tercera, los dos
-- siguientes; y así sucesivamente. Por ejemplo,
--    λ> take 6 (listasParciales [1..])
--    [[],[1],[2,3],[4,5,6],[7,8,9,10],[11,12,13,14,15]]
--    λ> take 6 (listasParciales [1,3..])
--    [[],[1],[3,5],[7,9,11],[13,15,17,19],[21,23,25,27,29]]
-- ---------------------------------------------------------------------

-- 1ª solución
listasParciales :: [a] -> [[a]]
listasParciales = aux 0
  where aux n xs = ys : aux (n+1) zs
          where (ys,zs) = splitAt n xs

-- 2ª solución
listasParciales2 :: [a] -> [[a]]
listasParciales2 = aux 0
  where aux n xs = ys : aux (n+1) zs
          where (ys,zs) = (take n xs, drop n xs)

-- 3ª solución
listasParciales3 :: [a] -> [[a]]
listasParciales3 xs = aux xs 0
  where aux xs n = take n xs : aux (drop n xs) (n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2 Definir la función
--    sumasParciales  :: [Int] -> [Int]
-- tal que (sumasParciales xs) es la lista de las sumas de las listas
-- parciales de la lista infinita xs. Por ejemplo,
--    λ> take 15 (sumasParciales [1..])
--    [0,1,5,15,34,65,111,175,260,369,505,671,870,1105,1379]
--    λ> take 15 (sumasParciales [1,3..])
```

```
--    [0,1,8,27,64,125,216,343,512,729,1000,1331,1728,2197,2744]
-- ---------------------------------------------------------------------

-- 1ª solución
sumasParciales :: [Int] -> [Int]
sumasParciales xs = map sum (listasParciales xs)

-- 2ª solución
sumasParciales2 :: [Int] -> [Int]
sumasParciales2 = map sum . listasParciales

-- ---------------------------------------------------------------------
-- Ejercicio 3.3 Comprobar con QuickChek que, para todo número natural
-- n, el término n-ésimo de (sumasParciales [1,3..]) es n^3.
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

-- La propiedad es
prop_sumasParciales :: Int -> Property
prop_sumasParciales n =
  n >= 0 ==> (sumasParciales [1,3..] !! n == n^3)

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

-- 2ª solución
-- ===========

-- La propiedad es
prop_sumasParciales2 :: Positive Int -> Bool
prop_sumasParciales2 (Positive n) =
  sumasParciales [1,3..] !! n == n^3

-- La comprobación es
--    λ> quickCheck prop_sumasParciales2
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- 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 8 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 8 [], N 1 []]]
--
-- Definir la función
--    nodosConNHijosMaximales :: Arbol a -> [a]
-- tal que (nodosConNHijosMaximales x) es la lista de los nodos del
-- árbol x que tienen una cantidad máxima de hijos. Por ejemplo,
--    nodosConNHijosMaximales ej1  ==  [1]
--    nodosConNHijosMaximales ej2  ==  [3,7]
-- ---------------------------------------------------------------------

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

ej1, ej2, ej3 :: Arbol Int
ej1 = N 1 [N 2 [], N 3 [N 4 []]]
ej2 = N 3 [N 5 [N 6 []],
           N 4 [],
           N 7 [N 2 [], N 8 [], N 1 []]]
ej3 = N 3 [N 5 [N 6 []],
           N 4 [N 1 [N 2 [], N 8 [N 9 []]]],
           N 12 [N 20 [], N 0 [], N 10 []]]

-- 1ª solución
```

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

nodosConNHijosMaximales :: Arbol a -> [a]
nodosConNHijosMaximales x =
  let nh = nHijos x
      m  = maximum (map fst nh)
  in map snd (filter (\ (l,v) -> m == l) (nHijos x))

-- (nHijos x) es la lista de los pares (k,n) donde n es un nodo del
-- árbol x y k es su número de hijos. Por ejemplo,
--    λ> nHijos ej1
--    [(2,1),(0,2),(1,3),(0,4)]
--    λ> nHijos ej2
--    [(3,3),(1,5),(0,6),(0,4),(3,7),(0,2),(0,8),(0,1)]
nHijos :: Arbol a -> [(Int,a)]
nHijos (N r []) = [(0,r)]
nHijos (N r xs) = (length xs, r) : concatMap nHijos xs

-- 2ª solución
-- ===========

nodosConNHijosMaximales2 :: Arbol a -> [a]
nodosConNHijosMaximales2 x =
  [y | (n,y) <- nh, n == m]
  where nh = nHijos x
        m  = maximum (map fst nh)

-- 3ª solución
-- ===========

nodosConNHijosMaximales3 :: Ord a => Arbol a -> [a]
nodosConNHijosMaximales3 =
  map snd . head . groupBy eq . sortBy (flip compare) . nHijos
  where eq (a,b) (c,d) = a == c
```

4.4. Examen 4 (15 de marzo de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 4º examen de evaluación continua (15 de marzo de 2018)
```

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

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

import Data.List
import Data.Array
import Graphics.Gnuplot.Simple

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Sea a(0), a(1), a(2), ... una sucesión de enteros
-- definida por:
-- + a(0) = 1 y
-- + a(n) es la suma de los dígitos de todos los términos anteriores,
--   para n ≥ 1.
--
-- Los primeros términos de la sucesión son:
--      1, 1, 2, 4, 8, 16, 23, 28, 38, 49, ...
--
-- Definir la constante
--    sucSumaDigitos :: [Integer]
-- tal que sucSumaDigitos es la sucesión anterior. Por ejemplo,
--    take 10 sucSumaDigitos   == [1,1,2,4,8,16,23,28,38,49]
--    sucSumaDigitos !! (10^3) == 16577
--    sucSumaDigitos !! (10^4) == 213677
--    sucSumaDigitos !! (10^5) == 2609882
--    sucSumaDigitos !! (10^6) == 31054319
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

sucSumaDigitos1 :: [Integer]
sucSumaDigitos1 = map termino [0..]

termino :: Integer -> Integer
termino 0 = 1
termino n = sum [sumaDigitos (termino k) | k <- [0..n-1]]
```

```haskell
-- 2ª definición
-- =============

sucSumaDigitos2 :: [Integer]
sucSumaDigitos2 = 1 : iterate f 1
  where f x = x + sumaDigitos x

sumaDigitos :: Integer -> Integer
sumaDigitos x | x < 10 = x
              | otherwise = x `mod` 10 + sumaDigitos (x `div` 10)

-- 3ª definición
-- =============

sucSumaDigitos3 :: [Integer]
sucSumaDigitos3 = 1 : unfoldr (\x -> Just (x, f x)) 1
  where f x = x + sumaDigitos x


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

--    λ> maximum (take 23 sucSumaDigitos1)
--    161
--    (11.52 secs, 1,626,531,216 bytes)
--    λ> maximum (take 23 sucSumaDigitos2)
--    161
--    (0.01 secs, 153,832 bytes)
--    λ> maximum (take 23 sucSumaDigitos3)
--    161
--    (0.01 secs, 153,072 bytes)
--
--    λ> maximum (take 200000 sucSumaDigitos2)
--    5564872
--    (1.72 secs, 233,208,200 bytes)
--    λ> maximum (take 200000 sucSumaDigitos3)
--    5564872
--    (3.46 secs, 438,250,928 bytes)

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

```haskell
-- Ejercicio 1.2. Definir la función
--    grafica :: Integer -> IO ()
-- tal que (grafica n) es la gráfica de los primeros n términos de la
-- sucesión anterior.
-- ---------------------------------------------------------------------

grafica :: Integer -> IO ()
grafica n =
  plotList [Key Nothing] (genericTake n sucSumaDigitos2)

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. La suma de las sumas de los cuadrados de los
-- divisores de los 6 primeros números enteros positivos es
--      1² + (1²+2²) + (1²+3²) + (1²+2²+4²) + (1²+5²) + (1²+2²+3²+6²)
--    = 1  + 5       + 10      + 21         + 26      + 50
--    = 113
--
-- Definir la función
--    sumaSumasCuadradosDivisores :: Integer -> Integer
-- tal que (sumaSumasCuadradosDivisores n) es la suma de las sumas de
-- los cuadrados de los divisores de los n primeros números enteros
-- positivos. Por ejemplo,
--    sumaSumasCuadradosDivisores 6       ==  113
--    sumaSumasCuadradosDivisores (10^3)  ==  401382971
--    sumaSumasCuadradosDivisores (10^6)  ==  400686363385965077
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

sumaSumasCuadradosDivisores1 :: Integer -> Integer
sumaSumasCuadradosDivisores1 n =
  sum (map (^2) (concatMap divisores [1..n]))

-- (divisores x) es la lista de divisores de n. Por ejemplo,
--    divisores 6 == [1,2,3,6]
divisores :: Integer -> [Integer]
divisores n = [y | y <- [1..n], n `mod` y == 0]

-- 2ª definición
```

```haskell
sumaSumasCuadradosDivisores2 :: Integer -> Integer
sumaSumasCuadradosDivisores2 x =
  sum (zipWith (*) (map (x `div`) xs) (map (^2) xs))
  where xs = [1..x]

-- 3ª definición
sumaSumasCuadradosDivisores3 :: Integer -> Integer
sumaSumasCuadradosDivisores3 n =
  sum $ zipWith (*) ((map (^2)  xs)) (zipWith div (repeat n) xs)
  where xs = takeWhile (<= n) [1..]

-- 4ª definición
sumaSumasCuadradosDivisores4 :: Integer -> Integer
sumaSumasCuadradosDivisores4 n =
  sum [k^2 * (n `div` k) | k <- [1..n]]

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

--    λ> sumaSumasCuadradosDivisores1 (2*10^3)
--    3208172389
--    (3.10 secs, 412,873,104 bytes)
--    λ> sumaSumasCuadradosDivisores2 (2*10^3)
--    3208172389
--    (0.03 secs, 2,033,352 bytes)
--    λ> sumaSumasCuadradosDivisores3 (2*10^3)
--    3208172389
--    (0.03 secs, 2,178,496 bytes)
--    λ> sumaSumasCuadradosDivisores4 (2*10^3)
--    3208172389
--    (0.03 secs, 1,924,072 bytes)
--
--    λ> sumaSumasCuadradosDivisores2 (4*10^5)
--    25643993117893355
--    (1.93 secs, 378,385,664 bytes)
--    λ> sumaSumasCuadradosDivisores3 (4*10^5)
--    25643993117893355
--    (2.08 secs, 407,185,792 bytes)
--    λ> sumaSumasCuadradosDivisores4 (4*10^5)
--    25643993117893355
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    sumaSumasCuadradosDivisoresInter :: IO ()
-- que realice lo mismo que la función sumaSumasCuadradosDivisores, pero
-- de forma interactiva. Por ejemplo,
--    λ> sumaSumasCuadradosDivisoresInter
--       Escribe un número:
--       1234
--       La suma de los cuadrados de sus divisores es: 753899047
-- ---------------------------------------------------------------------

sumaSumasCuadradosDivisoresInter :: IO ()
sumaSumasCuadradosDivisoresInter = do
  putStrLn "Escribe un número: "
  c <- getLine
  let n = read c
  putStrLn ("La suma de los cuadrados de sus divisores es: "
            ++ show (sumaSumasCuadradosDivisores4 n))

-- ---------------------------------------------------------------------
-- Ejercicio 3. Dados los vectores v = [1,2,3,4] y w = [1,-2,5,0] con
-- el primer elemento común, construimos una matriz 4x4, cuya primera
-- fila es v, su primera columna es w, y de forma que cada elemento es
-- la suma de sus tres vecinos que ya tienen un valor.
--
-- La matriz se construye de forma incremental como sigue:
--     1  2  3  4       1  2  3  4      1  2  3  4       1  2   3   4
--    -2               -2  1          -2  1  6         -2  1   6  13
--     5                5              5  4             5  4  11
--     0                0              0                0  9
--
-- Definir la función
--    matrizG :: Array Int Int -> Array Int Int -> Array (Int,Int) Int
-- tal que (matrizG v w) es la matriz cuadrada generada por los vectores
-- v y w (que se supone que tienen la misma dimensión). Por ejemplo,
--    λ> matrizG (listArray (1,4) [1..4]) (listArray (1,4) [1,-2,5,0])
--    array ((1,1),(4,4)) [((1,1), 1),((1,2),2),((1,3), 3),((1,4), 4),
--                         ((2,1),-2),((2,2),1),((2,3), 6),((2,4),13),
--                         ((3,1), 5),((3,2),4),((3,3),11),((3,4),30),
```

```
--                          ((4,1), 0),((4,2),9),((4,3),24),((4,4),65)]
-- ---------------------------------------------------------------------

matrizG :: Array Int Int -> Array Int Int -> Array (Int,Int) Int
matrizG v w = p
  where p = array ((1,1), (n,n))
                  [((i,j), f i j) | i <- [1..n], j <- [1..n]]
        n = snd $ bounds v
        f 1 j = v ! j
        f i 1 = w ! i
        f i j = p ! (i-1,j-1) + p!(i-1,j) + p!(i,j-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. 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           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
--    descomposicion :: Arbol t -> [(t, [t])]
-- tal que (descomposicion ar) es una lista de pares  (x,xs) donde x es
-- un nodo y xs es la lista de hijos de x. Por ejemplo,
--    descomposicion ej1 == [(1,[2,3]),(2,[4,5]),(4,[]),(5,[]),(3,[])]
--    descomposicion ej2 == [(1,[2,3]),(2,[4]),(4,[]),(3,[])]
--    descomposicion ej3 == [(1,[2,7,3]),(2,[4,5]),(4,[]),(5,[]),
--                           (7,[]),(3,[])]
-- ---------------------------------------------------------------------
```

```haskell
data Arbol a = N a [Arbol a]
  deriving (Show,Eq)

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 []]

descomposicion :: Arbol t -> [(t, [t])]
descomposicion (N r []) = [(r,[])]
descomposicion (N r as) = (r, map raiz as) : concatMap descomposicion as

-- (raiz t) es la raíz del árbol t. Por ejemplo,
--    raiz (N 8 [N 2 [N 4 []], N 3 []])  ==  8
raiz :: Arbol t -> t
raiz (N r _) = r

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función recíproca
--    descAarbol :: Eq t => [(t, [t])] -> Arbol t
-- tal que (descAarbol ps) es el árbol correspondiente a ps. Ejemplos,
--    (descAarbol (descomposicion ej1) == ej1) == True
--    (descAarbol (descomposicion ej2) == ej2) == True
--    (descAarbol (descomposicion ej3) == ej3) == True
-- ---------------------------------------------------------------------

descAarbol :: Eq t => [(t, [t])] -> Arbol t
descAarbol ps@((r,xs):qs) = aux r ps
  where aux r [] = N r []
        aux r ps = N r [aux h qs | h <- hs]
          where hs = head [xs | (x,xs) <- ps, x == r]
                qs = delete (r,hs) ps
```

4.5. Examen 5 (3 de mayo de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 5º examen de evaluación continua (3 de mayo de 2018)
-- ---------------------------------------------------------------------
```

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

import Data.List
import I1M.Cola
import qualified Data.Matrix as M
import Data.Array
import Test.QuickCheck
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número entero n es muy primo si es n primo y todos
-- los números que resultan de ir suprimimiendo la última cifra también
-- son primos. Por ejemplo, 7193 es muy primo pues los números 7193,
-- 719, 71 y 7 son todos primos.
--
-- Definir la función
--    muyPrimo :: Int -> Bool
-- que (muyPrimo n) se verifica si n es muy primo. Por ejemplo,
--    muyPrimo 7193  == True
--    muyPrimo 71932 == False
-- ---------------------------------------------------------------------

muyPrimo :: Int -> Bool
muyPrimo n | n < 10    = isPrime n
           | otherwise = isPrime n && muyPrimo (n `div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. ¿Cuántos números de cinco cifras son muy primos?
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> length (filter muyPrimo [10^4..99999])
--    15

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    muyPrimosF :: FilePath -> FilePath -> IO ()
-- tal que al evaluar (muyPrimosF f1 f2) lee el contenido del fichero
```

```haskell
-- f1 (que estará compuesto por números, cada uno en una línea) y
-- escribe en el fichero f2 los números de f1 que son muy primos, cada
-- uno en una línea. Por ejemplo, si el contenido de ej.txt es
--     7193
--     1870
--     271891
--     23993
--     1013
-- y evaluamos (muyPrimosF "ej.txt" "sol.txt"), entonces el contenido
-- del fichero "sol.txt" será
--     7193
--     23993
-- ---------------------------------------------------------------------

muyPrimosF :: FilePath -> FilePath -> IO ()
muyPrimosF f1 f2 = do
  cs <- readFile f1
  writeFile f2 (( unlines
                . map show
                . filter muyPrimo
                . map read
                . lines )
               cs)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se define la relación de orden entre las colas como el
-- orden lexicográfico. Es decir, la cola c1 es "menor" que c2 si el
-- primer elemento de c1 es menor que el primero de c2, o si son
-- iguales, el resto de la cola c1 es "menor" que el resto de la cola
-- c2.
--
-- Definir la función
--    menorCola :: Ord a => Cola a -> Cola a -> Bool
-- tal que (menorCola c1 c2) se verifica si c1 es "menor" que c2. Por
-- ejemplo, para las colas
--    c1 = foldr inserta vacia [1..20]
--    c2 = foldr inserta vacia [1..5]
--    c3 = foldr inserta vacia [3..10]
--    c4 = foldr inserta vacia [4,-1,7,3,8,10,0,3,3,4]
-- se verifica que
```

```
--    menorCola c1 c2    == False
--    menorCola c2 c1    == True
--    menorCola c4 c3    == True
--    menorCola vacia c1 == True
--    menorCola c1 vacia == False
-- ---------------------------------------------------------------------

menorCola :: Ord a => Cola a -> Cola a -> Bool
menorCola c1 c2 | esVacia c1 = True
                | esVacia c2 = False
                | a1 < a2    = True
                | a1 > a2    = False
                | otherwise  = menorCola r1 r2
  where a1 = primero c1
        a2 = primero c2
        r1 = resto c1
        r2 = resto c2

c1, c2, c3, c4 :: Cola Int
c1 = foldr inserta vacia [1..20]
c2 = foldr inserta vacia [1..5]
c3 = foldr inserta vacia [3..10]
c4 = foldr inserta vacia [4,-1,7,3,8,10,0,3,3,4]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Una triangulación de un polígono es una división del
-- área en un conjunto de triángulos, de forma que la unión de todos
-- ellos es igual al polígono original, y cualquier par de triángulos es
-- disjunto o comparte únicamente un vértice o un lado. En el caso de
-- polígonos convexos, la cantidad de triangulaciones posibles depende
-- únicamente del número de vértices del polígono.
--
-- Si llamamos T(n) al número de triangulaciones de un polígono de n
-- vértices, se verifica la siguiente relación de recurrencia:
--     T(2) = 1
--     T(n) = T(2)*T(n-1) + T(3)*T(n-2) + ... + T(n-1)*T(2)
--
-- Definir la función
--    numeroTriangulaciones :: Integer -> Integer
-- tal que (numeroTriangulaciones n) es el número de triangulaciones de
```

```haskell
-- un polígono convexo de n vértices. Por ejemplo,
--    numeroTriangulaciones 3  == 1
--    numeroTriangulaciones 5  == 5
--    numeroTriangulaciones 6  == 14
--    numeroTriangulaciones 7  == 42
--    numeroTriangulaciones 50 == 131327898242169365477991900
--    numeroTriangulaciones 100
--      ==  577433580696013577821877006080428563340207316247566110000
-- ----------------------------------------------------------------------

-- 1ª solución (por recursión)
-- ===========================

numeroTriangulaciones :: Integer -> Integer
numeroTriangulaciones 2 = 1
numeroTriangulaciones n = sum (zipWith (*) ts (reverse ts))
  where ts = [numeroTriangulaciones k | k <- [2..n-1]]

-- 2ª solución
-- ===========

numeroTriangulaciones2 :: Integer -> Integer
numeroTriangulaciones2 n =
  head (sucNumeroTriangulacionesInversas `genericIndex` (n-2))

--    λ> mapM_ print (take 10 sucNumeroTriangulacionesInversas)
--    [1]
--    [1,1]
--    [2,1,1]
--    [5,2,1,1]
--    [14,5,2,1,1]
--    [42,14,5,2,1,1]
--    [132,42,14,5,2,1,1]
--    [429,132,42,14,5,2,1,1]
--    [1430,429,132,42,14,5,2,1,1]
--    [4862,1430,429,132,42,14,5,2,1,1]
sucNumeroTriangulacionesInversas :: [[Integer]]
sucNumeroTriangulacionesInversas = iterate f [1]
  where f ts = sum (zipWith (*) ts (reverse ts)) : ts
```

```haskell
-- 3ª solución (con programación dinámica)
-- =======================================

numeroTriangulaciones3 :: Integer -> Integer
numeroTriangulaciones3 n = vectorTriang n ! n

--    λ> vectorTriang 9
--    array (2,9) [(2,1),(3,1),(4,2),(5,5),(6,14),(7,42),(8,132),(9,429)]
vectorTriang :: Integer -> Array Integer Integer
vectorTriang n = v
  where v = array (2,n) [(i, f i) | i <-[2..n]]
        f 2 = 1
        f i = sum [v!j*v!(i-j+1) | j <-[2..i-1]]

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

--    λ> numeroTriangulaciones 22
--    6564120420
--    (3.97 secs, 668,070,936 bytes)
--    λ> numeroTriangulaciones2 22
--    6564120420
--    (0.01 secs, 180,064 bytes)
--    λ> numeroTriangulaciones3 22
--    6564120420
--    (0.01 secs, 285,792 bytes)
--
--    λ> length (show (numeroTriangulaciones2 800))
--    476
--    (0.59 secs, 125,026,824 bytes)
--    λ> length (show (numeroTriangulaciones3 800))
--    476
--    (1.95 secs, 334,652,936 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que se verifica la siguiente
-- propiedad: el número de triangulaciones posibles de un polígono
-- convexo de n vértices es igual al (n-2)-simo número de Catalan, donde
--
--              (2n)!
```

```
--    C(n)  =  ----------
--             (n+1)! n!
-- ----------------------------------------------------------------------

propNumeroTriangulaciones :: Integer -> Property
propNumeroTriangulaciones n =
  n > 1 ==> numeroTriangulaciones2 n == numeroCatalan (n-2)

numeroCatalan :: Integer -> Integer
numeroCatalan n = factorial (2*n) `div` (factorial (n+1) * factorial n)

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

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

-- ----------------------------------------------------------------------
-- Ejercicio 4. En el siguiente gráfico se representa en una cuadrícula
-- el plano de Manhattan. Cada línea es una opción a seguir; el número
-- representa las atracciones que se pueden visitar si se elige esa
-- opción.
--
--          3         2         4         0
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |1        |0        |2        |4        |3
--     |    3    |    2    |    4    |    2    |
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |4        |6        |5        |2        |1
--     |    0    |    7    |    3    |    4    |
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |4        |4        |5        |2        |1
--     |    3    |    3    |    0    |    2    |
--     * ------- * ------- * ------- * ------- *
--     |         |         |         |         |
--     |5        |6        |8        |5        |3
```

```
--     |    1    |    3    |    2    |    2    |
--     * ------- * ------- * ------- * ------- *
--
-- El turista entra por el extremo superior izquierda y sale por el
-- extremo inferior derecha. Sólo puede moverse en las direcciones Sur y
-- Este (es decir, hacia abajo o hacia la derecha).
--
-- Representamos el mapa mediante una matriz p tal que p(i,j) = (a,b),
-- donde a = nº de atracciones si se va hacia el sur y b = nº de
-- atracciones si se va al este. Además, ponemos un 0 en el valor del
-- número de atracciones por un camino que no se puede elegir. De esta
-- forma, el mapa anterior se representa por la matriz siguiente:
--
--    ( (1,3)   (0,2)   (2,4)   (4,0)  (3,0) )
--    ( (4,3)   (6,2)   (5,4)   (2,2)  (1,0) )
--    ( (4,0)   (4,7)   (5,3)   (2,4)  (1,0) )
--    ( (5,3)   (6,3)   (8,0)   (5,2)  (3,0) )
--    ( (0,1)   (0,3)   (0,2)   (0,2)  (0,0) )
--
-- En este caso, si se hace el recorrido
--    [S, E, S, E, S, S, E, E],
-- el número de atracciones es
--     1  3  6  7  5  8  2  2
-- cuya suma es 34.
--
-- Definir la función
--    mayorNumeroV:: M.Matrix (Int,Int) -> Int
-- tal que (mayorNumeroV p) es el máximo número de atracciones que se
-- pueden visitar en el plano representado por la matriz p. Por ejemplo,
-- si se define la matriz anterior por
--    ej1b :: M.Matrix (Int,Int)
--    ej1b = M.fromLists  [[(1,3),(0,2),(2,4),(4,0),(3,0)],
--                         [(4,3),(6,2),(5,4),(2,2),(1,0)],
--                         [(4,0),(4,7),(5,3),(2,4),(1,0)],
--                         [(5,3),(6,3),(8,0),(5,2),(3,0)],
--                         [(0,1),(0,3),(0,2),(0,2),(0,0)]]
-- entonces
--    mayorNumeroV ej1b == 34
--    mayorNumeroV (M.fromLists [[(1,3),(0,0)],
--                               [(0,3),(0,0)]]) == 4
```

```
--    mayorNumeroV (M.fromLists [[(1,3),(0,2),(2,0)],
--                               [(4,3),(6,2),(5,0)],
--                               [(0,0),(0,7),(0,0)]]) == 17
-- ------------------------------------------------------------------

ej1b :: M.Matrix (Int,Int)
ej1b = M.fromLists  [[(1,3),(0,2),(2,4),(4,0),(3,0)],
                     [(4,3),(6,2),(5,4),(2,2),(1,0)],
                     [(4,0),(4,7),(5,3),(2,4),(1,0)],
                     [(5,3),(6,3),(8,0),(5,2),(3,0)],
                     [(0,1),(0,3),(0,2),(0,2),(0,0)]]


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

mayorNumeroV1 :: M.Matrix (Int,Int) -> Int
mayorNumeroV1 p = aux m n
  where m = M.nrows p
        n = M.ncols p
        aux 1 1 = 0
        aux 1 j = sum [snd (p M.!(1,k)) | k <-[1..j-1]]
        aux i 1 = sum [fst (p M.!(k,1)) | k <-[1..i-1]]
        aux i j = max (aux (i-1) j + fst (p M.!(i-1,j)))
                      (aux i (j-1) + snd (p M.!(i,j-1)))

-- 2ª solución (con programación dinámica)
-- =======================================

mayorNumeroV :: M.Matrix (Int,Int) -> Int
mayorNumeroV p = matrizNumeroV p M.! (m,n)
  where m = M.nrows p
        n = M.ncols p

matrizNumeroV :: M.Matrix (Int,Int) -> M.Matrix Int
matrizNumeroV p = q
  where m = M.nrows p
        n = M.ncols p
        q = M.matrix m n f
          where f (1,1) = 0
```

```
                f (1,j) = sum [snd (p M.!(1,k)) | k <-[1..j-1]]
                f (i,1) = sum [fst (p M.!(k,1)) | k <-[1..i-1]]
                f (i,j) = max (q M.!(i-1,j) + fst (p M.!(i-1,j)))
                              (q M.!(i,j-1) + snd (p M.!(i,j-1)))

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

--    λ> mayorNumeroVR (M.fromList 12 12 [(n,n+1) | n <- [1..]])
--    2200
--    (7.94 secs, 1,348,007,672 bytes)
--    λ> mayorNumeroV (M.fromList 12 12 [(n,n+1) | n <- [1..]])
--    2200
--    (0.01 secs, 348,336 bytes)
```

4.6. Examen 6 (12 de junio de 2018)

```
-- Informática (1º del Grado en Matemáticas, Grupos 1, 2 y 3)
-- 6º examen de evaluación continua (12 de junio de 2018)
-- ----------------------------------------------------------------------

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

import Data.Array
import Data.List
import Data.Matrix
import Test.QuickCheck

-- ----------------------------------------------------------------------
-- Ejercicio 1.1. Decimos que una lista de números enteros está reducida
-- si no tiene dos elementos consecutivos tales que uno sea el sucesor
-- del otro. Por ejemplo, la lista [-6,-8,-7,3,2,4] no está reducida
-- porque tiene dos elementos consecutivos (el -8 y el -7) tales que -7
-- es el sucesor de -8 (también el 3 y el 2).
--
-- Una forma de reducir la lista es repetir el proceso de sustituir el
-- primer par de elementos consecutivos de la lista por el mayor de los
-- dos hasta que la lista quede reducida. Por ejemplo,
```

```
--        [-6,-8,-7,3,2,4]
--    ==> [-6,   -7,3,2,4]
--    ==> [-6,      3,2,4]
--    ==> [-6,      3,  4]
--    ==> [-6,          4]
--
-- Definir la función
--    reducida :: (Num a, Ord a) => [a] -> [a]
-- tal que (reducida xs) es la lista reducida obtenida a partir de
-- xs. Por ejemplo,
--    reducida [-6,-8,-7,3,2,4]  == [-6,4]
--    reducida [4,-7,-6,4,4,3,5] == [4,-6,5]
--    reducida [3,4,5,4,3,4]     == [5]
--    reducida [1..10]           == [10]
--    reducida [1,3..15]         == [1,3,5,7,9,11,13,15]
-- ---------------------------------------------------------------------

reducida :: (Num a, Ord a) => [a] -> [a]
reducida = until esReducida reduce

-- (esReducida xs) se verifica si xs es una lista reducida. Por ejemplo,
--    esReducida [-6,-9,-7,2,0,4]  ==  True
--    esReducida [-6,-8,-7,2,3,4]  ==  False
esReducida :: (Num a, Ord a) => [a] -> Bool
esReducida xs =
  reduce xs == xs

reduce :: (Num a, Ord a) => [a] -> [a]
reduce []  = []
reduce [x] = [x]
reduce (x:y:xs) | abs (x-y) == 1 = max x y : xs
                | otherwise      = x : reduce (y:xs)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que, si n es un número
-- positivo, la reducida de la lista [1..n] es la lista [n].
-- ---------------------------------------------------------------------

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

```haskell
propReducida n =
  n > 0 ==> reducida [1..n] == [n]

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1 Para cada número n con k dígitos se define una sucesión
-- de tipo Fibonacci cuyos k primeros elementos son los dígitos de n y
-- los siguientes se obtienen sumando los k anteriores términos de la
-- sucesión. Por ejemplo, la sucesión definida por 197 es
--    1, 9, 7, 17, 33, 57, 107, 197, ...
--
-- Definir la función
--    sucFG :: Integer -> [Integer]
-- tal que (sucFG n) es la sucesión de tipo Fibonacci definida por
-- n. Por ejemplo,
--    take 10 (sucFG 197) == [1,9,7,17,33,57,107,197,361,665]
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

sucFG :: Integer -> [Integer]
sucFG k = suc
  where ks  = digitos k
        d   = length ks
        suc = ks ++ aux [drop r suc | r <- [0..d-1]]
          where aux xss = sum (map head xss) : aux (map tail xss)

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

-- 2ª solución
-- ===========

sucFG2 :: Integer -> [Integer]
sucFG2 k = init ks ++ map last (iterate f ks)
  where ks   = digitos k
```

```
        f xs = tail xs ++ [sum xs]

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

--    λ> length (show (sucFG 197 !! 60000))
--    15880
--    (2.05 secs, 475,918,520 bytes)
--    λ> length (show (sucFG2 197 !! 60000))
--    15880
--    (1.82 secs, 451,127,104 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Un número n > 9 es un número de Keith si n aparece en
-- la sucesión de tipo Fibonacci definida por n. Por ejemplo, 197 es un
-- número de Keith.
--
-- Definir la función
--    esKeith :: Integer -> Bool
-- tal que (esKeith n) se verifica si n es un número de Keith. Por
-- ejemplo,
--    esKeith 197   == True
--    esKeith 54798 == False
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Las expresiones aritméticas construidas con una
-- variable, los números enteros y las operaciones de sumar y
-- multiplicar se pueden representar mediante el tipo de datos Exp
-- definido por
--    data Exp = Var | Const Int | Sum Exp Exp | Mul Exp  Exp
--      deriving Show
--
-- Por ejemplo, la expresión 3+5x^2 se puede representar por
--    exp1 :: Exp
--    exp1 = Sum (Const 3) (Mul Var (Mul Var (Const 5)))
--
```

```haskell
-- Por su parte, los polinomios se pueden representar por la lista de
-- sus coeficientes. Por ejemplo, el polinomio 3+5x^2 se puede
-- representar por [3,0,5].
--
-- Definir la función
--    valorE :: Exp -> Int -> Int
-- tal que (valorE e n) es el valor de la expresión e cuando se
-- sustituye su variable por n. Por ejemplo,
--    λ> valorE (Sum (Const 3) (Mul Var (Mul Var (Const 5)))) 2
--    23
-- ---------------------------------------------------------------------

data Exp = Var | Const Int | Sum Exp Exp | Mul Exp  Exp
  deriving Show

valorE :: Exp -> Int -> Int
valorE Var n         = n
valorE (Const c) n   = c
valorE (Sum e1 e2) n = valorE e1 n + valorE e2 n
valorE (Mul e1 e2) n = valorE e1 n * valorE e2 n

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    expresion :: [Int] -> Exp
-- tal que (expresion p) es una expresión aritmética equivalente al
-- polinomio p. Por ejemplo,
--     λ> expresion [3,0,5]
--     Sum (Const 3) (Mul Var (Sum (Const 0) (Mul Var (Const 5))))
-- ---------------------------------------------------------------------

expresion :: [Int] -> Exp
expresion []     = Const 0
expresion [c]    = Const c
expresion (x:xs) = Sum (Const x) (Mul Var (expresion xs))

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    valorP :: [Int] -> Int -> Int
-- tal que (valorP p n) es el valor del polinomio p cuando se sustituye
-- su variable por n. Por ejemplo,
```

```haskell
--    valorP [3,0,5] 2  ==  23
-- ---------------------------------------------------------------------

-- 2ª solución
valorP :: [Int] -> Int -> Int
valorP xs n = valorE (expresion xs) n

-- 2ª solución
valorP2 :: [Int] -> Int -> Int
valorP2 = valorE . expresion

-- 3ª solución
valorP3 :: [Int] -> Int -> Int
valorP3 xs n = foldr f 0 xs
  where f x y = x + y*n


-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Dado n, consideremos la matriz cuadrada (nxn) cuyas
-- diagonales secundarias están formadas por los números 1,2..,2*n-1.
-- Por ejemplo, si n = 5,

--    ( 1 2 3 4 5 )
--    ( 2 3 4 5 6 )
--    ( 3 4 5 6 7 )
--    ( 4 5 6 7 8 )
--    ( 5 6 7 8 9 )

-- Definir la función
--    matrizDiagS :: Integer -> Matrix Integer
-- tal que (matrizDiagS n) construya dicha matriz de dimensión
-- (nxn). Por ejemplo,
--    λ> matrizDiagS 3
--    ( 1 2 3 )
--    ( 2 3 4 )
--    ( 3 4 5 )
--
--    λ> matrizDiagS 10
--    (  1  2  3  4  5  6  7  8  9 10 )
--    (  2  3  4  5  6  7  8  9 10 11 )
```

```
--    (  3  4  5  6  7  8  9 10 11 12 )
--    (  4  5  6  7  8  9 10 11 12 13 )
--    (  5  6  7  8  9 10 11 12 13 14 )
--    (  6  7  8  9 10 11 12 13 14 15 )
--    (  7  8  9 10 11 12 13 14 15 16 )
--    (  8  9 10 11 12 13 14 15 16 17 )
--    (  9 10 11 12 13 14 15 16 17 18 )
--    ( 10 11 12 13 14 15 16 17 18 19 )
-- ---------------------------------------------------------------------

matrizDiagS :: Integer -> Matrix Integer
matrizDiagS n = fromLists xss
  where xss = genericTake n (iterate (map (+1)) [1..n])

-- Con Data.Array
matrizDiagS_b :: Integer -> Array (Integer,Integer) Integer
matrizDiagS_b n =
  listArray ((1,1),(n,n)) (concat [[i .. i+n-1] | i <- [1 .. n]])

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    sumaMatrizDiagS :: Integer -> Integer
-- tal que (sumaMatrizDiagS n) es la suma de los elementos
-- (matrizDiagS n). Por ejemplo,
--    sumaMatrizDiagS 3       ==  27
--    sumaMatrizDiagS (10^3)  ==  1000000000
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

sumaMatrizDiagS1 :: Integer -> Integer
sumaMatrizDiagS1 = sum . toList . matrizDiagS

-- Con Data.Array
sumaMatrizDiagS_b :: Integer -> Integer
sumaMatrizDiagS_b = sum . elems . matrizDiagS_b

-- 2ª solución
-- ===========
```

```haskell
sumaMatrizDiagS2 :: Integer -> Integer
sumaMatrizDiagS2 = sum . matrizDiagS

-- 3ª solución
-- ===========

-- Contando cuántas veces aparece cada número

sumaMatrizDiagS3 :: Integer -> Integer
sumaMatrizDiagS3 n =
  sum (zipWith (*) [1..2*n-1] ([1..n] ++ [n-1,n-2..1]))

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

--    λ> sumaMatrizDiagS1 (10^3)
--    1000000000
--    (2.85 secs, 466,422,504 bytes)
--    λ> sumaMatrizDiagS2 (10^3)
--    1000000000
--    (2.63 secs, 418,436,656 bytes)
--    λ> sumaMatrizDiagS3 (10^3)
--    1000000000
--    (0.03 secs, 900,464 bytes)
```

4.7. Examen 7 (27 de junio de 2018)

El examen es común con el del grupo 4 (ver página 58).

4.8. Examen 8 (10 de septiembre de 2018)

El examen es común con el del grupo 4 (ver página 64).

5

Exámenes del grupo 5

Andrés Cordón y Miguel A. Martínez

5.1. Examen 1 (25 de octubre de 2017)

```
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 1º examen de evaluación continua (25 de octubre de 2017)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La carga de una lista es el número de elementos
-- estrictamente positivos menos el número de elementos estrictamente
-- negativos.
--
-- Definir la función
--    carga :: [Int] -> Int
-- tal que (carga xs) es la carga de la lista xs. Por ejemplo,
--    carga [1,0,2,0,3]    ==  3
--    carga [1,0,-2,0,3]   ==  1
--    carga [1,0,-2,0,-3]  ==  -1
--    carga [1,0,-2,2,-3]  ==  0
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
carga :: [Int] -> Int
carga xs = length [x | x <- xs, x > 0] - length [x | x <- xs, x < 0]

-- 2ª definición
carga2 :: [Int] -> Int
carga2 xs = sum [signum x | x <- xs]

-- 3ª definición
carga3 :: [Int] -> Int
carga3 [] = 0
carga3 (x:xs) = signum x + carga xs

-- 4ª definición
carga4 :: [Int] -> Int
carga4 = sum . map signum

-- Propiedad de equivalencia
prop_carga :: [Int] -> Bool
prop_carga xs =
  carga xs == carga2 xs &&
  carga xs == carga3 xs &&
  carga xs == carga4 xs

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Una lista es equilibrada si el número de elementos
-- estrictamente positivos difiere en, a lo más, una unidad del número
-- de elementos estrictamente negativos.
--
-- Definir la función
--    equilibrada :: [Int] -> Bool
-- tal que (equilibrada xs) se verifica si xs es una lista
-- equilibrada. Por ejemplo,
--    equilibrada [1,0,2,0,3]    ==  False
--    equilibrada [1,0,-2,0,3]   ==  True
--    equilibrada [1,0,-2,0,-3]  ==  True
```

```
--    equilibrada [1,0,-2,2,-3]  ==  True
-- ---------------------------------------------------------------------

equilibrada :: [Int] -> Bool
equilibrada xs = abs (carga xs) <= 1

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    triples :: Int -> [(Int,Int,Int)]
-- tal que (triples n) es la lista de todos los triples (x,y,z) con
-- 1 <= x, y, z <= n que están formados por coordenas distintas. Por
-- ejemplo,
--    ghci> triples 3
--    [(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)]
--    ghci> triples 4
--    [(1,2,3),(1,2,4),(1,3,2),(1,3,4),(1,4,2),(1,4,3),(2,1,3),(2,1,4),
--     (2,3,1),(2,3,4),(2,4,1),(2,4,3),(3,1,2),(3,1,4),(3,2,1),(3,2,4),
--     (3,4,1),(3,4,2),(4,1,2),(4,1,3),(4,2,1),(4,2,3),(4,3,1),(4,3,2)]
-- ---------------------------------------------------------------------

triples :: Int -> [(Int,Int,Int)]
triples n = [(x,y,z) | x <- [1..n]
                     , y <- [1..n]
                     , z <- [1..n]
                     , x /= y
                     , y /= z
                     , x /= z]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Los resultados de las votaciones a delegado en un
-- grupo de clase se recogen mediante listas de asociación. Por ejemplo,
--    votos :: [(String,Int)]
--    votos =  [("Ana Perez",10),("Juan Lopez",7), ("Julia Rus", 27),
--              ("Pedro Val",1), ("Pedro Ruiz",27),("Berta Gomez",11)]
--
-- Definir la función
--    mayorV :: [(String,Int)] -> Int
-- tal que (mayorV xs) es el número de votos obtenido por los ganadores
-- de la votación xs. Por ejemplo,
--    mayorV votos == 27
```

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

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

-- 1ª definición
mayorV :: [(String,Int)] -> Int
mayorV xs = maximum [j | (_,j) <- xs]

-- 2ª definición
mayorV2 :: [(String,Int)] -> Int
mayorV2 = maximum . map snd

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    ganadores :: [(String,Int)] -> [String]
-- tal que (ganadores xs) es la lista de los estudiantes con mayor
-- número de votos en xs. Por ejemplo,
--    ganadores votos == ["Julia Rus","Pedro Ruiz"]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Una lista es muy creciente si cada elemento es mayor
-- estricto que el triple del siguiente.
--
-- Definir la función
--    muyCreciente :: [Integer] -> Bool
-- tal que (muyCreciente xs) se verifica si xs es muy creciente. Por
-- ejemplo,
--    muyCreciente [1,5,23,115]  == True
--    muyCreciente [1,2,7,14]    == False
--    muyCreciente [7]           == True
--    muyCreciente []            == True
-- ---------------------------------------------------------------------
```

```haskell
muyCreciente :: [Integer] -> Bool
muyCreciente xs = and [3*x < y | (x,y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    busca :: Integer -> Integer
-- tal que (busca n) devuelve el menor número natural de n o más cifras
-- no primo cuya lista de divisores es una lista muy creciente. Por
-- ejemplo,
--    busca 2 == 25
--    busca 3 == 115
--    busca 6 == 100001
-- ---------------------------------------------------------------------

busca :: Integer -> Integer
busca n = head [i | i <- [10^(n-1)..10^n-1]
                  , muyCreciente (divisores i)
                  , not (primo i)]

divisores :: Integer -> [Integer]
divisores x = [i | i <- [1..x], rem x i == 0]

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

5.2. Examen 2 (18 de diciembre de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas), Grupo 5
-- 2º examen de evaluación continua (18 de diciembre de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número entero positivo se dirá 2-pandigital si en
```

```haskell
-- su cuadrado aparecen todos los dígitos del 0 al 9, al menos una vez.
-- Por ejemplo, 100287 es 2-pandigital porque 100287^2=10057482369.

-- Definir la lista infinita
--    dosPandigitales :: [Integer]
-- cuyos elementos son los números 2-pandigitales. Por ejemplo,
-- take 10 dosPandigitales ==
--   [32043,32286,33144,35172,35337,35757,35853,37176,37905,38772]
-- ---------------------------------------------------------------------

dosPandigitales :: [Integer]
dosPandigitales = filter esPandigital [1..]

esPandigital :: Integer -> Bool
esPandigital x = all (`elem` xs) "0123456789"
  where xs = show (x^2)

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Calcular el menor número 2-pandigital que sea primo.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head (filter isPrime dosPandigitales)
--    101723

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--   coincidencias :: Eq a => Int -> [a] -> [a] -> Bool
-- tal que (coincidencias k xs ys) se verifica si las listas xs e ys
-- coinciden en, exactamente, k de sus posiciones. Por ejemplo,
--    coincidencias 7 "salamanca" "salamandra"  ==  True
-- ---------------------------------------------------------------------

-- 1ª solución
coincidencias1 :: Eq a => Int -> [a] -> [a] -> Bool
coincidencias1 n [] _                    = n == 0
coincidencias1 n _ []                    = n == 0
coincidencias1 n (x:xs) (y:ys) | x == y    = coincidencias1 (n-1) xs ys
                               | otherwise = coincidencias1 n xs ys
```

```haskell
-- 2ª solución
coincidencias2 :: Eq a => Int -> [a] -> [a] -> Bool
coincidencias2 n xs ys =
  length (filter (\(a,b) -> a == b) (zip xs ys)) == n

-- 3ª solución
coincidencias3 :: Eq a => Int -> [a] -> [a] -> Bool
coincidencias3 n xs ys =
  length (filter (uncurry (==)) (zip xs ys)) == n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una lista de listas xss se dirá encadenada si todos sus
-- elementos son no vacíos, y el máximo de cada elemento coincide con el
-- mínimo del siguiente.
--
-- Definir la función
--    encadenada :: Ord a => [[a]] -> Bool
-- tal que (encadenada xss) se verifica si xss es encadenada. Por
-- ejemplo,
--    encadenada [[2,1],[2,5,3],[6,5]]  ==  True
--    encadenada [[2,1],[2,0,3],[6,5]]  ==  False
--    encadenada [[2,1],[],[6,5]]       ==  False
-- ---------------------------------------------------------------------

-- 1ª solución
encadenada1 :: Ord a => [[a]] -> Bool
encadenada1 xss =
     all (not . null) xss
  && and [maximum xs == minimum ys | (xs,ys) <-  zip xss (tail xss)]

-- 2ª solución
encadenada2 :: Ord a => [[a]] -> Bool
encadenada2 []   = True
encadenada2 [xs] = not (null xs)
encadenada2 (xs:ys:zss) =
      not (null xs)
   && not (null ys)
   && maximum xs == minimum ys
   && encadenada2 (ys:zss)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Representamos los árboles binarios mediante el tipo de
-- dato
--    data Arbol a = H a | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
-- Por ejemplo, el árbol
--         4
--        / \
--       /   \
--      3     7
--           / \
--          1   6
--         / \
--        0   2
-- se define de la siguiente forma
--    ejArbol :: Arbol Int
--    ejArbol = N 4 (H 3) (N 7 (N 1 (H 0) (H 2)) (H 6))
--
-- Definir la función
--    hojasProf :: Arbol a -> [a]
-- tal que (hojasProf t) es la lista de los pares formados por las hojas
-- de t junto con su profundidad. Por ejemplo,
--    hojasProf (H 7)   == [(7,0)]
--    hojasProf ejArbol == [(3,1),(0,3),(2,3),(6,2)]
-- ---------------------------------------------------------------------

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

ejArbol :: Arbol Int
ejArbol = N 4 (H 3) (N 7 (N 1 (H 0) (H 2)) (H 6))

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    hojasMasProfundas:: Arbol a -> [a]
```

```haskell
-- tal que (hojasMasProfundas t) es la lista de las hojas más profundas
-- del árbol t. Por ejemplo,
--    hojasMasProfundas (H 7)   == [7]
--    hojasMasProfundas ejArbol == [0,2]
-- ---------------------------------------------------------------------

-- 1ª solución
hojasMasProfundas :: Arbol a -> [a]
hojasMasProfundas a = [i | (i,p) <- hojasProf a
                         , p == profundidad a]

profundidad :: Arbol a -> Int
profundidad (H a)     = 0
profundidad (N a i d) = 1 + max (profundidad i) (profundidad d)

-- 2ª solución
hojasMasProfundas2 :: Arbol a -> [a]
hojasMasProfundas2 a = [h | (h,p) <- hps
                          , p == m]
  where hps = hojasProf a
        m   = maximum [p | (_,p) <- hps]
```

5.3. Examen 3 (30 de enero de 2018)

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

5.4. Examen 4 (21 de marzo de 2018)

```haskell
-- Informática (1º del Grado en Matemáticas), Grupo 5
-- 4º examen de evaluación continua (21 de marzo de 2018)
-- ---------------------------------------------------------------------

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

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Se considera una enumeración de los números primos:
--     p(1)=2,  p(2)=3, p(3)=5, p(4)=7, p(5)=11, p(6)=13, p(7)=17,...
-- Dado un entero positivo x, definimos:
-- + longitud de x como el mayor i tal que el primo p(i) aparece en la
--   factorización en números primos de x
-- + altura de x el mayor exponente n que aparece en la factorización
--   en números primos de x
--
-- Por ejemplo, 3500 tiene longitud 4 y altura 3, pues 3500=2^2*5^3*7^1;
-- y 34 tiene longitud 7 y altura 1, pues 34 = 2*17.

-- Definir las funciones
--    longitud :: Integer -> Integer
--    altura   :: Integer -> Integer
-- que calculan la longitud y la altura, respectivamente, de un entero
-- positivo. Por ejemplo,
--    longitud 3500  ==  4
--    altura 3500    ==  3

longitud :: Integer -> Integer
longitud 1 = 0
longitud x = genericLength (takeWhile (<= last (primeFactors x)) primes)

altura :: Integer -> Integer
altura 1 = 0
altura x = maximum  (map genericLength (group (primeFactors x)))

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Diremos que dos enteros positivos a y b están
-- relacionados si tienen la misma longitud y la misma altura.
--
-- Definir la lista infinita
--    paresRel :: [(Integer,Integer)]
-- que enumera todos los pares (a,b), con 1<=a<b, tales que a y b están
-- relacionados. Por ejemplo,
--    λ> take 9 paresRel
--    [(3,6),(5,10),(9,12),(7,14),(5,15),(10,15),(9,18),(12,18),(7,21)]
-- ---------------------------------------------------------------------
```

```haskell
paresRel :: [(Integer,Integer)]
paresRel =
  [(a,b) | b <- [1..]
         , a <- [1..b-1]
         , longitud a == longitud b
         , altura a == altura b]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Calcular en qué posición aparece el par (31,310) en la
-- lista infinta ParesRel
-- ---------------------------------------------------------------------

-- El cálculo es
--    λ> 1 + genericLength (takeWhile (/= (31,310)) paresRel)
--    712

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los puntos del plano como pares de números
-- reales
--    type Punto = (Float,Float)
-- y un camino de k pasos como una lista de k+1 puntos,
--    xs = [x(0),x(1),....,x(k)]
-- Por ejemplo, consideramos el siguiente camino de 4 pasos:
--    camino :: [Punto]
--    camino = [(0,0),(2,2),(5,2),(5,-1),(0,-1)]
--
-- Definir la función
--    mediaPasos :: [Punto] -> Double
-- tal que (mediaPasos xs) es la media aritmética de las longitudes de
-- los pasos que conforman el camino xs. Por ejemplo
--    mediaPasos camino  ==  3.4571068
-- ---------------------------------------------------------------------

type Punto =(Float,Float)

camino :: [Punto]
camino = [(0,0),(2,2),(5,2),(5,-1),(0,-1)]

mediaPasos :: [Punto] -> Float
```

```haskell
mediaPasos xs =
  sum [distancia p q | (p,q) <- zip xs (tail xs)] / k
  where k = genericLength xs - 1

distancia :: Punto -> Punto -> Float
distancia (x1,y1) (x2,y2) = sqrt ((x1-x2)^2+(y1-y2)^2)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Representamos los árboles binarios mediante el tipo de
-- dato
--    data Arbol a = H a | N a (Arbol a) (Arbol a) deriving Show
--
-- Una forma de ampliar un árbol binario es añadiendo un nuevo nivel donde
-- las nuevas hojas sean iguales a la suma de los valores de los nodos
-- desde el padre hasta llegar a la raíz (inclusives). Por ejemplo:
--       5               5       |         3                3
--      / \             / \      |        / \              /   \
--     2   0    ==>    2   0     |       2  -9    ==>     2    -9
--                    / \ / \    |                      / \   / \
--                   7  7 5  5   |                     5   5 -6 -6
--                                                    /\   /\
--                                                   6  6 5  5
--
-- Definir la función
--    ampliaArbol:: Num a => Arbol a -> Arbol a
-- tal que (ampliaArbol a) es el árbol a ampliado en un nivel. Por
-- ejemplo,
--    λ> ampliaArbol (N 5 (H 2)(H 0))
--    N 5 (N 2 (H 7) (H 7)) (N 0 (H 5) (H 5))
--    λ> ampliaArbol (H 1)
--    N 1 (H 1) (H 1)
--    λ> ampliaArbol N 1 (H 1) (H 1)
--    N 1 (N 1 (H 2) (H 2)) (N 1 (H 2) (H 2))
-- ---------------------------------------------------------------------

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

ampliaArbol :: Num a => Arbol a -> Arbol a
```

```haskell
ampliaArbol a = aux 0 a
  where aux n (H x)     = N x (H (n+x)) (H (n+x))
        aux n (N x i d) = N x (aux (n+x) i) (aux (n+x) d)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Representamos las matrices mediante la librería
-- Data.Matrix.
--
-- Definir la función
--    ampliada :: Num a => Matrix a -> Matrix a
-- tal que (ampliada p) devuelve la matriz obtenida a partir de p
-- al ampliar cada fila con la suma de sus elementos. Por ejemplo,
--   λ> ampliada (fromLists [[1,2,3],[1,-1,0],[2,2,5]])
--   (  1  2  3  6 )
--   (  1 -1  0  0 )
--   (  2  2  5  9 )
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    bordeNulo :: Num a => Matrix a -> Matrix a
-- tal que (bordeNulo p) se verifica si todos los elementos de los
-- bordes de la matriz p son nulos. Por ejemplo.
--    λ> bordeNulo (fromLists [[0,0,0,0],[0,3,1,0],[0,0,0,0]])
--    True
--    λ> bordeNulo (fromLists [[0,0,0,0],[0,3,1,0],[0,0,0,-1]])
--    False
-- ---------------------------------------------------------------------

bordeNulo :: (Num a,Eq a) => Matrix a -> Bool
bordeNulo p = all (==0) ([p!(1,j) | j <- [1..m]] ++
                         [p!(n,j) | j <- [1..m]] ++
                         [p!(i,1) | i <- [2..n-1]] ++
```

```
                        [p!(i,m) | i <- [2..n-1]])
  where n = nrows p
        m = ncols p
```

5.5. Examen 5 (7 de mayo de 2018)

```
-- Informática (1º del Grado en Matemáticas) Grupo 5
-- 5º examen de evaluación continua (7 de mayo de 2018)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Matrix
import I1M.Pila
import I1M.PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número natural x es un cuadrado perfecto si x = b^2
-- para algún natural b. Por ejemplo, 25 es un cuadrado perfecto y 24 no
-- lo es. Un número entero positivo se dirá regular si sus cifras están
-- ordenadas, ya sea en orden creciente o en orden decreciente. Por
-- ejemplo, 11468 y 974000 son regulares y 16832 no lo es.
--
-- Definir la lista infinita
--    regularesPerfectos :: [Integer]
-- cuyos elementos son los cuadrados perfectos que son regulares. Por
-- ejemplo,
--    λ> take 19 regularesPerfectos
--    [1,4,9,16,25,36,49,64,81,100,144,169,225,256,289,400,441,841,900]
-- ---------------------------------------------------------------------

regularesPerfectos :: [Integer]
regularesPerfectos = filter regular [i^2 | i <- [1..]]

regular :: Integer -> Bool
regular x = cs == ds || cs == reverse ds
  where cs = cifras x
```

```
        ds = sort cs

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcula el mayor cuadrado perfecto regular de 7 cifras.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    posicionesPares :: Pila a -> Pila a
-- tal que (posicionesPares p) devuelve la pila obtenida tomando los elementos
-- que ocupan las posiciones pares en la pila p. Por ejemplo,
--    λ> posicionesPares (foldr apila vacia [0..9])
--    1|3|5|7|9|-
--    λ> posicionesPares (foldr apila vacia "salamanca")
--    'a'|'a'|'a'|'c'|-
-- ---------------------------------------------------------------------

posicionesPares :: Pila a -> Pila a
posicionesPares p
  | esVacia p = vacia
  | esVacia q = vacia
  | otherwise = apila (cima q) (posicionesPares r)
  where q = desapila p
        r = desapila q

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una matriz de enteros p se dirá segura para la torre si
--    + p solo contiene ceros y unos;
--    + toda fila de p contiene, a lo más, un 1;
--    + toda columna de p contiene, a lo más, un 1.
--
--  Definir la función
--     seguraTorre :: Matrix Int -> Bool
```

```haskell
--  tal que (seguraTorre p) se verifica si la matriz p es segura para
--  la torre según la definición anterior. Por ejemplo,
--     λ> seguraTorre (fromLists [[0,0,1,0,0],[1,0,0,0,0],[0,0,0,0,1]])
--     True
--     λ> seguraTorre (fromLists [[0,0,0,0],[0,1,0,0],[0,0,0,1],[0,1,1,0]])
--     False
--     λ> seguraTorre (fromLists [[0,1],[2,0]])
--     False
-- ---------------------------------------------------------------------

seguraTorre :: Matrix Int -> Bool
seguraTorre p =
  all (`elem` [0,1]) (toList p) &&
  all tieneComoMaximoUnUno filas &&
  all tieneComoMaximoUnUno columnas
  where m       = nrows p
        n       = ncols 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
--    trunca :: (Eq a,Num a) => Int -> Polinomio a -> Polinomio a
-- tal que (trunca k p) es el polinomio formado por aquellos términos de
-- p de grado mayor o igual que k. Por ejemplo,
--    λ> trunca 3 (consPol 5 2 (consPol 3 (-7) (consPol 2 1 polCero)))
--    2*x^5 + -7*x^3
--    λ> trunca 2 (consPol 5 2 (consPol 3 (-7) (consPol 2 1 polCero)))
--    2*x^5 + -7*x^3 + x^2
--    λ> trunca 4 (consPol 5 2 (consPol 3 (-7) (consPol 2 1 polCero)))
--    2*x^5
-- ---------------------------------------------------------------------

trunca :: (Eq a, Num a) => Int -> Polinomio a -> Polinomio a
trunca k p
  | k < 0       = polCero
  | esPolCero p = polCero
```

```
  | k > n       = polCero
  | otherwise   = consPol n a (trunca 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 definición anterior. Por ejemplo,
--    λ> imparPol (consPol 5 2 (consPol 3 6 (consPol 0 3 polCero)))
--    True
--    λ> imparPol (consPol 5 2 (consPol 3 6 (consPol 0 4 polCero)))
--    False
--    λ> 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
-- ===========
```

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

5.6. Examen 6 (12 de junio de 2018)

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

5.7. Examen 7 (27 de junio de 2018)

El examen es común con el del grupo 1 (ver página 58).

5.8. Examen 8 (10 de septiembre de 2018)

El examen es común con el del grupo 1 (ver página 64).

Apéndice A

Resumen de funciones predefinidas de Haskell

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

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

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

A.1. Resumen de funciones sobre TAD en Haskell

A.1.1. Polinomios

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

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

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

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

A.1.3. Tablas

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

A.1.4. Grafos

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

Apéndice B

Método de Pólya para la resolución de problemas

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

Para resolver un problema se necesita:

Paso 1: Entender el problema

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

Paso 2: Configurar un plan

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

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

Paso 3: Ejecutar el plan

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

Paso 4: Examinar la solución obtenida

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

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

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

Para resolver un problema se necesita:

Paso 1: Entender el problema

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

Paso 2: Diseñar el programa

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

Paso 3: Escribir el programa

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

Paso 4: Examinar la solución obtenida

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

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

Bibliografía

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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