Exámenes de “Programación funcional con Haskell”

Vol. 8 (Curso 2016–17)

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 2017

Índice general

Introducción

Este libro es una recopilación de las soluciones de ejercicios de los exámenes de programación funcional con Haskell de la asignatura de Informática (curso 2016–17) 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 2016–17) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2016–17) [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 8º 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-16/temas/2016-17-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-16/ejercicios/ejercicios-I1M-2016.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]

José A. Alonso
Sevilla, 20 de diciembre de 2017

[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

1

Exámenes del grupo 1

José A. Alonso

1.1. Examen 1 (26 de octubre de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (26 de octubre de 2016)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    ceros :: Int -> Int
-- tal que (ceros n) es el número de ceros en los que termina el número
-- n. Por ejemplo,
--    ceros 30500  ==  2
--    ceros 30501  ==  0
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
ceros :: Int -> Int
ceros n | n `rem` 10 == 0 = 1 + ceros (n `div`10)
        | otherwise       = 0

-- 2ª definición (por comprensión):
ceros2 :: Int -> Int
ceros2 0 = 1
ceros2 n = head [x | x <- [0..]
                   , n `rem` (10^x) /= 0] - 1
```

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

enBase :: Int -> [Int] -> Int
enBase b xs = sum [y*b^n | (y,n) <- zip xs [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    repetidos :: Eq a => [a] -> [a]
-- tal que (repetidos xs) es la lista de los elementos repetidos de
-- xs. Por ejemplo,
--    repetidos [1,3,2,1,2,3,4] == [1,3,2]
--    repetidos [1,2,3]         == []
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. [Problema 37 del proyecto Euler] Un número
-- primo es truncable si los números que se obtienen eliminado cifras,
-- de derecha a izquierda, son primos. Por ejemplo, 599 es un primo
-- truncable porque 599, 59 y 5 son primos; en cambio, 577 es un primo
-- no truncable porque 57 no es primo.
--
```

```haskell
-- Definir la función
--    primoTruncable :: Int -> Bool
-- tal que (primoTruncable x) se verifica si x es un primo
-- truncable. Por ejemplo,
--    primoTruncable 599  ==  True
--    primoTruncable 577  ==  False
-- ---------------------------------------------------------------------

primoTruncable :: Int -> Bool
primoTruncable x
  | x < 10    = primo x
  | otherwise = primo x && primoTruncable (x `div` 10)

-- (primo x) se verifica si x es primo.
primo :: Int -> Bool
primo x = factores x == [1,x]

-- (factores x) es la lista de los factores de x.
factores :: Int -> [Int]
factores x = [y | y <- [1..x]
                , x `rem` y == 0]
```

1.2. Examen 2 (29 de noviembre de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 2º examen de evaluación continua (29 de noviembre de 2016)
-- ---------------------------------------------------------------------

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

import Debug.Trace

import Data.List
import Data.Numbers.Primes

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2 puntos] La persistencia multiplicativa de un número
-- es la cantidad de pasos requeridos para reducirlo a una dígito
```

```haskell
-- multiplicando sus dígitos. Por ejemplo, la persistencia de 39 es 3
-- porque 3*9 = 27, 2*7 = 14 y 1*4 = 4.
--
-- Definir la función
--    persistencia     :: Integer -> Integer
-- tale que (persistencia x) es la persistencia de x. Por ejemplo,
--      persistencia 39                                 ==   3
--      persistencia 2677889                            ==   8
--      persistencia 26888999                           ==   9
--      persistencia 3778888999                         ==  10
--      persistencia 277777788888899                    ==  11
--      persistencia 77777733332222222222222222222      ==  11
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Definir la función
--    intercala :: [a] -> [a] -> [[a]]
-- tal que (intercala xs ys) es la lista obtenida intercalando los
-- elementos de ys entre los de xs. Por ejemplo,
--    ghci> intercala "79" "15"
--    ["719","759"]
--    ghci> intercala "79" "154"
--    ["719","759","749"]
--    ghci> intercala "796" "15"
--    ["71916","71956","75916","75956"]
--    ghci> intercala "796" "154"
--    ["71916","71956","71946",
--     "75916","75956","75946",
--     "74916","74956","74946"]
```

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

intercala :: [a] -> [a] -> [[a]]
intercala []     _  = []
intercala [x]    _  = [[x]]
intercala (x:xs) ys = [x:y:zs | y <- ys
                              , zs <- intercala xs ys]

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2 puntos] Un número primo se dice que es un primo de
-- Kamenetsky  si al anteponerlo cualquier dígito se obtiene un número
-- compuesto. Por ejemplo, el 5 es un primo de Kamenetsky ya que 15, 25,
-- 35, 45, 55, 65, 75, 85 y 95 son compuestos. También lo es 149 ya que
-- 1149, 2149, 3149, 4149, 5149, 6149, 7149, 8149 y 9149 son compuestos.
--
-- Definir la sucesión
--    primosKamenetsky :: [Integer]
-- tal que sus elementos son los números primos de Kamenetsky. Por
-- ejemplo,
--    take 5 primosKamenetsky  ==  [2,5,149,401,509]
-- ---------------------------------------------------------------------

primosKamenetsky :: [Integer]
primosKamenetsky =
  [x | x <- primes
     , esKamenetsky x]

esKamenetsky :: Integer -> Bool
esKamenetsky x =
  all (not . isPrime) [read (d:xs) | d <- "123456789"]
  where xs = show x

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2 puntos] Representamos los árboles binarios con
-- elementos en las hojas y en los nodos mediante el tipo de dato
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo,
--    ej1 :: Arbol Int
```

```haskell
--    ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))
--
-- Definir la función
--    ramasCon :: Eq a => Arbol a -> a -> [[a]]
-- tal que (ramasCon a x) es la lista de las ramas del árbol a en las
-- que aparece el elemento x. Por ejemplo,
--   ramasCon ej1 2 ==  [[5,2,1],[5,2,2],[5,3,2]]
-- ---------------------------------------------------------------------

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

ej1 :: Arbol Int
ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))

ramasCon :: Eq a => Arbol a -> a -> [[a]]
ramasCon a x = [ys | ys <- ramas a, x `elem` ys]

ramas :: Arbol a -> [[a]]
ramas (H x)     = [[x]]
ramas (N x i d) = [x:ys | ys <- ramas i ++ ramas d]

-- ---------------------------------------------------------------------
-- Ejercicio 5. [2 puntos] El valor máximo de la suma de elementos
-- consecutivos de la lista [3,-4,4,1] es 5 obtenido sumando los
-- elementos del segmento [4,1] y para la lista [-2,1,4,-3,5,-7,6] es 7
-- obtenido sumando los elementos del segmento [1,4,-3,5].
--
-- Definir la función
--    sumaMaxima :: [Integer] -> Integer
-- tal que (sumaMaxima xs) es el valor máximo de la suma de elementos
-- consecutivos de la lista xs. Por ejemplo,
--    sumaMaxima1 [3,-4,4,1]          ==  5
--    sumaMaxima1 [-2,1,4,-3,5,-7,6]  ==  7
--    sumaMaxima []                   ==  0
--    sumaMaxima [2,-2,3,-3,4]        ==  4
--    sumaMaxima [-1,-2,-3]           ==  0
--    sumaMaxima [2,-1,3,-2,3]        ==  5
--    sumaMaxima [1,-1,3,-2,4]        ==  5
```

```haskell
--    sumaMaxima [2,-1,3,-2,4]        ==  6
--    sumaMaxima [1..10^6]            ==  500000500000
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

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

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

-- 2ª definición
-- =============

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

sumaMaximaAux :: Integer -> Integer -> [Integer] -> Integer
sumaMaximaAux m v xs | trace ("aux " ++ show m ++ " " ++ show v ++ " " ++ show xs
sumaMaximaAux m v [] = max m v
sumaMaximaAux m v (x:xs)
    | x >= 0    = sumaMaximaAux m (v+x) xs
    | v+x > 0   = sumaMaximaAux (max m v) (v+x) xs
    | otherwise = sumaMaximaAux (max m v) 0 xs

-- 3ª definición
-- =============

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

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

-- 4ª definición
-- =============

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

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

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

1.3. Examen 3 (31 de enero de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupos 2 y 4)
-- 3º examen de evaluación continua (31 de enero de 2017)
-- ---------------------------------------------------------------------

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

```haskell
import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. (2.5 puntos) Se observa que en la cadena "aabbccddeffgg"
-- todos los caracteres están duplicados excepto el 'e'. Al añadirlo
-- obtenemos la cadena "aabbccddeeffgg" y se dice que esta última está
-- duplicada.
--
-- También se observa que la cadena "aaaabbbccccdd" no está duplicada
-- (porque hay un número impar de 'b' consecutivas). Añadiendo una 'b'
-- se obtiene la cadena "aaaabbbbccccdd" que sí está duplicada.
--
-- Definir la función
--    duplica :: Eq a => [a] -> [a]
-- tal que (duplica xs) es la lista obtenida duplicando los elementos de
-- xs que no lo están. Por ejemplo,
--    duplica "b"        ==  "bb"
--    duplica "abba"     ==  "aabbaa"
--    duplica "Betis"    ==  "BBeettiiss"
--    duplica [1,1,1]    ==  [1,1,1,1]
--    duplica [1,1,1,1]  ==  [1,1,1,1]
-- ---------------------------------------------------------------------

-- 1ª definición
duplica :: Eq a => [a] -> [a]
duplica []       = []
duplica [x]      = [x,x]
duplica (x:y:zs) | x == y    = x : x : duplica zs
                 | otherwise = x : x : duplica (y:zs)

-- 2ª definición
duplica2 :: Eq a => [a] -> [a]
duplica2 xs = concatMap dupl (group xs)
  where dupl ys@(y:_) | (even.length) ys = ys
                      | otherwise        = y : ys

-- ---------------------------------------------------------------------
-- Ejercicio 2. (2.5 puntos) Las matrices se pueden representar mediante
-- listas de listas. Por ejemplo, la matriz
```

```
--    |1 2 5|
--    |3 0 7|
--    |9 1 6|
--    |6 4 2|
-- se puede representar por
--    ej :: [[Int]]
--    ej = [[1,2,5],
--          [3,0,7],
--          [9,1,6],
--          [6,4,2]]
--
-- Definir la función
--    ordenaPor :: Ord a => [[a]] -> Int -> [[a]]
-- tal que (ordenaPor xss k) es la matriz obtenida ordenando la matriz
-- xss por los elementos de la columna k. Por ejemplo,
--    ordenaPor ej 0  ==  [[1,2,5],[3,0,7],[6,4,2],[9,1,6]]
--    ordenaPor ej 1  ==  [[3,0,7],[9,1,6],[1,2,5],[6,4,2]]
--    ordenaPor ej 2  ==  [[6,4,2],[1,2,5],[9,1,6],[3,0,7]]
-- ---------------------------------------------------------------------

ej :: [[Int]]
ej = [[1,2,5],
      [3,0,7],
      [9,1,6],
      [6,4,2]]

-- 1ª definición
ordenaPor :: Ord a => [[a]] -> Int -> [[a]]
ordenaPor xss k =
  map snd (sort [(xs!!k,xs) | xs <- xss])

-- 2ª definición
ordenaPor2 :: Ord a => [[a]] -> Int -> [[a]]
ordenaPor2 xss k =
  map snd (sort (map (\xs -> (xs!!k, xs)) xss))

-- 4ª definición
ordenaPor4 :: Ord a => [[a]] -> Int -> [[a]]
ordenaPor4 xss k =
  map snd (sort (zip (map (!! k) xss) xss))
```

```haskell
-- 5ª definición
ordenaPor5 :: Ord a => [[a]] -> Int -> [[a]]
ordenaPor5 xss k = sortBy comp xss
  where comp ys zs = compare (ys!!k) (zs!!k)

-- ---------------------------------------------------------------------
-- Ejercicio 3. (2.5 puntos) Definir la función
--    sumas3Capicuas  :: Integer -> [(Integer, Integer, Integer)]
-- tal que (sumas3Capicuas n) es la lista de las descomposiciones del
-- número natural n como suma de tres números capicúas (con los sumandos
-- no decrecientes). Por ejemplo,
--    sumas3Capicuas 0             ==  [(0,0,0)]
--    sumas3Capicuas 1             ==  [(0,0,1)]
--    sumas3Capicuas 2             ==  [(0,0,2),(0,1,1)]
--    sumas3Capicuas 3             ==  [(0,0,3),(0,1,2),(1,1,1)]
--    sumas3Capicuas 4             ==  [(0,0,4),(0,1,3),(0,2,2),(1,1,2)]
--    length (sumas3Capicuas 17)   ==  17
--    length (sumas3Capicuas 2017) ==  47
--
-- Comprobar con QuickCheck que todo número natural se puede escribir
-- como suma de tres capicúas.
-- ---------------------------------------------------------------------

sumas3Capicuas :: Integer -> [(Integer, Integer, Integer)]
sumas3Capicuas x =
  [(a,b,c) | a <- as
           , b <- dropWhile (< a) as
           , let c = x - a - b
           , b <= c
           , esCapicua c]
  where as = takeWhile (<= x) capicuas

-- capicuas es la sucesión de los números capicúas. Por ejemplo,
--    ghci> take 45 capicuas
--    [0,1,2,3,4,5,6,7,8,9,11,22,33,44,55,66,77,88,99,101,111,121,131,
--     141,151,161,171,181,191,202,212,222,232,242,252,262,272,282,292,
--     303,313,323,333,343,353]
capicuas :: [Integer]
capicuas = capicuas1
```

```haskell
-- 1ª definición de capicuas
-- =========================

capicuas1 :: [Integer]
capicuas1 = [n | n <- [0..]
               , esCapicua n]

-- (esCapicua x) se verifica si x es capicúa. Por ejemplo,
--    esCapicua 353   ==  True
--    esCapicua 3553  ==  True
--    esCapicua 3535  ==  False
esCapicua :: Integer -> Bool
esCapicua x =
  xs == reverse xs
  where xs = show x


-- 2ª definición de capicuas
-- =========================

capicuas2 :: [Integer]
capicuas2 = capicuasImpares `mezcla` capicuasPares

-- capicuasPares es la sucesión del cero y las capicúas con un número
-- par de dígitos. Por ejemplo,
--    ghci> take 17 capicuasPares
--    [0,11,22,33,44,55,66,77,88,99,1001,1111,1221,1331,1441,1551,1661]
capicuasPares :: [Integer]
capicuasPares =
  [read (ns ++ reverse ns) | n <- [0..]
                           , let ns = show n]

-- capicuasImpares es la sucesión de las capicúas con un número
-- impar de dígitos a partir de 1. Por ejemplo,
--    ghci> take 20 capicuasImpares
--    [1,2,3,4,5,6,7,8,9,101,111,121,131,141,151,161,171,181,191,202]
capicuasImpares :: [Integer]
capicuasImpares =
  [1..9] ++ [read (ns ++ [z] ++ reverse ns)
```

```
            | n <- [1..]
            , let ns = show n
            , z <- "0123456789"]

-- (mezcla xs ys) es la lista ordenada obtenida mezclando las dos listas
-- ordenadas xs e ys, suponiendo que ambas son infinitas y con elementos
-- distintos. Por ejemplo,
--    take 10 (mezcla [2,12..] [5,15..])  ==  [2,5,12,15,22,25,32,35,42,45]
--    take 10 (mezcla [2,22..] [5,15..])  ==  [2,5,15,22,25,35,42,45,55,62]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla us@(x:xs) vs@(y:ys)
  | x < y     = x : mezcla xs vs
  | otherwise = y : mezcla us ys

-- 3ª definición de capicuas
-- =========================

capicuas3 :: [Integer]
capicuas3 = iterate sigCapicua 0

-- (sigCapicua x) es el capicúa siguiente del número x. Por ejemplo,
--    sigCapicua 12321           == 12421
--    sigCapicua 1298921         == 1299921
--    sigCapicua 999             == 1001
--    sigCapicua 9999            == 10001
--    sigCapicua 898             == 909
--    sigCapicua 123456777654321 == 123456787654321
sigCapicua :: Integer -> Integer
sigCapicua n = read cs
    where l  = length (show (n+1))
          k  = l `div` 2
          xs = show ((n `div` (10^k)) + 1)
          cs = xs ++ drop (l `rem` 2) (reverse xs)

-- 4ª definición de capicuas
-- =========================

capicuas4 :: [Integer]
capicuas4 =
  concatMap generaCapicuas4 [1..]
```

```haskell
generaCapicuas4 :: Integer -> [Integer]
generaCapicuas4 1 = [0..9]
generaCapicuas4 n
  | even n    = [read (xs ++ reverse xs)
                | xs <- map show [10^(m-1)..10^m-1]]
  | otherwise = [read (xs ++ (y : reverse xs))
                | xs <- map show [10^(m-1)..10^m-1]
                , y <- "0123456789"]
  where m = n `div` 2

-- 5ª definición de capicuas
-- ========================

capicuas5 :: [Integer]
capicuas5 = 0 : aux 1
  where aux n =    [read (show x ++ tail (reverse (show x)))
                   | x <- [10^(n-1)..10^n-1]]
                ++ [read (show x ++ reverse (show x))
                   | x <- [10^(n-1)..10^n-1]]
                ++ aux (n+1)

-- 6ª definición de capicuas
-- ========================

capicuas6 :: [Integer]
capicuas6 = 0 : map read (capicuas6Aux [1..9])

capicuas6Aux :: [Integer] -> [String]
capicuas6Aux xs =  map duplica1 xs'
                ++ map duplica2 xs'
                ++ capicuas6Aux [head xs * 10 .. last xs * 10 + 9]
  where
    xs'         = map show xs
    duplica1 cs = cs ++ tail (reverse cs)
    duplica2 cs = cs ++ reverse cs

-- 7ª definición de capicuas
-- ========================
```

```haskell
capicuas7 :: [Integer]
capicuas7 = 0 : map read (capicuas7Aux [1..9])

capicuas7Aux :: [Integer] -> [String]
capicuas7Aux xs =  map duplica1 xs'
                ++ map duplica2 xs'
                ++ capicuas7Aux [head xs * 10 .. last xs * 10 + 9]
  where
    xs'      = map show xs
    duplica1 = (++) <$> id <*> tail . reverse
    duplica2 = (++) <$> id <*> reverse

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_capicuas :: (Positive Int) -> Bool
prop_capicuas (Positive k) =
  all (== capicuas1 !! k) [f !! k | f <- [ capicuas2
                                         , capicuas3
                                         , capicuas4
                                         , capicuas5
                                         , capicuas6
                                         , capicuas7]]

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

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

--    ghci> capicuas1 !! 2000
--    1001001
--    (2.25 secs, 598,879,552 bytes)
--    ghci> capicuas2 !! 2000
--    1001001
--    (0.05 secs, 28,630,552 bytes)
--    ghci> capicuas3 !! 2000
--    1001001
```

```
--    (0.06 secs, 14,721,360 bytes)
--    ghci> capicuas4 !! 2000
--    1001001
--    (0.01 secs, 0 bytes)
--    ghci> capicuas5 !! 2000
--    1001001
--    (0.01 secs, 0 bytes)
--    ghci> capicuas6 !! 2000
--    1001001
--    (0.01 secs, 0 bytes)
--    ghci> capicuas7 !! 2000
--    1001001
--    (0.01 secs, 0 bytes)
--
--    ghci> capicuas2 !! (10^5)
--    900010009
--    (2.03 secs, 1,190,503,952 bytes)
--    ghci> capicuas3 !! (10^5)
--    900010009
--    (5.12 secs, 1,408,876,328 bytes)
--    ghci> capicuas4 !! (10^5)
--    900010009
--    (0.21 secs, 8,249,296 bytes)
--    ghci> capicuas5 !! (10^5)
--    900010009
--    (0.10 secs, 31,134,176 bytes)
--    ghci> capicuas6 !! (10^5)
--    900010009
--    (0.14 secs, 55,211,272 bytes)
--    ghci> capicuas7 !! (10^5)
--    900010009
--    (0.03 secs, 0 bytes)

-- La propiedad es
prop_sumas3Capicuas :: Integer -> Property
prop_sumas3Capicuas x =
  x >= 0 ==> not (null (sumas3Capicuas x))

-- La comprobación es
--    ghci> quickCheck prop_sumas3Capicuas
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. (2.5 puntos) 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
--     / \       / \           / \
--    8   3     8   3         8   3
--        |       /|\       /|\  |
--        4      4 5 6     4 5 6 7
-- se pueden representar 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
--    nodos :: Int -> Arbol a -> [a]
-- tal que (nodos k t) es la lista de los nodos del árbol t que tienen k
-- sucesores. Por ejemplo,
--    nodos 0 ej1  ==  [8,4]
--    nodos 1 ej1  ==  [3]
--    nodos 2 ej1  ==  [1]
--    nodos 3 ej1  ==  []
--    nodos 3 ej2  ==  [3]
-- ---------------------------------------------------------------------

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 []]]

-- 1ª definición
nodos :: Int -> Arbol a -> [a]
```

```haskell
nodos k (N x ys)
  | k == length ys = x : concatMap (nodos k) ys
  | otherwise      = concatMap (nodos k) ys

-- 2ª definición
nodos2 :: Int -> Arbol a -> [a]
nodos2 k (N x ys)
  | k == length ys = x : zs
  | otherwise      = zs
  where zs = concat [nodos2 k y | y <- ys]

-- 3ª definición
nodos3 :: Int -> Arbol a -> [a]
nodos3 k (N x ys)
  = [x | k == length ys]
    ++ concat [nodos3 k y | y <- ys]
```

1.4. Examen 4 (22 de marzo de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (22 de marzo de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Matrix

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    reducida :: Eq a => [a] -> [a]
-- tal que (reducida xs) es la lista obtenida a partir de xs de forma
-- que si hay dos o más elementos idénticos consecutivos, borra las
-- repeticiones y deja sólo el primer elemento. Por ejemplo,
--    ghci> reducida "eesssooo essss   toodddooo"
--    "eso es todo"
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución (por recursión):
reducida1 :: Eq a => [a] -> [a]
reducida1 []     = []
reducida1 (x:xs) = x : reducida1 (dropWhile (==x) xs)

-- 2ª solución (por comprensión):
reducida2 :: Eq a => [a] -> [a]
reducida2 xs = [x | (x:_) <- group xs]

-- 3ª solución (sin argumentos):
reducida3 :: Eq a => [a] -> [a]
reducida3 = map head . group

-- ---------------------------------------------------------------------
-- Ejercicio 2. La codificación de Luka consiste en añadir detrás de
-- cada vocal la letra 'p' seguida de la vocal. Por ejemplo, la palabra
-- "elena" se codifica como "epelepenapa" y "luisa" por "lupuipisapa".
--
-- Definir la función
--    codifica   :: String -> String
-- tal que (codifica cs) es la codificación de Luka de la cadena cs. Por
-- ejemplo,
--    ghci> codifica "elena admira a luisa"
--    "epelepenapa apadmipirapa apa lupuipisapa"
--    ghci> codifica "todo para nada"
--    "topodopo paparapa napadapa"
-- ---------------------------------------------------------------------

codifica :: String -> String
codifica "" = ""
codifica (c:cs) | esVocal c = c : 'p' : c : codifica cs
                | otherwise = c : codifica cs

esVocal :: Char -> Bool
esVocal c = c `elem` "aeiou"

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un elemento de una matriz es un máximo local si es mayor
-- que todos sus vecinos. Por ejemplo, sea ejM la matriz definida por
```

```
--    ejM :: Matrix Int
--    ejM = fromLists [[1,0,0,8],
--                     [0,2,0,3],
--                     [0,0,0,5],
--                     [3,5,7,6],
--                     [1,2,3,4]]
-- Los máximos locales de ejM son 8 (en la posición (1,4)), 2 (en la
-- posición (2,2)) y 7 (en la posición (4,3)).
--
-- Definir la función
--    maximosLocales :: Matrix Int -> [((Int,Int),Int)]
-- tal que (maximosLocales p) es la lista de las posiciones en las que
-- hay un máximo local, con el valor correspondiente. Por ejemplo,
--    maximosLocales ejM == [((1,4),8),((2,2),2),((4,3),7)]
-- ---------------------------------------------------------------------

ejM :: Matrix Int
ejM = fromLists [[1,0,0,8],
                 [0,2,0,3],
                 [0,0,0,5],
                 [3,5,7,6],
                 [1,2,3,4]]

maximosLocales :: Matrix Int -> [((Int,Int),Int)]
maximosLocales p =
    [((i,j),p!(i,j)) | i <- [1..m]
                     , j <- [1..n]
                     , and [p!(a,b) < p!(i,j) | (a,b) <- vecinos (i,j)]]
    where m = nrows p
          n = ncols p
          vecinos (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)]
                                 , b <- [max 1 (j-1)..min n (j+1)]
                                 , (a,b) /= (i,j)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los árboles binarios se pueden representar con el de
-- dato algebraico
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
```

```
-- Por ejemplo, los árboles
--         9                9
--        / \              /
--       /   \            /
--      8     6          8
--     / \   / \        / \
--    3   2 4   5      3   2
-- se pueden representar por
--    ej1, ej2:: Arbol Int
--    ej1 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--    ej2 = N 9 (N 8 (N 3 H H) (N 2 H H)) H
--
-- Para indicar las posiciones del árbol se define el tipo
--   type Posicion = [Direccion]
-- donde
--   data Direccion = D | I
--     deriving Eq
-- representa un movimiento hacia la derecha (D) o a la izquierda (I). Por
-- ejemplo, las posiciones de los elementos del ej1 son
--   9 []
--   8 [I]
--   3 [I,I]
--   2 [I,D]
--   6 [D]
--   4 [D,I]
--   5 [D,D]
--
-- Definir la función
--    sustitucion :: Posicion -> a -> Arbol a -> Arbol a
-- tal que (sustitucion ds z x) es el árbol obtenido sustituyendo el
-- elemento de x en la posición ds por z. Por ejemplo,
--   ghci> sustitucion [I,D] 7 ej1
--   N 9 (N 8 (N 3 H H) (N 7 H H)) (N 6 (N 4 H H) (N 5 H H))
--   ghci> sustitucion [D,D] 7 ej1
--   N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 7 H H))
--   ghci> sustitucion [I] 7 ej1
--   N 9 (N 7 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--   ghci> sustitucion [] 7 ej1
--   N 7 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
-- ---------------------------------------------------------------------
```

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

ej1, ej2:: Arbol Int
ej1 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
ej2 = N 9 (N 8 (N 3 H H) (N 2 H H)) H

data Direccion = D | I
  deriving Eq

type Posicion = [Direccion]

sustitucion :: Posicion  -> a -> Arbol a -> Arbol a
sustitucion (I:ds) z (N x i d) = N x (sustitucion ds z i) d
sustitucion (D:ds) z (N x i d) = N x i (sustitucion ds z d)
sustitucion []     z (N _ i d) = N z i d
sustitucion _      _ H         = H
```

1.5. Examen 5 (28 de abril de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (28 de abril de 2017)
-- ---------------------------------------------------------------------

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

import Data.Char
import Data.Matrix

-- ---------------------------------------------------------------------
-- Ejercicio 1. Las rotaciones de 928160 son 928160, 281609, 816092,
-- 160928, 609281 y 92816. De las cuales, las divisibles por 4 son
-- 928160, 816092, 160928 y 92816.
--
-- Definir la función
--    nRotacionesDivisibles :: Integer -> Int
-- tal que (nRotacionesDivisibles n) es el número de rotaciones del
```

```haskell
-- número n divisibles por 4. Por ejemplo,
--    nRotacionesDivisibles 928160         ==  4
--    nRotacionesDivisibles 44             ==  2
--    nRotacionesDivisibles (1234^1000)    ==  746
--    nRotacionesDivisibles (1234^100000)  ==  76975
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

nRotacionesDivisibles :: Integer -> Int
nRotacionesDivisibles n =
  length [x | x <- rotacionesNumero n, x `mod` 4 == 0]

-- (rotacionesNumero) es la lista de la rotaciones del número n. Por
-- ejemplo,
--    rotacionesNumero 235  ==  [235,352,523]
rotacionesNumero :: Integer -> [Integer]
rotacionesNumero = map read . rotaciones . show

-- (rotaciones xs) es la lista de las rotaciones obtenidas desplazando
-- el primer elemento xs al final. Por ejemplo,
--    rotaciones [2,3,5]  ==  [[2,3,5],[3,5,2],[5,2,3]]
rotaciones :: [a] -> [[a]]
rotaciones xs = take (length xs) (iterate rota xs)

-- (rota xs) es la lista añadiendo el primer elemento de xs al
-- final. Por ejemplo,
--    rota [3,2,5,7]  ==  [2,5,7,3]
rota :: [a] -> [a]
rota (x:xs) = xs ++ [x]

-- 2ª definición
-- =============

nRotacionesDivisibles2 :: Integer -> Int
nRotacionesDivisibles2 n =
  length [x | x <- pares n, x `mod` 4 == 0]

-- (pares n) es la lista de pares de elementos consecutivos, incluyendo
```

```haskell
-- el último con el primero. Por ejemplo,
--    pares 928160  ==  [9,92,28,81,16,60]
pares :: Integer -> [Int]
pares n =
  read [last ns,head ns] : [read [a,b] | (a,b) <- zip ns (tail ns)]
  where ns = show n

-- 3ª definición
-- =============

nRotacionesDivisibles3 :: Integer -> Int
nRotacionesDivisibles3 n =
  ( length
  . filter (0 ==)
  . map (`mod` 4)
  . zipWith (\x y -> 2*x + y) d
  . tail
  . (++[i])) d
  where
    d@(i:dn) = (map digitToInt . show) n

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

--    ghci> nRotacionesDivisibles (123^1500)
--    803
--    (8.15 secs, 7,109,852,800 bytes)
--    ghci> nRotacionesDivisibles2 (123^1500)
--    803
--    (0.05 secs, 0 bytes)
--    ghci> nRotacionesDivisibles3 (123^1500)
--    803
--    (0.02 secs, 0 bytes)
--
--    ghci> nRotacionesDivisibles2 (1234^50000)
--    38684
--    (2.24 secs, 1,160,467,472 bytes)
--    ghci> nRotacionesDivisibles3 (1234^50000)
--    38684
--    (0.31 secs, 68,252,040 bytes)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios se pueden representar con el tipo
-- de dato algebraico Arbol definido por
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- Por ejemplo, los árboles
--        3                7
--       / \              / \
--      2   4            5   8
--     / \   \          / \   \
--    1   3   5        6   4   10
--                        /   /
--                       9   1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    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
--    aplica :: (a -> b) -> (a -> b) -> Arbol a -> Arbol b
-- tal que (aplica f g) es el árbol obtenido aplicando la función f a
-- los nodos que están a una distancia par de la raíz del árbol y la
-- función g a los nodos que están a una distancia impar de la raíz. Por
-- ejemplo,
--    ghci> aplica (+1) (*10) ej1
--    N 4 (N 20 (N 2 H H) (N 4 H H)) (N 40 H (N 6 H H))
--    ghci> aplica even odd ej2
--    N False (N True (N True H H) (N True (N True H H) H))
--            (N False H (N True (N True H H) H))
--    ghci> let ej3 = (N "bac" (N "de" (N "tg" (N "hi" H H) (N "js" H H)) H) H)
--    ghci> aplica head last ej3
--    N 'b' (N 'e' (N 't' (N 'i' H H) (N 's' H H)) H) H
-- ---------------------------------------------------------------------

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

```haskell
ej1, ej2 :: Arbol Int
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))

aplica :: (a -> b) -> (a -> b) -> Arbol a -> Arbol b
aplica _ _ H         = H
aplica f g (N x i d) = N (f x) (aplica g f i) (aplica g f d)

-- ----------------------------------------------------------------------
-- Ejercicio 3. Definir, usando Data.Matrix, 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
-- ejM es la matriz definida por
--    ejM :: Matrix Char
--    ejM = fromLists [" x ",
--                     "x x",
--                     " x "]
-- entonces
--    ghci> ampliaMatriz ejM 1 2
--    ( ' ' ' ' 'x' 'x' ' ' ' ' )
--    ( 'x' 'x' ' ' ' ' 'x' 'x' )
--    ( ' ' ' ' 'x' 'x' ' ' ' ' )
--
--    ghci> ampliaMatriz ejM 2 1
--    ( ' ' 'x' ' ' )
--    ( ' ' 'x' ' ' )
--    ( 'x' ' ' 'x' )
--    ( 'x' ' ' 'x' )
--    ( ' ' 'x' ' ' )
--    ( ' ' 'x' ' ' )
-- ----------------------------------------------------------------------

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

-- 1ª definición
-- =============
```

```haskell
ampliaMatriz :: Matrix a -> Int -> Int -> Matrix a
ampliaMatriz p f =
  ampliaColumnas (ampliaFilas p f)

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 (replicate f . concatMap (replicate c))
  . toLists) p

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

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

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

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

```haskell
-- Ejercicio 4. Una serie de potencias es una serie de la forma
--    a(0) + a(1)x + a(2)x^2 + a(3)x^3 + ...
--
-- Las series de potencias se pueden representar mediante listas
-- infinitas. Por ejemplo, la serie de la función exponencial es
--    e^x = 1 + x + x^2/2! + x^3/3! + ...
-- se puede representar por [1, 1, 1/2, 1/6, 1/24, 1/120, ...]
--
-- Las operaciones con series se pueden ver como una generalización de
-- las de los polinomios.
--
-- Definir la función
--    producto :: Num a => [a] -> [a] -> [a]
-- tal que (producto xs ys) es el producto de las series xs e ys. Por
-- ejemplo,
--    ghci> take 15 (producto [3,5..] [2,4..])
--    [6,22,52,100,170,266,392,552,750,990,1276,1612,2002,2450,2960]
--    ghci> take 14 (producto [10,20..] [1,3..])
--    [10,50,140,300,550,910,1400,2040,2850,3850,5060,6500,8190,10150]
--    ghci> take 16 (producto [1..] primes)
--    [2,7,17,34,62,103,161,238,338,467,627,824,1062,1343,1671,2052]
--    ghci> take 16 (producto [1,1..] primes)
--    [2,5,10,17,28,41,58,77,100,129,160,197,238,281,328,381]
--    ghci> take 16 (scanl1 (+) primes)
--    [2,5,10,17,28,41,58,77,100,129,160,197,238,281,328,381]
-- ---------------------------------------------------------------------

producto :: Num a => [a] -> [a] -> [a]
producto (x:xs) zs@(y:ys) =
  x*y : suma (producto xs zs) (map (x*) ys)

-- (suma xs ys) es la suma de las series xs e ys. Por ejemplo,
--    ghci> take 7 (suma [1,3..] [2,4..])
--    [3,7,11,15,19,23,27]
suma :: Num a => [a] -> [a] -> [a]
suma = zipWith (+)
```

1.6. Examen 6 (12 de junio de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (12 de junio de 2017)
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir las siguientes funciones
--    potenciasDe2  :: Integer -> [Integer]
--    menorPotenciaDe2 :: Integer -> Integer
-- tales que
-- + (potenciasDe2 a) es la lista de las potencias de 2 que comienzan
--   por a. Por ejemplo,
--      ghci> take 5 (potenciasDe2 3)
--      [32,32768,33554432,34359738368,35184372088832]
--      ghci> take 2 (potenciasDe2 102)
--      [1024,1028440348325753776346855739098344065614209916020987414592880064]
--      ghci> take 2 (potenciasDe2 40)
--      [4096,40564819207303340847894502572032]
-- + (menorPotenciaDe2 a) es la menor potencia de 2 que comienza con
--   el número a. Por ejemplo,
--      ghci> menorPotenciaDe2 10
--      1024
--      ghci> menorPotenciaDe2 25
--      256
--      ghci> menorPotenciaDe2 62
--      627710173538668076383578942320766641610235544464034512896
--      ghci> menorPotenciaDe2 425
--      425352958651173079329218259289710264 32
--      ghci> menorPotenciaDe2 967140655691
--      9671406556917033397649408
```

```haskell
--      ghci> [menorPotenciaDe2 a | a <- [1..10]]
--      [1,2,32,4,512,64,70368744177664,8,9007199254740992,1024]
--
-- Comprobar con QuickCheck que, para todo entero positivo a, existe una
-- potencia de 2 que empieza por a.
-- ---------------------------------------------------------------------

-- 1ª definición de potenciasDe2
-- =============================

potenciasDe2A :: Integer -> [Integer]
potenciasDe2A a =
  [x | x <- potenciasA
     , a `esPrefijo` x]

potenciasA :: [Integer]
potenciasA = [2^n | n <- [0..]]

esPrefijo :: Integer -> Integer -> Bool
esPrefijo x y = show x `isPrefixOf` show y

-- 2ª definición de potenciasDe2
-- =============================

potenciasDe2 :: Integer -> [Integer]
potenciasDe2 a = filter (a `esPrefijo`) potenciasA

-- 3ª definición de potenciasDe2
-- =============================

potenciasDe2C :: Integer -> [Integer]
potenciasDe2C a = filter (a `esPrefijo`) potenciasC

potenciasC :: [Integer]
potenciasC = iterate (*2) 1

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

-- ghci> length (show (head (potenciasDe2A 123456)))
```

```haskell
-- 19054
-- (7.17 secs, 1,591,992,792 bytes)
-- ghci> length (show (head (potenciasDe2 123456)))
-- 19054
-- (5.96 secs, 1,273,295,272 bytes)
-- ghci> length (show (head (potenciasDe2C 123456)))
-- 19054
-- (6.24 secs, 1,542,698,392 bytes)

-- Definición de menorPotenciaDe2
menorPotenciaDe2 :: Integer -> Integer
menorPotenciaDe2 = head . potenciasDe2

-- Propiedad
prop_potenciasDe2 :: Integer -> Property
prop_potenciasDe2 a =
  a > 0 ==> not (null (potenciasDe2 a))

-- Comprobación
--    ghci> quickCheck prop_potenciasDe2
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejrcicio 2. Un número natural n es una potencia perfecta si existen
-- dos números naturales m > 1 y k > 1 tales que n = m^k. Las primeras
-- potencias perfectas son
--    4 = 2², 8 = 2³, 9 = 3², 16 = 2⁴, 25 = 5², 27 = 3³, 32 = 2⁵,
--    36 = 6², 49 = 7², 64 = 2⁶, ...
--
-- Definir la sucesión
--    potenciasPerfectas :: [Integer]
-- cuyos términos son las potencias perfectas. Por ejemplo,
--    take 10 potenciasPerfectas     ==  [4,8,9,16,25,27,32,36,49,64]
--    144 `elem` potenciasPerfectas  ==  True
--    potenciasPerfectas !! 100      ==  6724
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============
```

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

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

-- (potenciasPerfectasDe x) es la lista de pares (a,b) tales que
-- x = a^b. Por ejemplo,
--    potenciasPerfectasDe 64  ==  [(2,6),(4,3),(8,2)]
--    potenciasPerfectasDe 72  ==  []
potenciasPerfectasDe :: Integer -> [(Integer,Integer)]
potenciasPerfectasDe n =
    [(m,k) | m <- takeWhile (\x -> x*x <= n) [2..]
           , k <- takeWhile (\x -> m^x <= n) [2..]
           , m^k == n]

-- 2ª solución
-- ===========

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

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

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

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

-- 3ª definición
-- =============

potenciasPerfectas :: [Integer]
potenciasPerfectas = mezclaTodas potencias

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

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

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

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

--    ghci> potenciasPerfectas1 !! 100
--    6724
--    (3.39 secs, 692758212 bytes)
```

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

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios se pueden representar con el de
-- tipo de dato algebraico
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--      deriving Show
-- Por ejemplo, los árboles
--        3                7
--       / \              / \
--      2   4            5   8
--     / \   \          / \   \
--    1   3   5        6   4   10
-- se representan por
--    ej1, ej2 :: Arbol Int
--    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 H H)) (N 8 H (N 10 H H))
--
-- Un árbol binario es continuo si el valor absoluto de la
-- diferencia de los elementos adyacentes es 1. Por ejemplo, el árbol ej1
-- es continuo ya que el valor absoluto de sus pares de elementos
-- adyacentes son
--    |3-2| = |2-1| = |2-3| = |3-4| = |4-5| = 1
-- En cambio, el ej2 no lo es ya que |8-10| =/= 1.
--
-- Definir la función
--    esContinuo :: (Num a, Eq a) => Arbol a -> Bool
-- tal que (esContinuo x) se verifica si el árbol x es continuo. Por
-- ejemplo,
--    esContinuo ej1  ==  True
--    esContinuo ej2  ==  False
-- ---------------------------------------------------------------------

data Arbol a = H
```

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

ej1, ej2 :: Arbol Int
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 H H)) (N 8 H (N 10 H H))

-- 1ª solución
-- ===========

esContinuo :: (Num a, Eq a) => Arbol a -> Bool
esContinuo H          = True
esContinuo (N _ H H) = True
esContinuo (N x i@(N y _ _) H) =
  abs (x - y) == 1 && esContinuo i
esContinuo (N x H d@(N y _ _)) =
  abs (x - y) == 1 && esContinuo d
esContinuo (N x i@(N y _ _) d@(N z _ _)) =
  abs (x - y) == 1 && esContinuo i && abs (x - z) == 1 && esContinuo d

-- 2ª solución
-- ===========

esContinuo2 :: (Num a, Eq a) => Arbol a -> Bool
esContinuo2 x =
  all esContinua (ramas x)

-- (ramas x) es la lista de las ramas del árbol x. Por ejemplo,
--    ramas ej1  ==  [[3,2,1],[3,2,3],[3,4,5]]
--    ramas ej2  ==  [[7,5,6],[7,5,4],[7,8,10]]
ramas :: Arbol a -> [[a]]
ramas H         = []
ramas (N x H H) = [[x]]
ramas (N x i d) = [x : xs | xs <- ramas i ++ ramas d]

-- (esContinua xs) se verifica si el valor absoluto de la diferencia de
-- los elementos adyacentes de xs es 1. Por ejemplo,
--    esContinua [3,2,3]   ==  True
--    esContinua [7,8,10]  ==  False
esContinua :: (Num a, Eq a) => [a] -> Bool
```

```haskell
esContinua xs =
  and [abs (x - y) == 1 | (x, y) <- zip xs (tail xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 4. El problema de las N torres consiste en colocar N torres
-- en un tablero con N filas y N columnas de forma que no haya dos
-- torres en la misma fila ni en la misma columna.
--
-- Cada solución del problema de puede representar mediante una matriz
-- con ceros y unos donde los unos representan las posiciones ocupadas
-- por las torres y los ceros las posiciones libres. Por ejemplo,
--    ( 0 1 0 )
--    ( 1 0 0 )
--    ( 0 0 1 )
-- representa una solución del problema de las 3 torres.
--
-- Definir las funciones
--    torres  :: Int -> [Matrix Int]
--    nTorres :: Int -> Integer
-- tales que
-- + (torres n) es la lista de las soluciones del problema de las n
--   torres. Por ejemplo,
--       ghci> torres 3
--       [( 1 0 0 )
--        ( 0 1 0 )
--        ( 0 0 1 )
--       ,( 1 0 0 )
--        ( 0 0 1 )
--        ( 0 1 0 )
--       ,( 0 1 0 )
--        ( 1 0 0 )
--        ( 0 0 1 )
--       ,( 0 1 0 )
--        ( 0 0 1 )
--        ( 1 0 0 )
--       ,( 0 0 1 )
--        ( 1 0 0 )
--        ( 0 1 0 )
--       ,( 0 0 1 )
--        ( 0 1 0 )
```

```
--         ( 1 0 0 )
--       ]
--   donde se ha indicado con 1 las posiciones ocupadas por las torres.
-- + (nTorres n) es el número de soluciones del problema de las n
--   torres. Por ejemplo,
--       ghci> nTorres 3
--       6
--       ghci> length (show (nTorres (10^4)))
--       35660
-- ---------------------------------------------------------------------

-- 1ª definición de torres
-- =======================

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

permutacionAmatriz :: Int -> [Int] -> Matrix Int
permutacionAmatriz n p =
    matrix n n f
    where f (i,j) | (i,j) `elem` posiciones = 1
                  | otherwise               = 0
          posiciones = zip [1..n] p

-- 2ª definición de torres
-- =======================

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

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

```
--    [[[1,0,0],[0,1,0],[0,0,1]],
--     [[0,1,0],[1,0,0],[0,0,1]],
--     [[0,0,1],[0,1,0],[1,0,0]],
--     [[0,1,0],[0,0,1],[1,0,0]],
--     [[0,0,1],[1,0,0],[0,1,0]],
--     [[1,0,0],[0,0,1],[0,1,0]]]
--    ghci> map fromLists it
--    [( 1 0 0 )
--     ( 0 1 0 )
--     ( 0 0 1 )
--    ,( 0 1 0 )
--     ( 1 0 0 )
--     ( 0 0 1 )
--    ,( 0 0 1 )
--     ( 0 1 0 )
--     ( 1 0 0 )
--    ,( 0 1 0 )
--     ( 0 0 1 )
--     ( 1 0 0 )
--    ,( 0 0 1 )
--     ( 1 0 0 )
--     ( 0 1 0 )
--    ,( 1 0 0 )
--     ( 0 0 1 )
--     ( 0 1 0 )
--    ]

-- 1ª definición de nTorres
-- ========================

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

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

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

-- Comparación de eficiencia
```

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

--    ghci> nTorres1 9
--    362880
--    (4.22 secs, 693,596,128 bytes)
--    ghci> nTorres2 9
--    362880
--    (0.00 secs, 0 bytes)
```

1.7. Examen 7 (29 de junio de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- 7º examen de evaluación continua (29 de junio de 2017)
-- ---------------------------------------------------------------------

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

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

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

```haskell
-- lista se verifica que la diferencia entre el último y el primer
-- elemento de ys es igual a la longitud de ys menos 1.
-- ---------------------------------------------------------------------

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

segmentos1 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos1 []  = []
segmentos1 [x] = [[x]]
segmentos1 (x:xs)
  | y == succ x = (x:y:ys):zs
  | otherwise   = [x] : (y:ys):zs
  where ((y:ys):zs) = segmentos1 xs

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

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

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

--    ghci> length (segmentos1 (show (5^(10^6))))
--    636114
--    (2.05 secs, 842,837,680 bytes)
--    ghci> length (segmentos2 (show (5^(10^6))))
--    636114
```

```
--    (1.21 secs, 833,432,080 bytes)

-- La propiedad es
prop_segmentos :: [Int] ->  Bool
prop_segmentos xs =
  all (\ys -> last ys - head ys == length ys - 1) (segmentos2 xs)

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

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un número de la suerte es un número natural que se
-- genera por una criba, similar a la criba de Eratóstenes, como se
-- indica a continuación:
--
-- Se comienza con la lista de los números enteros a partir de 1:
--    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25...
-- Se eliminan los números de dos en dos
--    1,  3,  5,  7,  9,   11,   13,   15,   17,   19,   21,   23,   25...
-- Como el segundo número que ha quedado es 3, se eliminan los números
-- restantes de tres en tres:
--    1,  3,      7,  9,         13,   15,         19,   21,         25...
-- Como el tercer número que ha quedado es 7, se eliminan los números
-- restantes de siete en siete:
--    1,  3,      7,  9,         13,   15,               21,         25...
--
-- Este procedimiento se repite indefinidamente y los supervivientes son
-- los números de la suerte:
--    1,3,7,9,13,15,21,25,31,33,37,43,49,51,63,67,69,73,75,79
--
-- Definir la sucesión
--    numerosDeLaSuerte :: [Int]
-- cuyos elementos son los números de la suerte. Por ejemplo,
--    ghci> take 20 numerosDeLaSuerte
--    [1,3,7,9,13,15,21,25,31,33,37,43,49,51,63,67,69,73,75,79]
--    ghci> numerosDeLaSuerte !! 1500
--    13995
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición
numerosDeLaSuerte :: [Int]
numerosDeLaSuerte = criba 3 [1,3..]
  where
    criba i (n:s:xs) =
      n : criba (i + 1) (s : [x | (n, x) <- zip [i..] xs
                                , rem n s /= 0])

-- 2ª definición
numerosDeLaSuerte2 :: [Int]
numerosDeLaSuerte2 =  1 : criba 2 [1, 3..]
  where criba k xs = z : criba (k + 1) (aux xs)
          where z = xs !! (k - 1 )
                aux ws = us ++ aux vs
                  where (us, _:vs) = splitAt (z - 1) ws

-- Comparación de eficiencia
--    ghci> numerosDeLaSuerte2 !! 300
--    2217
--    (3.45 secs, 2,873,137,632 bytes)
--    ghci> numerosDeLaSuerte !! 300
--    2217
--    (0.04 secs, 22,416,784 bytes)

-- ---------------------------------------------------------------------
-- 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   3
--        |       /|\      /|\  |
--        4      4 5 6    4 5 6 7
--
-- se representan por
--    ej1, ej2, ej3 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 1 [N 2 [], N 3 [N 4 [], N 5 [], N 6 []]
```

```
--    ej3 = N 1 [N 2 [N 4 [], N 5 [], N 6 []], N 3 [N 7 []]]
--
-- En el primer ejemplo la máxima ramificación es 2 (en el nodo 1 que
-- tiene 2 hijos), la del segundo es 3 (en el nodo 3 que tiene 3
-- hijos) y la del tercero es 3 (en el nodo 3 que tiene 3 hijos).
--
-- Definir la función
--    maximaRamificacion :: Arbol a -> Int
-- tal que (maximaRamificacion a) es la máxima ramificación del árbol
-- a. Por ejemplo,
--    maximaRamificacion ej1  ==  2
--    maximaRamificacion ej2  ==  3
--    maximaRamificacion ej3  ==  3
-- ---------------------------------------------------------------------

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

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

maximaRamificacion :: Arbol a -> Int
maximaRamificacion (N _ []) = 0
maximaRamificacion (N x xs) =
  max (length xs) (maximum (map maximaRamificacion xs))

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un mapa con dos tipos de regiones (por ejemplo, tierra y
-- mar) se puede representar mediante una matriz de ceros y unos.
--
-- Para los ejemplos usaremos los mapas definidos por
--    type Punto = (Int,Int)
--    type Mapa  = Array Punto Int
--
--    mapa1, mapa2 :: Mapa
--    mapa1 = listArray ((1,1),(3,4))
--                      [1,1,0,0,
--                       0,1,0,0,
```

```
--                        1,0,0,0]
--    mapa2 = listArray ((1,1),(10,20))
--                       [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
--                        1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,
--                        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
--                        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
--                        1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,
--                        0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
--                        0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
--                        1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,
--                        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
--                        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
--
-- Definir las funciones
--    alcanzables :: Mapa -> Punto -> [Punto]
--    esAlcanzable :: Mapa -> Punto -> Punto -> Bool
-- tales que
-- + (alcanzables p) es la lista de los puntos de mapa m que se pueden
--   alcanzar a partir del punto p moviéndose en la misma región que p
--   (es decir, a través de ceros si el elemento de m en p es un cero o a
--   través de unos, en caso contrario) y los movimientos permitidos son
--   ir hacia el norte, sur este u oeste (pero no en diagonal). Por
--   ejemplo,
--      alcanzables mapa1 (1,1)  ==  [(2,2),(1,2),(1,1)]
--      alcanzables mapa1 (1,2)  ==  [(2,2),(1,1),(1,2)]
--      alcanzables mapa1 (1,3)  ==  [(3,2),(3,4),(3,3),(2,3),(2,4),(1,4),(1,3)]
--      alcanzables mapa1 (3,1)  ==  [(3,1)]
-- + (esAlcanzable m p1 p2) se verifica si el punto p1 es alcanzable
--   desde el p1 en el mapa m. Por ejemplo,
--      esAlcanzable mapa1 (1,4) (3,2)    ==  True
--      esAlcanzable mapa1 (1,4) (3,1)    ==  False
--      esAlcanzable mapa2 (2,3) (8,16)   ==  True
--      esAlcanzable mapa2 (8,1) (7,3)    ==  True
--      esAlcanzable mapa2 (1,1) (10,20)  ==  False
-- ---------------------------------------------------------------------

type Punto = (Int,Int)
type Mapa  = Array Punto Int

mapa1, mapa2 :: Mapa
```

```haskell
mapa1 = listArray ((1,1),(3,4))
                  [1,1,0,0,
                   0,1,0,0,
                   1,0,0,0]
mapa2 = listArray ((1,1),(10,20))
                  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                   1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,
                   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
                   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
                   1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,
                   0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                   0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                   1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,
                   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

alcanzables :: Mapa -> Punto -> [Punto]
alcanzables mapa p = aux [p] []
  where region = mapa ! p
        (_,(m,n)) = bounds mapa
        vecinos (i,j) = [(a,b) | (a,b) <- [(i,j+1),(i,j-1),(i+1,j),(i-1,j)]
                               , 1 <= a && a <= m
                               , 1 <= b && b <= n
                               , mapa ! (a,b) == region]
        aux [] ys = ys
        aux (x:xs) ys
          | x `elem` ys = aux xs ys
          | otherwise   = aux (vecinos x ++ xs) (x:ys)

esAlcanzable :: Mapa -> Punto -> Punto -> Bool
esAlcanzable m p1 p2 =
  p2 `elem` alcanzables m p1
```

1.8. Examen 8 (8 de septiembre de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- Examen de la convocatoria de Septiembre (8 de septiembre de 2017)
-- ---------------------------------------------------------------------
```

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. La órbita prima de un número n es la sucesión construida
-- de la siguiente forma:
-- + si n es compuesto su órbita no tiene elementos
-- + si n es primo, entonces su órbita está formada por n y la órbita del
--   número obtenido sumando n y sus dígitos.
-- Por ejemplo, para el número 11 tenemos:
--   11 es primo, consideramos 11+1+1 = 13
--   13 es primo, consideramos 13+1+3 = 17
--   17 es primo, consideramos 17+1+7 = 25
--   25 es compuesto, su órbita está vacía
-- Así, la órbita prima de 11 es [11, 13, 17].
--
-- Definir la función
--    orbita :: Integer -> [Integer]
-- tal que (orbita n) es la órbita prima de n. Por ejemplo,
--    orbita 11 == [11,13,17]
--    orbita 59 == [59,73,83]
--
-- Calcular el menor número cuya órbita prima tiene más de 3 elementos.
-- ---------------------------------------------------------------------

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

orbita :: Integer -> [Integer]
orbita n | not (isPrime n) = []
         | otherwise       = n : orbita (n + sum (cifras n))

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
--    cifras 32542 == [3,2,5,4,2]
```

```haskell
cifras :: Integer -> [Integer]
cifras n = [read [x]| x <- show n]

-- 2ª definición (con iterate)
-- ===========================

orbita2 :: Integer -> [Integer]
orbita2 n = takeWhile isPrime (iterate f n)
  where f x = x + sum (cifras x)

-- Comprobación de la equivalencia de las definiciones de 'orbita'
-- ===============================================================

--    > quickCheck prop_equiv_orbita
--    +++ OK, passed 100 tests.
prop_equiv_orbita :: Integer -> Bool
prop_equiv_orbita n =
  orbita n == orbita2 n

-- El cálculo es
--    > head [x | x <- [1,3..], length (orbita x) > 3]
--    277
--
--    > orbita 277
--    [277,293,307,317]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumas :: (Num a, Ord a) => Int -> [a] -> a -> [[a]]
-- tal que (sumas n ns x) es la lista de las descomposiciones de x como
-- sumas de n sumandos de la lista ns. Por ejemplo,
--    sumas 2 [1,2] 3   == [[1,2]]
--    sumas 2 [-1] (-2) == [[-1,-1]]
--    sumas 2 [-1,3,-1] 2 == [[-1,3]]
--    sumas 2 [1,2] 4     == [[2,2]]
--    sumas 2 [1,2] 5     == []
--    sumas 3 [1,2] 5     == [[1,2,2]]
--    sumas 3 [1,2] 6     == [[2,2,2]]
--    sumas 2 [1,2,5] 6   == [[1,5]]
--    sumas 2 [1,2,3,5] 4 == [[1,3],[2,2]]
```

```haskell
--    sumas 2 [1..5] 6    == [[1,5],[2,4],[3,3]]
--    sumas 3 [1..5] 7    == [[1,1,5],[1,2,4],[1,3,3],[2,2,3]]
--    sumas 3 [1..200] 4  == [[1,1,2]]
-- ---------------------------------------------------------------------

-- 1ª definición (fuerza bruta)
-- ============================

sumas1 :: (Num a, Ord a) => Int -> [a] -> a -> [[a]]
sumas1 n ns x =
  [xs | xs <- combinacionesR n (nub (sort ns))
      , sum xs == x]

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

-- 2ª definición (divide y vencerás)
-- ==================================

sumas2 :: (Num a, Ord a) => Int -> [a] -> a -> [[a]]
sumas2 n ns x = nub (sumasAux n ns x)
  where sumasAux :: (Num a, Ord a) => Int -> [a] -> a -> [[a]]
        sumasAux 1 ns' x'
          | x' `elem` ns' = [[x']]
          | otherwise     = []
        sumasAux n' ns' x' =
          concat [[y:zs | zs <- sumasAux (n'-1) ns' (x'-y)
                        , y <= head zs]
                  | y <- ns']

-- 3ª definición
```

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

sumas3 :: (Num a, Ord a) => Int -> [a] -> a -> [[a]]
sumas3 n ns x = nub $ aux n (sort ns) x
  where aux 0 _ _  = []
        aux _ [] _ = []
        aux 1 ys x | x `elem` ys = [[x]]
                   | otherwise   = []
        aux n (y:ys) x = aux n ys x ++
                         map (y:) (aux (n - 1) (y : ys) (x - y))

-- Equivalencia de las definiciones
-- ================================

-- (prop_equiv_sumas n ns x) se verifica si las definiciones de
-- 'sumas' son equivalentes para n, ns y x. Por ejemplo,
--    > quickCheckWith (stdArgs {maxSize=7}) prop_equiv_sumas
--    +++ OK, passed 100 tests.
prop_equiv_sumas :: Positive Int -> [Int] -> Int -> Bool
prop_equiv_sumas (Positive n) ns x =
  all (== normal (sumas1 n ns x))
      [ normal (sumas2 n ns x)
      , normal (sumas3 n ns x)]
  where normal = sort . map sort

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

--    > sumas1 3 [1..200] 4
--    [[1,1,2]]
--    (2.52 secs, 1,914,773,472 bytes)
--    > sumas2 3 [1..200] 4
--    [[1,1,2]]
--    (0.17 secs, 25,189,688 bytes)
--    ghci> sumas3 3 [1..200] 4
--    [[1,1,2]]
--    (0.08 secs, 21,091,368 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
```

```haskell
--    diagonalesPrincipales :: Array (Int,Int) a -> [[a]]
-- tal que (diagonalesPrincipales p) es la lista de las diagonales
-- principales de la matriz p. Por ejemplo, para la matriz
--    1  2  3  4
--    5  6  7  8
--    9 10 11 12
-- la lista de sus diagonales principales es
--    [[9],[5,10],[1,6,11],[2,7,12],[3,8],[4]]
-- En Haskell,
--    > diagonalesPrincipales (listArray ((1,1),(3,4)) [1..12])
--    [[9],[5,10],[1,6,11],[2,7,12],[3,8],[4]]
-- ---------------------------------------------------------------------

diagonalesPrincipales :: Array (Int,Int) a -> [[a]]
diagonalesPrincipales p =
    [[p!ij1 | ij1 <- extension ij] | ij <- iniciales]
    where (_,(m,n)) = bounds p
          iniciales = [(i,1) | i <- [m,m-1..2]] ++ [(1,j) | j <- [1..n]]
          extension (i,j) = [(i+k,j+k) | k <- [0..min (m-i) (n-j)]]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Las expresiones aritméticas formadas por sumas de
-- números y variables se pueden representar mediante el siguiente tipo
-- de datos
--    data Exp = N Int
--             | V String
--             | S Exp Exp
--      deriving Show
-- Por ejemplo, la expresión "(x + 3) + (2 + y)" se representa por
--    S (S (V "x") (N 3)) (S (N 2) (V "y"))
--
-- Definir la función
--    simplificada :: Exp -> Exp
-- tal que (simplificada e) es una expresión equivalente a la expresión
-- e pero sólo con un número como máximo. Por ejemplo,
--    > simplificada (S (S (V "x") (N 3)) (S (N 2) (V "y")))
--    S (N 5) (S (V "x") (V "y"))
--    > simplificada (S (S (S (V "x") (N 3)) (S (N 2) (V "y"))) (N (-5)))
--    S (V "x") (V "y")
--    ghci> simplificada (S (V "x") (V "y"))
```

```
--    S (V "x") (V "y")
--    ghci> simplificada (S (N 2) (N 3))
--    N 5
-- ---------------------------------------------------------------------

data Exp = N Int
         | V String
         | S Exp Exp
  deriving Show

simplificada :: Exp -> Exp
simplificada e
  | null vs   = N x
  | x == 0    = e1
  | otherwise = S (N x) e1
  where x  = numero e
        vs = variables e
        e1 = sumaVariables vs

-- (numero e) es el número obtenido sumando los números de la expresión
-- e. Por ejemplo,
--    numero (S (S (V "x") (N 3)) (S (N 2) (V "y")))              == 5
--    numero (S (S (S (V "x") (N 3)) (S (N 2) (V "y"))) (N (-5))) == 0
--    numero (S (V "x") (V "y"))                                  == 0
numero :: Exp -> Int
numero (N x)     = x
numero (V _)     = 0
numero (S e1 e2) = numero e1 + numero e2

-- (variables e) es la lista de las variables de la expresión e. Por
-- ejemplo,
--    variables (S (S (V "x") (N 3)) (S (N 2) (V "y")))   == ["x","y"]
--    variables (S (S (V "x") (N 3)) (S (V "y") (V "x"))) == ["x","y","x"]
--    variables (S (N 2) (N 3))                           == []
variables :: Exp -> [String]
variables (N _)     = []
variables (V x)     = [x]
variables (S e1 e2) = variables e1 ++ variables e2

-- (sumaVariables xs) es la expresión obtenida sumando las variables
```

```
-- xs. Por ejemplo,
--    ghci> sumaVariables ["x","y","z"]
--    S (V "x") (S (V "y") (V "z"))
--    ghci> sumaVariables ["x"]
--    V "x"
sumaVariables :: [String] -> Exp
sumaVariables [x]    = V x
sumaVariables (x:xs) = S (V x) (sumaVariables xs)
```

1.9. Examen 9 (21 de noviembre de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- Convocatoria de diciembre (21 de noviembre de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Matrix

-- ---------------------------------------------------------------------
-- Ejercicio 1. Dado un conjunto A de enteros positivos, una partición
-- de A son dos conjuntos disjuntos y no vacíos A1 y A2 cuya unión es
-- A. Decimos que una partición es "buena" si el mínimo común múltiplo
-- de los elementos de A1 coincide con el máximo común divisor de los
-- elementos de A2.
--
-- Definir la función
--    particionesBuenas :: [Int] -> [([Int],[Int])]
-- tal que (particionesBuenas xs) es la lista de las particiones buenas
-- de xs. Por ejemplo,
--    particionesBuenas [1,2,3,6]  == [([1],[2,3,6]),([1,2,3],[6])]
--    particionesBuenas [1..10]    == [([1],[2,3,4,5,6,7,8,9,10])]
--    particionesBuenas [2..10]    == []
-- ---------------------------------------------------------------------

-- 1ª definición de particionesBuenas
```

```haskell
particionesBuenas :: [Int] -> [([Int],[Int])]
particionesBuenas xs =
  [p | p <- particiones xs
     , esBuena p]

-- 2ª definición de particionesBuenas
particionesBuenas2 :: [Int] -> [([Int],[Int])]
particionesBuenas2 xs =
  filter esBuena (particiones xs)

-- 3ª definición de particionesBuenas
particionesBuenas3 :: [Int] -> [([Int],[Int])]
particionesBuenas3 =
  filter esBuena . particiones

-- (particiones xs) es la lista de las particiones de xs. Por ejemplo,
--    ghci> particiones [3,2,5]
--    [([],[3,2,5]),
--     ([3],[2,5]),
--     ([2],[3,5]),
--     ([3,2],[5]),
--     ([5],[3,2]),
--     ([3,5],[2]),
--     ([2,5],[3]),
--     ([3,2,5],[])]

-- 1ª definición de particiones
particiones1 :: [Int] -> [([Int],[Int])]
particiones1 xs = [(ys,xs \\ ys) | ys <- subsequences xs]

-- 2ª definición de particiones
particiones2 :: [Int] -> [([Int],[Int])]
particiones2 xs = map f (subsequences xs)
  where f ys = (ys, xs \\ ys)

-- 3ª definición de particiones
particiones3 :: [Int] -> [([Int],[Int])]
particiones3 xs = map (\ys -> (ys, xs \\ ys)) (subsequences xs)

-- Comparación de eficiencia de particiones
```

```haskell
--    ghci> length (particiones1 [1..24])
--    16777216
--    (3.04 secs, 4,160,894,648 bytes)
--    ghci> length (particiones2 [1..24])
--    16777216
--    (0.72 secs, 3,221,368,088 bytes)
--    ghci> length (particiones3 [1..24])
--    16777216
--    (0.72 secs, 3,221,367,168 bytes)

-- Usaremos la 3ª definición de particiones
particiones :: [Int] -> [([Int],[Int])]
particiones = particiones3

-- (esBuena (xs,ys)) se verifica si las listas xs e ys son no vacía y el
-- mínimo común múltiplo de xs es igual al máximo común divisor de ys.
esBuena :: ([Int],[Int]) -> Bool
esBuena (xs,ys) =
     not (null xs)
  && not (null ys)
  && mcmL xs == mcdL ys

-- (mcdL xs) es el máximo común divisor de xs.

-- 1ª definición de mcdL
mcdL1 :: [Int] -> Int
mcdL1 [x] = x
mcdL1 (x:y:xs) = gcd x (mcdL (y:xs))

-- 2ª definición de mcdL
mcdL2 :: [Int] -> Int
mcdL2 = foldl1' gcd

-- Comparación de eficiencia de mcdL
--    ghci> mcdL1 [1..3*10^6]
--    1
--    (2.02 secs, 1,770,213,520 bytes)
--    ghci> mcdL2 [1..3*10^6]
--    1
--    (0.50 secs, 1,032,135,400 bytes)
```

```
--
--    ghci> mcdL1 [1..5*10^6]
--    *** Exception: stack overflow
--    ghci> mcdL2 [1..5*10^6]
--    1
--    (1.42 secs, 1,720,137,128 bytes)

-- Usaremos la 2ª definición de mcdL
mcdL :: [Int] -> Int
mcdL = mcdL2

-- (mcmL xs) es el mínimo común múltiplo de xs.

-- 1ª definición de mcmL
mcmL1 :: [Int] -> Int
mcmL1 [x] = x
mcmL1 (x:y:xs) = lcm x (mcmL1 (y:xs))

-- 2ª definición de mcmL
mcmL2 :: [Int] -> Int
mcmL2 = foldl1' lcm

-- Comparación de eficiencia de mcmL
--    ghci> mcmL1 [1..5*10^6]
--    505479828665794560
--    (6.18 secs, 7,635,567,360 bytes)
--    ghci> mcmL2 [1..5*10^6]
--    26232203725766656
--    (2.71 secs, 5,971,248,720 bytes)
--    ghci> mcmL1 [1..10^7]
--    *** Exception: stack overflow
--    ghci> mcmL2 [1..10^7]
--    1106056739033186304
--    (5.99 secs, 12,269,073,648 bytes)

-- Usaremos la 2ª definición de mcmL
mcmL :: [Int] -> Int
mcmL = mcdL2

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

```haskell
-- Ejercicio 2. Un número natural n se puede descomponer de varias
-- formas como suma de potencias k-ésimas de números naturales. Por
-- ejemplo, 100 se puede descomponer en potencias cuadradas
--    100 = 1 + 9 + 16 + 25 + 49
--        = 36 + 64
--        = 100
-- O bien como potencias cúbicas,
--    100 = 1 + 8 + 27 + 64
-- No hay ninguna descomposición de 100 como potencias cuartas.
--
-- Definir la función
--    descomPotencia :: Int -> Int -> [[Int]]
-- tal que (descomPotencia n k) es la lista de las descomposiciones de n
-- como potencias k-ésimas de númenos naturales. Por ejemplo,
--    descomPotencia 100 2 == [[1,9,16,25,49],[36,64],[100]]
--    descomPotencia 100 3 == [[1,8,27,64]]
--    descomPotencia 100 4 == []
-- ---------------------------------------------------------------------

descomPotencia :: Int -> Int -> [[Int]]
descomPotencia x n =
  descomposiciones x (takeWhile (<=x) (map (^n) [1..]))

descomposiciones :: Int -> [Int] -> [[Int]]
descomposiciones n [] = []
descomposiciones n (x:xs)
  | x > n     = []
  | x == n    = [[n]]
  | otherwise = map (x:) (descomposiciones (n-x) xs)
                ++ descomposiciones n xs

-- ---------------------------------------------------------------------
-- 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,
--    ghci> matrizPascal 5
--    ( 1 0 0 0 0 )
--    ( 1 1 0 0 0 )
--    ( 1 2 1 0 0 )
--    ( 1 3 3 1 0 )
--    ( 1 4 6 4 1 )
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- 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         1             1
--     / \       / \           / \
--    8   3     5   3         5   3
--        |       /|\       /|\  |
--        4      4 7 6     4 7 6 7
-- se representan por
--    ej1, ej2, ej3 :: Arbol Int
--    ej1 = N 1 [N 8 [],N 3 [N 4 []]]
--    ej2 = N 1 [N 5 [], N 3 [N 4 [], N 7 [], N 6 []]]
--    ej3 = N 1 [N 5 [N 4 [], N 7 [], N 6 []], N 3 [N 7 []]]
--
-- El peso de una rama de un árbol es la suma de los valores en sus
-- nodos y en sus hojas. Por ejemplo, en el primer árbol su rama
-- izquierda pesa 9 (1+8) y su rama derecha pesa 8 (1+3+4).
--
-- Definir la función
--    minimoPeso :: Arbol Int -> Int
-- tal que (minimoPeso x) es el mínimo de los pesos de la rama del árbol
-- x. Por ejemplo,
--    minimoPeso ej1  ==  8
--    minimoPeso ej2  ==  6
--    minimoPeso ej3  ==  10
-- ---------------------------------------------------------------------

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 5 [], N 3 [N 4 [], N 7 [], N 6 []]]
ej3 = N 1 [N 5 [N 4 [], N 7 [], N 6 []], N 3 [N 7 []]]

-- 1ª definición
-- =============

minimoPeso :: Arbol Int -> Int
```

```haskell
minimoPeso x = minimum (map sum (ramas x))

-- (ramas x) es la lista de las ramas del árbol x. Por ejemplo,
--    ramas ej1  ==  [[1,8],[1,3,4]]
--    ramas ej2  ==  [[1,5],[1,3,4],[1,3,7],[1,3,6]]
--    ramas ej3  ==  [[1,5,4],[1,5,7],[1,5,6],[1,3,7]]
ramas :: Arbol a -> [[a]]
ramas (N r []) = [[r]]
ramas (N r as) = [(r:rs) |  a<-as, rs <-ramas a]

-- 2ª definición
-- =============

minimoPeso2 :: Arbol Int -> Int
minimoPeso2 = minimum . map sum . ramas

-- 3ª definición
-- =============

minimoPeso3 :: Arbol Int -> Int
minimoPeso3 (N r []) = r
minimoPeso3 (N r as) = r + minimum (map minimoPeso3 as)
```

2

Exámenes del grupo 2

Francisco J. Martín

2.1. Examen 1 (26 de octubre de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 2)
-- 1º examen de evaluación continua (26 de octubre de 2015)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Consideremos las siguientes regiones del plano:
--    R1: Puntos (x,y) tales que |x| + |y| <= 1    (rombo unidad)
--    R2: Puntos (x,y) tales que x^2 + y^2 <= 1    (círculo unidad)
--    R3: Puntos (x,y) tales que max(|x|,|y|) <= 1 (cuadrado unidad)
-- donde |x| es el valor absoluto del número x.
--
-- Se cumple que la región R1 está contenida en la región R2 y la región
-- R2 está contenida en la región R3, pero ninguna de ellas son iguales.
--
-- Definir la función
--    numeroRegiones :: (Float,Float) -> Int
-- tal que (numeroRegiones p) es el número de regiones R1, R2 y R3 en
-- las que está contenido el punto p. Por ejemplo,
--    numeroRegiones (0.2,0.3)    ==  3
--    numeroRegiones (-0.5,0.6)   ==  2
--    numeroRegiones (0.8,-0.8)   ==  1
--    numeroRegiones (-0.9,-1.2)  ==  0
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución
numeroRegiones :: (Float,Float) -> Int
numeroRegiones (x,y)
  | abs x + abs y <= 1        =  3
  | x^2 + y^2 <= 1            =  2
  | max (abs x) (abs y) <= 1  =  1
  | otherwise                 =  0

-- 2ª solución
numeroRegiones2 :: (Float,Float) -> Int
numeroRegiones2 (x,y) =
  sum [1 | c <- [ abs x + abs y <= 1
                , x^2 + y^2 <= 1
                , max (abs x) (abs y) <= 1],
           c]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Dos números A y B son coprimos si no tienen ningún
-- factor primo común. Por ejemplo los números 15 y 49 son coprimos, ya
-- que los factores primos de 15 son 3 y 5; y el único factor primo de
-- 49 es 7. Por otro lado, los números 15 y 35 no son coprimos, ya que
-- ambos son divisibles por 5.
--
-- Definir por comprensión la función
--    primerParCoprimos :: [Integer] -> (Integer,Integer)
-- tal que (primerParCoprimos xs) es el primer par de números
-- consecutivos en la lista xs que son coprimos entre sí. Si en la
-- lista no hay números coprimos consecutivos el resultado debe ser
-- (0,0). Por ejemplo,
--    primerParCoprimos [3,9,13,26]   ==  (9,13)
--    primerParCoprimos [3,6,1,7,14]  ==  (6,1)
--    primerParCoprimos [3,6,9,12]    ==  (0,0)
-- ---------------------------------------------------------------------

primerParCoprimos :: [Integer] -> (Integer,Integer)
primerParCoprimos xs =
  head ([(x,y) | (x,y) <- zip xs (tail xs), gcd x y == 1] ++ [(0,0)])

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Una forma de aproximar el logaritmo de 2 es usando la
```

```
-- siguiente igualdad:
--
--                       1     1     1     1
--          ln 2 = 1 - --- + --- - --- + --- - ....
--                       2     3     4     5
--
-- Es decir, la serie cuyo término general n-ésimo es el cociente entre
-- (-1)^(n+1) y el propio número n:
--
--                     (-1)^(n+1)
--           s(n) =  ------------
--                         n
--
-- Definir por comprensión la función:
--    aproximaLn2C :: Double -> Double
-- tal que (aproximaLn2C n) es la aproximación del logaritmo de 2
-- calculada con la serie anterior hasta el término n-ésimo. Por
-- ejemplo,
--    aproximaLn2C 10  ==  0.6456349206349207
--    aproximaLn2C 30  ==  0.6687714031754279
--    aproximaLn2C 50  ==  0.6767581376913979
-- ----------------------------------------------------------------------

aproximaLn2C :: Double -> Double
aproximaLn2C n =
  sum [(-1)**(i+1) / i | i <- [1..n] ]

-- ----------------------------------------------------------------------
-- Ejercicio 3.2. Definir por recursión la función:
--    aproximaLn2R :: Double -> Double
-- tal que (aproximaLn2R n) es la aproximación del logaritmo de 2
-- calculada con la serie anterior hasta el término n-ésimo. Por
-- ejemplo,
--    aproximaLn2R 10  ==  0.6456349206349207
--    aproximaLn2R 30  ==  0.6687714031754279
--    aproximaLn2R 50  ==  0.6767581376913979
-- ----------------------------------------------------------------------

aproximaLn2R :: Double -> Double
aproximaLn2R 1 = 1
```

```haskell
aproximaLn2R n = ((-1)**(n+1) / n) + aproximaLn2R (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Dado un número natural cualquiera, N, podemos formar
-- otros dos números, uno a partir de las cifras pares de N y otro a
-- partir de las cifras impares de N, a los que llamaremos
-- respectivamente componente par de N y componente impar de N. Por
-- ejemplo, la componente par del número 1235678 es 268 y la componente
-- impar es 1357.
--
-- Definir por recursión la función
--    componentePar :: Integer -> Integer
-- tal que (componentePar n) es la componente par del número natural
-- n. En el caso en que n no tenga cifras pares el resultado debe ser
-- 0. Por ejemplo,
--    componentePar 1235678  ==  268
--    componentePar 268      ==  268
--    componentePar 375      ==  0
-- ---------------------------------------------------------------------

componentePar :: Integer -> Integer
componentePar n
  | null pares = 0
  | otherwise  = read pares
  where pares = [c | c <- show n
                   , c `elem` "02468"]
```

2.2. Examen 2 (30 de noviembre de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (30 de noviembre de 2016)
-- =====================================================================

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

import Data.List

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

```haskell
-- Ejercicio 1. Dada una lista de números ns, su lista de degradación es
-- la lista que se obtiene contando para cada elemento de la lista
-- original n, el número de elementos consecutivos en la lista que son
-- estrictamente menores que n. Por ejemplo, la lista de degradación de
-- [5,3,1,7,6,2,8] es [2,1,0,2,1,0,0] pues:
-- + Al 5 le siguen 2 elementos consecutivos estrictamente menores (3 y 1)
-- + Al 3 le sigue 1 elemento consecutivo estrictamente menor (1)
-- + Al 1 no le sigue ningún elemento estrictamente menor
-- + Al 7 le siguen 2 elementos consecutivos estrictamente menores (6 y 2)
-- + Al 6 le sigue 1 elemento consecutivo estrictamente menor (2)
-- + Al 2 no le sigue ningún elemento estrictamente menor
-- + Al 8 no le sigue ningún elemento.
--
-- Definir la función
--   listaDegradacion :: [Int] -> [Int]
-- tal que (listaDegradacion ns) es la lista de degradación de la lista
-- ns. Por ejemplo,
--   listaDegradacion [5,3,1,7,6,2,8]  ==  [2,1,0,2,1,0,0]
--   listaDegradacion [1,2,3,4,5]      ==  [0,0,0,0,0]
--   listaDegradacion [5,4,3,2,1]      ==  [4,3,2,1,0]
--   listaDegradacion [9,7,1,4,8,4,0]  ==  [6,2,0,0,2,1,0]
-- ---------------------------------------------------------------------

listaDegradacion :: [Int] -> [Int]
listaDegradacion xs =
  [length (takeWhile (<y) ys) | (y:ys) <- init (tails xs)]

-- ---------------------------------------------------------------------
-- Ejercicio 2. 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 en la primera 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]. Ocasionalmente, la descripción de una lista es más
-- corta que la propia lista.
--
-- Se considera la función
--   originalDescripcion :: [Int] -> [Int]
-- tal que (originalDescripcion xs) es la lista ys tal que la descripción
```

```haskell
-- de ys es la lista xs. Es decir, la lista xs indica qué elementos hay
-- en ys y cuántas veces se repiten. Por ejemplo,
--   originalDescripcion [3,1,2,3,2,2]  ==  [1,1,1,3,3,2,2]
--   originalDescripcion [1,1,3,2,2,3]  ==  [1,2,2,2,3,3]
--   originalDescripcion [2,1,3,3,3,1]  ==  [1,1,3,3,3,1,1,1]
-- ---------------------------------------------------------------------

originalDescripcion :: [Int] -> [Int]
originalDescripcion (n:x:xs) =
  replicate n x ++ originalDescripcion xs
originalDescripcion _ = []

-- ---------------------------------------------------------------------
-- Ejercicio 3. Se dice que un elemento x de una lista xs respeta la
-- ordenación si x es mayor o igual que todos lo que tiene delante en xs
-- y es menor o igual que todos lo que tiene detrás en xs. Por ejemplo,
-- en la lista lista [3,2,1,4,6,5,7,9,8] el número 4 respeta la
-- ordenación pero el número 5 no la respeta (porque es mayor que el 6
-- que está delante).
--
-- Definir la función
--    respetuosos :: Ord a => [a] -> [a]
-- tal que (respetuosos xs) es la lista de los elementos de xs que
-- respetan la ordenación. Por ejemplo,
--    respetuosos [3,2,1,4,6,4,7,9,8]  ==  [4,7]
--    respetuosos [2,1,3,4,6,4,7,8,9]  ==  [3,4,7,8,9]
--    respetuosos "abaco"               ==  "aco"
--    respetuosos "amor"                ==  "amor"
--    respetuosos "romanos"             ==  "s"
--    respetuosos [1..9]                ==  [1,2,3,4,5,6,7,8,9]
--    respetuosos [9,8..1]              ==  []
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
respetuosos :: Ord a => [a] -> [a]
respetuosos xs =
  [z | k <- [0..n-1]
     , let (ys,z:zs) = splitAt k xs
     , all (<=z) ys
     , all (>=z) zs]
```

```
  where n = length xs

-- 2ª definición (por recursión):
respetuosos2 :: Ord a => [a] -> [a]
respetuosos2 = aux [] []
  where aux zs _  []     = reverse zs
        aux zs ys (x:xs)
          | all (<=x) ys && all (>=x) xs = aux (x:zs) (x:ys) xs
          | otherwise                    = aux zs     (x:ys) xs

-- 2ª definición
respetuosos3 :: Ord a => [a] -> [a]
respetuosos3 xs = [ x | (ys,x,zs) <- zip3 (inits xs) xs (tails xs)
                      , all (<=x) ys
                      , all (x<=) zs ]

-- 4ª solución
respetuosos4 :: Ord a =>[a] ->[a]
respetuosos4 xs =
  [x | (a, x, b) <- zip3 (scanl1 max xs) xs (scanr1 min xs)
     , a <= x && x <= b]

-- Comparación de eficiencia
--    ghci> length (respetuosos [1..3000])
--    3000
--    (3.31 secs, 1,140,407,224 bytes)
--    ghci> length (respetuosos2 [1..3000])
--    3000
--    (2.85 secs, 587,082,160 bytes)
--    ghci> length (respetuosos3 [1..3000])
--    3000
--    (2.12 secs, 785,446,880 bytes)
--    ghci> length (respetuosos4 [1..3000])
--    3000
--    (0.02 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un número equilibrado es aquel en el que la suma de
-- los dígitos que ocupan una posición par es igual a la suma de los
-- dígitos que ocupan una posición impar.
```

```
--
-- Definir la constante
--    equilibrados :: [Int]
-- cuyo valor es la lista infinita de todos los números equilibrados. Por
-- ejemplo,
--    take 13 equilibrados  ==  [0,11,22,33,44,55,66,77,88,99,110,121,132]
--    equilibrados!!1000    ==  15345
--    equilibrados!!2000    ==  31141
--    equilibrados!!3000    ==  48686
-- ---------------------------------------------------------------------

equilibrados :: [Int]
equilibrados =
  filter equilibrado [0..]

equilibrado :: Int -> Bool
equilibrado n =
  sum (zipWith (*) (digitos n) (cycle [1,-1])) == 0

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

2.3. Examen 3 (31 de enero de 2017)

El examen es común con el del grupo 1 (ver página 24).

2.4. Examen 4 (8 de marzo de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (8 de marzo de 2017)
-- =====================================================================

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

import Data.Array
import qualified Data.Set as S
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Un número paritario es aquel que tiene una cantidad par
-- de cifras pares y una cantidad impar de cifras impares. Por ejemplo,
-- 111, 290, 11690, 29451 y 1109871 son paritarios, mientras que 21,
-- 2468 y 11358 no lo son.
--
-- Definir la constante
--    numerosParitarios :: [Integer]
-- cuyo valor es la lista ordenada de todos los números paritarios. Por
-- ejemplo,
--    take 10 numerosParitarios  ==  [1,3,5,7,9,100,102,104,106,108]
--    numerosParitarios !! 10     ==  111
--    numerosParitarios !! 100    ==  290
--    numerosParitarios !! 1000   ==  11090
--    numerosParitarios !! 10000  ==  29091
-- ---------------------------------------------------------------------

numerosParitarios :: [Integer]
numerosParitarios =
  filter numeroParitario [1..]

numeroParitario :: Integer -> Bool
numeroParitario n =
  even (length ps) && odd (length is)
  where ns = show n
        ps = filter (`elem` "02468") ns
        is = filter (`elem` "13579") ns

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    subconjuntosSinConsecutivos :: [Int] -> [[Int]]
-- tal que (subconjuntosSinConsecutivos xs) es la lista de los
-- subconjuntos de la lista xs en los que no haya números consecutivos.
-- Por ejemplo,
--    subconjuntosSinConsecutivos [2..4]   ==  [[2,4],[2],[3],[4],[]]
--    subconjuntosSinConsecutivos [2,4,3]  ==  [[2,4],[2],[4],[3],[]]
--    subconjuntosSinConsecutivos [1..5]   ==
--       [[1,3,5],[1,3],[1,4],[1,5],[1],[2,4],[2,5],[2],[3,5],[3],[4],[5],[]]
--    subconjuntosSinConsecutivos [5,2,4,1,3]  ==
```

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

subconjuntosSinConsecutivos :: [Int] -> [[Int]]
subconjuntosSinConsecutivos [] = [[]]
subconjuntosSinConsecutivos (x:xs) =
  [x:ys | ys <- yss
        , x-1 `notElem` ys
        , x+1 `notElem`  ys]
  ++ yss
  where yss = subconjuntosSinConsecutivos xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    matrizBloque :: Int -> Int -> Int -> Int -> Matriz a -> Matriz a
-- tal que (matrizBloque i1 i2 j1 j2 m) es la submatriz de la matriz m
-- formada por todos los elementos en las filas desde la i1 hasta la i2
-- y en las columnas desde la j1 hasta la j2. Por ejemplo, si
--    m1 = array ((1,1),(9,9)) [((i,j),i*10+j) | i <- [1..9], j <- [1..9]]
-- entonces
--    matrizBloque 3 6 7 9 m1  ==
--       array ((1,1),(4,3)) [((1,1),37),((1,2),38),((1,3),39),
--                            ((2,1),47),((2,2),48),((2,3),49),
--                            ((3,1),57),((3,2),58),((3,3),59),
--                            ((4,1),67),((4,2),68),((4,3),69)]
--    matrizBloque 2 7 3 6 m1  ==
--       array ((1,1),(6,4)) [((1,1),23),((1,2),24),((1,3),25),((1,4),26),
--                            ((2,1),33),((2,2),34),((2,3),35),((2,4),36),
--                            ((3,1),43),((3,2),44),((3,3),45),((3,4),46),
--                            ((4,1),53),((4,2),54),((4,3),55),((4,4),56),
--                            ((5,1),63),((5,2),64),((5,3),65),((5,4),66),
--                            ((6,1),73),((6,2),74),((6,3),75),((6,4),76)]
-- ---------------------------------------------------------------------

type Matriz a = Array (Int,Int) a

m1 :: Matriz Int
m1 = array ((1,1),(9,9)) [((i,j),i*10+j) | i <- [1..9], j <- [1..9]]

matrizBloque :: Int -> Int -> Int -> Int -> Matriz a -> Matriz a
```

```
matrizBloque i1 i2 j1 j2 m
  | 1 <= i1 && i1 <= i2 && i2 <= p &&
    1 <= j1 && j1 <= j2 && j2 <= q =
      array ((1,1),(i2-i1+1,j2-j1+1))
            [((i,j),m!(i+i1-1,j+j1-1)) | i <- [1..i2-i1+1],
                                         j <- [1..j2-j1+1]]
  | otherwise = error "La operación no se puede realizar"
  where (_,(p,q)) = bounds m

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se dice que una operador @ es interno en un conjunto A
-- si al aplicar @ sobre elementos de A se obtiene como resultado
-- otro elemento de A. Por ejemplo, la suma es un operador interno en el
-- conjunto de los números naturales pares. La clausura de un conjunto A
-- con respecto a un operador @ es el menor conjunto B tal que A está
-- contenido en B y el operador @ es interno en el conjunto B. Por
-- ejemplo, la clausura del conjunto {2} con respecto a la suma es el
-- conjunto de los números pares positivos:
--    {2, 4, 6, 8, ...} = {2*k | k <- [1..]}
--
-- Definir la función
--    clausuraOperador :: (Int -> Int -> Int) -> S.Set Int -> S.Set Int
-- tal que (clausuraOperador op xs) es la clausura del conjunto xs con
-- respecto a la operación op. Por ejemplo,
--    clausuraOperador gcd (S.fromList [6,9,10])     ==
--       fromList [1,2,3,6,9,10]
--    clausuraOperador gcd (S.fromList [42,70,105])  ==
--       fromList [7,14,21,35,42,70,105]
--    clausuraOperador lcm (S.fromList [6,9,10])     ==
--       fromList [6,9,10,18,30,90]
--    clausuraOperador lcm (S.fromList [2,3,5,7])    ==
--       fromList [2,3,5,6,7,10,14,15,21,30,35,42,70,105,210]
-- ---------------------------------------------------------------------

clausuraOperador :: (Int -> Int -> Int) -> S.Set Int -> S.Set Int
clausuraOperador op =
  until (\ xs -> null [(x,y) | x <- S.elems xs,
                               y <- S.elems xs,
                               S.notMember (op x y) xs])
        (\ xs -> S.union xs (S.fromList [op x y | x <- S.elems xs,
```

```
                                                    y <- S.elems xs]))
```

2.5. Examen 5 (24 de abril de 2017)

```
-- Informática (1º del Grado en Matemáticas - Grupo 2)
-- 5º examen de evaluación continua (24 de abril de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Array
import qualified Data.Matrix as M
import Data.Number.CReal
import I1M.PolOperaciones
import I1M.Grafo

-- ---------------------------------------------------------------------
-- Ejercicio 1. El cociente entre dos números se puede calcular con el
-- operador de división (/), obteniendo tantos decimales como nos
-- permita la representación de los números reales. Por ejemplo,
--    123/11 = 11.181818181818182
--    123/13 = 9.461538461538462
--    123/15 = 8.2
--    123/17 = 7.235294117647059
--    123/19 = 6.473684210526316
--
-- Definir la función:
--    division :: Integer -> Integer -> [Integer]
-- tal que (division n m) es la lista (posiblemente infinita) cuyo
-- primer elemento es el cociente entero de la división del número
-- natural n entre el número natural m; y los demás elementos son todas
-- las cifras de la parte decimal de la división de n entre m. Por
-- ejemplo,
--    take 10 (division 123 11)  ==  [11,1,8,1,8,1,8,1,8,1]
--    take 10 (division 123 13)  ==  [9,4,6,1,5,3,8,4,6,1]
--    division 123 15            ==  [8,2]
--    take 30 (division 123 17)  ==
```

```
--       [7,2,3,5,2,9,4,1,1,7,6,4,7,0,5,8,8,2,3,5,2,9,4,1,1,7,6,4,7,0]
--    take 30 (division 123 19)  ==
--       [6,4,7,3,6,8,4,2,1,0,5,2,6,3,1,5,7,8,9,4,7,3,6,8,4,2,1,0,5,2]
-- ---------------------------------------------------------------------

division :: Integer -> Integer -> [Integer]
division x y
  | r == 0    = [q]
  | otherwise = q : division (r*10) y
  where (q,r) = quotRem x y

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    mapPol :: (Num a, Eq a) =>
--              (a -> a) -> (Int -> Int) -> Polinomio a -> Polinomio a
-- tal que (mapPol f g p) es el polinomio construido a partir de los
-- términos del polinomio p, tomando como nuevos coeficientes el
-- resultado de evaluar la función f sobre los coeficientes de los
-- términos de p y tomando como nuevos grados el resultado de evaluar la
-- función g sobre los grados de los términos de p. Por ejemplo, si p1
-- es el polinomio definido por
--    p1 :: Polinomio Int
--    p1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
-- entonces
--    p1                          =>  3*x^4 + -5*x^2 + 3
--    mapPol (+1) (+1) p1         =>  4*x^5 + -4*x^3 + 4*x
--    mapPol (+2) (*2) p1         =>  5*x^8 + -3*x^4 + 5
--    mapPol (+(-2)) (*2) p1      =>  x^8 + -7*x^4 + 1
--    mapPol (+(-2)) (*(-2)) p1   =>  -5
--    mapPol (+0) (*0) p1         =>  1
-- Nota: Si al aplicar una función al exponente el resultado es
-- negativo, se cambia el resultado por cero.
-- ---------------------------------------------------------------------

p1 :: Polinomio Int
p1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))

mapPol :: (Num a, Eq a) =>
          (a -> a) -> (Int -> Int) -> Polinomio a -> Polinomio a
mapPol f g p
```

```
  | esPolCero p = polCero
  | otherwise   = consPol (max 0 (g n)) (f b) (mapPol f g r)
  where n = grado p
        b = coefLider p
        r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 3. Dado un grafo no dirigido G, decimos que tres vértices
-- (v1,v2,v3) de G forman un triángulo si hay en G una arista entre
-- cada par de ellos; es decir, una arista de v1 a v2, otra de v2 a v3 y
-- una tercera de v3 a v1.
--
-- Definir la función
--    verticesSinTriangulo :: (Ix v,Num p) => Grafo v p -> [v]
-- tal que (verticesSinTriangulo g) es la lista de todos los vértices
-- del grafo g que no están en ningún triángulo. Por ejemplo, si g1 y g2
-- son los grafos definidos por
--    g1, g2 :: Grafo Int Int
--    g1 = creaGrafo ND (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
--                             (5,4,0),(6,2,0),(6,4,0),(6,5,0)]
--    g2 = creaGrafo ND (1,6) [(1,3,0),(1,5,0),(3,5,0),(5,6,0),
--                             (2,4,0),(2,6,0),(4,6,0)]
-- entonces,
--    verticesSinTriangulo g1  ==  [1,2,3]
--    verticesSinTriangulo g2  ==  []
-- ---------------------------------------------------------------------

g1, g2 :: Grafo Int Int
g1 = creaGrafo ND (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
                         (5,4,0),(6,2,0),(6,4,0),(6,5,0)]
g2 = creaGrafo ND (1,6) [(1,3,0),(1,5,0),(3,5,0),(5,6,0),
                         (2,4,0),(2,6,0),(4,6,0)]

verticesSinTriangulo :: (Ix v,Num p) => Grafo v p -> [v]
verticesSinTriangulo g =
  [x | x <- nodos g, sinTriangulo g x]

-- (sinTriangulo g x) se verifica si no hay ningún triángulo en el grafo
-- g que contenga al vértice x. Por ejemplo,
--    sinTriangulo g1 2  ==  True
```

```
--    sinTriangulo g1 5  ==  False
sinTriangulo :: (Ix v,Num p) => Grafo v p -> v -> Bool
sinTriangulo g x = null (triangulos g x)

-- (triangulos g x) es la lista de los pares de vértices (y,z) tales que
-- (x,y,z) es un triángulo en el grafo g. Por ejemplo,
--    triangulos g1 5  ==  [(6,4),(4,6)]
--    triangulos g1 2  ==  []
triangulos :: (Ix v,Num p) => Grafo v p -> v -> [(v,v)]
triangulos g x =
  [(y,z) | y <- ns
         , z <- ns
         , aristaEn g (y,z)]
  where ns = adyacentes g x

-- ---------------------------------------------------------------------
-- Ejercicio 4. El recorrido en ZigZag de una matriz consiste en pasar
-- de la primera fila hasta la última, de izquierda a derecha en las
-- filas impares y de derecha a izquierda en las filas pares, como se
-- indica en la figura.
--
--          /             \
--          | 1 -> 2 -> 3 |
--          |           | |
--          |           v |
--          | 4 <- 5 <- 6 |   =>  Recorrido ZigZag: [1,2,3,6,5,4,7,8,9]
--          | |           |
--          | v           |
--          | 7 -> 8 -> 9 |
--          \             /
--
-- Definir la función
--    recorridoZigZag :: M.Matrix a -> [a]
-- tal que (recorridoZigZag m) es la lista con los elementos de la
-- matriz m cuando se recorre esta en ZigZag. Por ejemplo,
--    ghci> recorridoZigZag (M.fromLists [[1,2,3],[4,5,6],[7,8,9]])
--    [1,2,3,6,5,4,7,8,9]
--    ghci> recorridoZigZag (M.fromLists [[1,2],[3,4],[5,6],[7,8]])
--    [1,2,4,3,5,6,8,7]
--    ghci> recorridoZigZag (M.fromLists [[1,2,3,4],[5,6,7,8],[9,10,11,12]])
```

```
--    [1,2,3,4,8,7,6,5,9,10,11,12]
-- ----------------------------------------------------------------------

recorridoZigZag :: M.Matrix a -> [a]
recorridoZigZag m =
  concat [f xs | (f,xs) <- zip (cycle [id,reverse]) (M.toLists m)]
```

2.6. Examen 6 (12 de junio de 2017)

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

2.7. Examen 7 (29 de junio de 2017)

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

2.8. Examen 8 (8 de septiembre de 2017)

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

2.9. Examen 9 (21 de noviembre de 2017)

El examen es común con el del grupo 1 (ver página 65).

3

Exámenes del grupo 3

Antonia M. Chávez

3.1. Examen 1 (28 de octubre de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 1º examen de evaluación continua (28 de octubre de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Decimos el número y es colega de x si la suma de los
-- dígitos de x coincide con el producto de los dígitos de y. Por
-- ejemplo, 24 es colega de 23.
--
-- Definir la función
--    esColega :: Int -> Int -> Bool
-- tal que (esColega x y) que se verifique si y es colega de x. Por
-- ejemplo,
--    esColega 24 23   ==  True
--    esColega 23 24   ==  False
--    esColega 24 611  ==  True
-- ---------------------------------------------------------------------
```

```haskell
esColega :: Int -> Int -> Bool
esColega x y = sum (digitos x) == product (digitos y)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    colegas3 :: Int -> [Int]
-- tal que (colegas3 x) es la lista de los colegas de x con tres
-- dígitos. Por ejemplo,
--    colegas3 24  ==  [116,123,132,161,213,231,312,321,611]
-- ---------------------------------------------------------------------

-- 1ª solución
colegas3 :: Int -> [Int]
colegas3 x = [y | y <- [100 .. 999], esColega x y]

-- 2ª solución
colegas3b :: Int -> [Int]
colegas3b x = filter (x `esColega`) [100 .. 999]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    colegasN :: Int -> Int -> [Int]
-- tal que (colegasN x n) es la lista de colegas de x con menos de n
-- dígitos. Por ejemplo,
--    ghci> colegasN 24 4
--    [6,16,23,32,61,116,123,132,161,213,231,312,321,611]
-- ---------------------------------------------------------------------

-- 1ª solución
colegasN :: Int -> Int -> [Int]
colegasN x n = [y | y <- [1..10^(n-1)-1], esColega x y]

-- 2ª solución
colegasN2 :: Int -> Int -> [Int]
colegasN2 x n = filter (x `esColega`) [1..10^(n-1)-1]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, usando condicionales sin guardas, la función
--    divideMitad1 :: [a] -> ([a],[a])
-- tal que (divideMitad1 xs) es el par de lista de igual longitud en que
-- se divide la lista xs (eliminando el elemento central si la longitud
-- de xs es impar). Por ejemplo,
--    divideMitad1 [2,3,5,1]  ==  ([2,3],[5,1])
--    divideMitad1 [2,3,5]    ==  ([2],[5])
--    divideMitad1 [2]        ==  ([],[])
-- ---------------------------------------------------------------------

divideMitad1 :: [a] -> ([a], [a])
divideMitad1 xs =
  if even (length xs)
  then (take n xs, drop n xs)
  else (take n xs, drop (n+1) xs)
  where n = div (length xs) 2

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, usando guardas sin condicionales, la función
--    divideMitad2 :: [a] -> ([a],[a])
-- tal que (divideMitad2 xs) es el par de lista de igual longitud en que
-- se divide la lista xs (eliminando el elemento central si la longitud
-- de xs es impar). Por ejemplo,
--    divideMitad2 [2,3,5,1]  ==  ([2,3],[5,1])
--    divideMitad2 [2,3,5]    ==  ([2],[5])
--    divideMitad2 [2]        ==  ([],[])
-- ---------------------------------------------------------------------

divideMitad2 :: [a] -> ([a], [a])
divideMitad2 xs
  | even (length xs) = (take n xs, drop n xs)
  | otherwise        = (take n xs, drop (n+1) xs)
  where n = div (length xs) 2

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que las funciones
-- divideMitad1 y divideMitad2 son equivalentes para listas de números
-- enteros.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_divideMitad :: [Int] -> Bool
prop_divideMitad xs =
  divideMitad1 xs == divideMitad2 xs

-- La comprobacion es:
--    ghci> quickCheck prop_divideMitad
--    +++ OK, passed 100 tests.

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un elemento de una lista es un pivote si ninguno de
-- los siguientes en la lista es mayor que él.
--
```

```haskell
-- Definirla función
--    pivotes :: Ord a => [a] -> [a]
-- tal que (pivotes xs) es la lista de los pivotes de xs. Por
-- ejemplo,
--    pivotes [80,1,7,8,4]  ==  [80,8,4]
-- ---------------------------------------------------------------------

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

pivotes :: Ord a => [a] -> [a]
pivotes xs = [x | (x,n) <- zip xs [1 ..], esMayor x (drop n xs)]

-- (esMayor x ys) se verifica si x es mayor que todos los elementos de
-- ys. Por ejemplo,
esMayor :: Ord a =>  a -> [a] -> Bool
esMayor x xs = and [x > y | y <- xs]

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

pivotes2 :: Ord a => [a] -> [a]
pivotes2 [] = []
pivotes2 (x:xs) | esMayor2 x xs = x : pivotes2 xs
                | otherwise     = pivotes2 xs

esMayor2 :: Ord a =>  a -> [a] -> Bool
esMayor2 x xs = all (x>) xs

-- 3ª definición
-- =============

pivotes3 :: Ord a => [a] -> [a]
pivotes3 xs =
  [y | (y:ys) <- init (tails xs)
     , y `esMayor` ys]
```

3.2. Examen 2 (2 de diciembre de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (2 de diciembre de 2016)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    elevaSumaR  :: [Integer] -> [Integer] -> Integer
-- tal que (elevaSuma xs ys) es la suma de la potencias de los elementos
-- de xs elevados a los elementos de ys respectivamente. Por ejemplo,
--    elevaSuma  [2,6,9]  [3,2,0]   = 8 + 36 + 1    = 45
--    elevaSuma  [10,2,5] [3,4,1,2] = 1000 + 16 + 5 = 1021
-- ---------------------------------------------------------------------

-- 1ª definición (Por recursión)
elevaSumaR :: [Integer] -> [Integer] -> Integer
elevaSumaR [] _ = 0
elevaSumaR _ [] = 0
elevaSumaR (x:xs) (y:ys) = x^y + elevaSumaR xs ys

-- 2ª definición (Por plegado a la derecha)
elevaSumaPR :: [Integer] -> [Integer] -> Integer
elevaSumaPR xs ys = foldr f 0 (zip xs ys)
    where f (a,b) ys = a^b + ys

-- 3ª definición (Por comprensión)
elevaSumaC :: [Integer] -> [Integer] -> Integer
elevaSumaC xs ys = sum [a^b | (a,b) <- zip xs ys]

-- 4ª definición (Con orden superior)
elevaSumaS  :: [Integer] -> [Integer] -> Integer
elevaSumaS xs ys = sum (map f (zip xs ys))
  where f (x,y) = x^y
```

```haskell
-- 5ª definición (Con acumulador)
elevaSumaA :: [Integer] -> [Integer] -> Integer
elevaSumaA xs ys = aux (zip xs ys) 0
  where aux [] ac = ac
        aux ((a,b):ps) ac = aux ps (a^b + ac)

-- 6ª definición (Por plegado a la izquierda)
elevaSumaPL :: [Integer] -> [Integer] -> Integer
elevaSumaPL xs ys = foldl f 0 (zip xs ys)
    where f ac (a,b) = ac + (a^b)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una lista de números es prima si la mayoría de sus
-- elementos son primos.
--
-- Definir la función
--    listaPrima:: [Int] -> Bool
-- tal que (listaPrima xs) se verifica si xs es prima. Por ejemplo,
--    listaPrima [3,2,6,1,5,11,19] == True
--    listaPrima [80,12,7,8,3]     == False
--    listaPrima [1,7,8,4]         ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
listaPrimaC :: [Int] -> Bool
listaPrimaC xs =
  length [x | x <- xs, esPrimo x] > length xs `div` 2

esPrimo :: Int -> Bool
esPrimo n = factores n == [1,n]

factores :: Int -> [Int]
factores n = [x | x <-[1 .. n], mod n x == 0]

-- 2ª definición (con filter)
listaPrimaS :: [Int] -> Bool
listaPrimaS xs =
  length (filter esPrimo xs) > length xs `div` 2
```

```haskell
-- 3ª definición (usando recursión):
listaPrimaR :: [Int] -> Bool
listaPrimaR xs = length (aux xs) > length xs `div` 2
  where  aux [] = []
         aux (x:xs) | esPrimo x = x : aux xs
                    | otherwise = aux xs

-- 4ª definición (con plegado a la derecha):
listaPrimaPR :: [Int] -> Bool
listaPrimaPR xs = length (foldr f [] xs) > length xs `div` 2
  where f x ys | esPrimo x = x:ys
               | otherwise = ys

-- 5ª definición (usando acumuladores)
listaPrimaA :: [Int] -> Bool
listaPrimaA xs = length (aux xs []) > length xs `div` 2
  where aux [] ac = ac
        aux (y:ys) ac | esPrimo y = aux ys (ac ++ [y])
                      | otherwise = aux ys ac

-- 6ª definición (usando plegado a la izquierda):
listaPrimaPL :: [Int] -> Bool
listaPrimaPL xs = length (foldl g [] xs) > length xs `div` 2
  where g acum prim | esPrimo prim = acum ++ [prim]
                    | otherwise    = acum
```

3.3. Examen 3 (31 de enero de 2017)

El examen es común con el del grupo 4 (ver página 117).

3.4. Examen 4 (13 de marzo de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 4º examen de evaluación continua (13 de marzo de 2017)
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número primo se dice que es doble primo si se le
-- puede anteponer otro primo con igual número de dígitos obteniéndose
-- un número primo. Por ejemplo, el 3 es dobleprimo ya que 23,53,73 son
-- primos. También lo es 19 ya que, por ejemplo, 1319 es primo.
--
-- Definir la función
--    prefijoPrimo :: Integer -> [Integer]
-- tal que (prefijoPrimo x) es la lista de los primos de igual longitud
-- que x tales que al anteponerlos a x se obtiene un primo. Por ejemplo,
--    prefijoPrimo  3 = [2,5,7]
--    prefijoPrimo 19 = [13,31,37,67,79,97]
--    prefijoPrimo  2 = []
-- ---------------------------------------------------------------------

prefijoPrimo :: Integer -> [Integer]
prefijoPrimo x =
  [y | y <- [10^(n-1)..10^n]
     , isPrime y
     , isPrime (pega y x)]
  where n = numDigitos x

pega :: Integer -> Integer -> Integer
pega a b = read (show a ++ show b)

numDigitos :: Integer -> Int
numDigitos y = length (show y)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la sucesión
--    doblesprimos :: [Integer]
-- tal que sus elementos son los dobles primos. Por ejemplo,
--    take 15 doblesprimos == [3,7,11,13,17,19,23,29,31,37,41,43,47,53,59]
--    doblesprimos !! 500  == 3593
```

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

doblesPrimos :: [Integer]
doblesPrimos = [x | x <- primes, esDoblePrimo x]

esDoblePrimo :: Integer -> Bool
esDoblePrimo x = isPrime x && not (null (prefijoPrimo x))

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los árboles binarios con
-- elementos en las hojas y en los nodos mediante el tipo de dato
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving Show
-- Por ejemplo,
--    ej1 :: Arbol Integer
--    ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))
--
-- Definir la función
--    numeros :: Arbol Integer-> Array Integer String -> Arbol String
-- tal que (numeros a v) es el árbol obtenido al sustituir los números
-- del árbol a por su valor en el vector v. Por ejemplo, para
--    v1 = listArray (0,10)
--                   ["cero","uno","dos","tres","cuatro","cinco","seis",
--                    "siete","ocho","nueve", " diez"]
--    v2 = listArray (0,20)
--                   [if even x then "PAR" else "IMPAR" | x <- [0..19]]
-- tenemos:
--    numeros ej1 v1 = N "cinco"
--                       (N "dos"
--                          (H "uno")
--                          (H "dos"))
--                       (N "tres"
--                          (H "cuatro")
--                          (H "dos"))
--    numeros ej1 v2 = N "IMPAR"
--                       (N "PAR" (H "IMPAR") (H "PAR"))
--                       (N "IMPAR" (H "PAR") (H "PAR"))
-- ---------------------------------------------------------------------
```

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

ej1 :: Arbol Integer
ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))

v1 = listArray (0,10)
               ["cero","uno","dos","tres","cuatro","cinco","seis",
                "siete","ocho","nueve", " diez"]
v2 = listArray (0,20)
               [if even x then "PAR" else "IMPAR" | x <- [0..19]]

numeros :: Arbol Integer -> Array Integer String -> Arbol String
numeros (H x) v     = H (v!x)
numeros (N x i d) v = N (v!x) (numeros i v) (numeros d v)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Observemos la siguiente secuencia de matrices:
--
-- (1 2) pos (1,1) (9 2) pos (1,2) (9 16) pos (2,1) (9 16) pos(2,2)(10 16)
-- (3 4)  ->       (3 4)   ->      (3  4)  ->       (29 4)    ->   (29 54)
--   m0              m1              m2               m3             m4
--
-- Una matriz en el paso n se obtiene cambiando en la matriz del
-- paso anterior el elemento n-ésimo por la suma de sus vecinos.
-- La idea es
--    m0 = m
--    m1 = es m  pero en (1,1) tiene la suma de sus vecinos en m,
--    m2 = es m1 pero en (1,2) tiene la suma de sus vecinos en m1,
--    m3 = es m2 pero en (2,1) tiene la suma de sus vecinos en m2,
--    m4 = es m3 pero en (2,2) tiene la suma de sus vecinos en m3.
--
-- Definir la función
--    sumaVecinos :: (Integer,Integer) ->
--                   Array (Integer, Integer) Integer ->
--                   Integer
-- tal que (sumaVecinos x p) es la suma de los vecinos del elemento de p
-- que está en la posición x. En el ejemplo,
--    sumaVecinos (2,1) m2 == 29
```

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

sumaVecinos :: (Int,Int) -> Array (Int, Int) Int -> Int
sumaVecinos (i,j) p =
  sum [p!(i+a,j+b) | a <- [-1..1]
                   , b <- [-1..1]
                   , a /= 0 || b /= 0
                   , inRange (bounds p) (i+a,j+b)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    transformada :: Array (Integer, Integer) Integer
--                    -> Int
--                    -> Array (Integer, Integer) Integer
-- tal que (transformada p n) es la matriz que se obtiene en el paso n a
-- partir de la matriz inicial p. Es decir, en el ejemplo de arriba,
--    transformada m 0 = m0
--    transformada m 1 = m1
-- ---------------------------------------------------------------------

transformada :: Array (Int, Int) Int -> Int -> Array (Int, Int) Int
transformada p 0 = p
transformada p n =
  transformada p (n-1)
  // [(indices p !! (n-1),
       sumaVecinos (indices p !! (n-1)) (transformada p (n-1)))]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    secuencia :: Array (Integer, Integer) Integer
--                 -> [Array (Integer, Integer) Integer]
-- tal que (secuencia p) es la lista que contiene todas las matrices que
-- se obtienen a partir de la matriz inicial p. En el ejemplo,
-- (secuencia m) será la lista [m0, m1, m2, m3, m4]
--    ghci> map elems (secuencia (listArray ((1,1),(2,2))[1,2,3,4]))
--    [[1,2,3,4],[9,2,3,4],[9,16,3,4],[9,16,29,4],[9,16,29,54]]
-- ---------------------------------------------------------------------

secuencia :: Array (Int, Int) Int -> [Array (Int, Int) Int]
secuencia p = [transformada p n | n <- [0 .. rangeSize (bounds p)]]
```

3.5. Examen 5 (24 de abril de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (23 de abril de 2017)
-- ---------------------------------------------------------------------

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

import Data.Array
import Data.List (nub, sort)
import Data.Numbers.Primes (isPrime, primeFactors)
import I1M.Pol
import I1M.Cola
import Test.QuickCheck

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

type Vector a = Array Int a

posMaxVec :: Ord a => Vector a -> [Int]
posMaxVec v =
  [k | k <- [1..n]
```

```
    , and [v!j < v!k | j <- posAdyacentes k]]
  where (_,n)           = bounds v
        posAdyacentes k = [k-1 | k > 1] ++ [k+1 | k < n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Los árboles binarios se pueden representar con el tipo
-- de dato algebraico
--    data Arbol a = H | N a (Arbol a) (Arbol a)
--                   deriving Show
-- Por ejemplo, los árboles

--         9                9
--        / \              /
--       /   \            /
--      8     6          8
--     / \   / \        / \
--    3   2 4   5      3   2
-- se pueden representar por
--    ej1, ej2:: Arbol Int
--    ej1 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--    ej2 = N 9 (N 8 (N 3 H H) (N 2 H H)) H
--
-- Para indicar las posiciones del árbol se define el tipo
--    type Posicion = [Direccion]
-- donde
--    data Direccion = D | I  deriving Eq
-- representa un movimiento hacia la derecha (D) o a la izquierda. Por
-- ejemplo, las posiciones de los elementos del ej1 son
-- + el 9 tiene posición []
-- + el 8 tiene posición [I]
-- + el 3 tiene posición [I,I]
-- + el 2 tiene posición [I,D]
-- + el 6 tiene posición [D]
-- + el 4 tiene posición [D,I]
-- + el 5 tiene posición [D,D]
--
-- Definir la función
--    sustitucion :: Posicion -> a -> Arbol a -> Arbol a
-- tal que (sustitucion ds z x) es el árbol obtenido sustituyendo el
-- elemento del arbol x en la posición ds por z. Por ejemplo,
```

```
--    ghci> sustitucion [I,D] 7 ej1
--    N 9 (N 8 (N 3 H H) (N 7 H H)) (N 6 (N 4 H H) (N 5 H H))
--    ghci> sustitucion [D,D] 7 ej1
--    N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 7 H H))
--    ghci> sustitucion [I] 7 ej1
--    N 9 (N 7 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--    ghci> sustitucion [] 7 ej1
--    N 7 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
-- ---------------------------------------------------------------------

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

ej1, ej2:: Arbol Int
ej1 = N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
ej2 = N 9 (N 8 (N 3 H H) (N 2 H H)) H

type Posicion = [Direccion]

data Direccion = D | I  deriving Eq

sustitucion :: Posicion  -> a -> Arbol a -> Arbol a
sustitucion (I:ds) z (N x i d) = N x (sustitucion ds z i) d
sustitucion (D:ds) z (N x i d) = N x i (sustitucion ds z d)
sustitucion []     z (N _ i d) = N z i d
sustitucion _      _ H         = H

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un número n es especial si al unir los dígitos de sus
-- factores primos, se obtienen exactamente los dígitos de n, aunque
-- puede ser en otro orden. Por ejemplo, 1255 es especial, pues los
-- factores primos de 1255 son 5 y 251.
--
-- Definir la función
--    esEspecial :: Integer -> Bool
-- tal que (esEspecial n) se verifica si un número n es especial. Por
-- ejemplo,
--    esEspecial 1255 == True
--    esEspecial 125  == False
--    esEspecial 132  == False
```

```haskell
-- Comprobar con QuickCheck que todo número primo es especial.
--
-- Calcular los 5 primeros números especiales que no son primos.
-- ---------------------------------------------------------------------

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

-- La propiedad es
prop_primos :: Integer -> Property
prop_primos n =
  isPrime (abs n) ==> esEspecial (abs n)

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un polinomio no nulo con coeficientes enteros es primo
-- si el máximo común divisor de sus coeficientes es 1.
--
-- Definir la función
--    primo :: Polinomio Int -> Bool
-- tal que (primo p) se verifica si el polinomio p es primo. Por ejemplo,
--    ghci> primo (consPol 6 2 (consPol  5 3 (consPol 4 8 polCero)))
--    True
--    ghci> primo (consPol 6 2 (consPol  5 6 (consPol 4 8 polCero)))
--    False
-- ---------------------------------------------------------------------

primo :: Polinomio Int -> Bool
primo p = foldl1 gcd (coeficientes p) == 1

coeficientes :: (Num a, Eq a) => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <- [n,n-1..0]]
```

```
  where n = grado p

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

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    penultimo :: Cola a -> a
-- tal que (penultimo c) es el penúltimo elemento de la cola c. Si la
-- cola está vacía o tiene un sólo elemento, dará el error
-- correspondiente, "cola vacia" o bien "cola unitaria". Por ejemplo,
--    ghci> penultimo (inserta 3 (inserta 2 vacia))
--    2
--    ghci> penultimo (inserta 5 (inserta 3 (inserta 2 vacia)))
--    3
--    ghci> penultimo vacia
--    *** Exception: cola vacia
--    ghci> penultimo (inserta 2 vacia)
--    *** Exception: cola unitaria
-- ---------------------------------------------------------------------

penultimo ::  Cola a -> a
penultimo c = aux (reverse (cola2Lista c))
  where aux []      = error "cola vacia"
        aux [_]     = error "cola unitaria"
        aux (_:x:_) = x

-- (cola2Lista c) es la lista formada por los elementos de p. Por
-- ejemplo,
--    cola2Lista c2 == [17,14,11,8,5,2]
cola2Lista :: Cola a -> [a]
cola2Lista c
  | esVacia c = []
  | otherwise = pc : cola2Lista rc
  where pc = primero c
        rc = resto c
```

3.6. Examen 6 (12 de junio de 2017)

El examen es común con el del grupo 4 (ver página 136).

3.7. Examen 7 (29 de junio de 2017)

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

3.8. Examen 8 (8 de septiembre de 2017)

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

3.9. Examen 9 (21 de noviembre de 2017)

El examen es común con el del grupo 1 (ver página 65).

4

Exámenes del grupo 4

María J. Hidalgo

4.1. Examen 1 (3 de noviembre de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (3 de noviembre de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La intersección de dos listas xs, ys es la lista
-- formada por los elementos de xs que también pertenecen a ys.
--
-- Definir, por comprensión, la función
--    interseccionC :: Eq a => [a] -> [a] -> [a]
-- tal que (interseccionC xs ys) es la intersección de xs e ys. Por
-- ejemplo,
--    interseccionC [2,7,4,1] [1,3,6] == [1]
--    interseccionC [2..10] [3..12]   == [3,4,5,6,7,8,9,10]
--    interseccionC [2..10] [23..120] == []
-- ---------------------------------------------------------------------
```

```haskell
interseccionC :: Eq a => [a] -> [a] -> [a]
interseccionC xs ys = [x | x <- xs, x `elem` ys]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    interseccionR :: Eq a => [a] -> [a] -> [a]
-- tal que (interseccionR xs ys) es la intersección de xs e ys. Por
-- ejemplo,
--    interseccionR [2,7,4,1] [1,3,6] == [1]
--    interseccionR [2..10] [3..12]   == [3,4,5,6,7,8,9,10]
--    interseccionR [2..10] [23..120] == []
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuikCheck que ambas definiciones
-- coinciden.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_interseccion :: (Eq a) => [a] -> [a] -> Bool
prop_interseccion xs ys =
  interseccionC xs ys == interseccionR xs ys

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. El doble factorial de un número n se define por
--    n!! = n*(n-2)* ... * 3 * 1, si n es impar
--    n!! = n*(n-2)* ... * 4 * 2, si n es par
--    1!! = 1
--    0!! = 1
-- Por ejemplo,
```

```
--    8!! = 8*6*4*2 = 384
--    9!! = 9*7*5*3*1 = 945
--
-- Definir, por comprensión, la función
--    dobleFactorialC :: Integer -> Integer
-- tal que (dobleFactorialC n) es el doble factorial de n. Por ejemplo,
--    dobleFactorialC 8 == 384
--    dobleFactorialC 9 == 945
-- ---------------------------------------------------------------------

dobleFactorialC :: Integer -> Integer
dobleFactorialC n = product [n,n-2..2]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    dobleFactorialR :: Integer -> Integer
-- tal que (dobleFactorialR n) es el doble factorial de n. Por ejemplo,
--    dobleFactorialR 8 == 384
--    dobleFactorialR 9 == 945
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuikCheck que ambas definiciones
-- coinciden para n >= 0.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_dobleFactorial :: Integer -> Property
prop_dobleFactorial n =
  n >= 0 ==> dobleFactorialC n == dobleFactorialR n

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3. Un número n es especial si es mayor que 9 y la suma de
-- cualesquiera dos dígitos consecutivos es un número primo. Por
-- ejemplo, 4116743 es especial pues se cumple que la suma de dos
-- dígitos consecutivos es un primo, ya que 4+1, 1+1, 1+6, 6+7, 7+4 y
-- 4+3 son primos.
--
-- Definir una función
--    especial :: Integer -> Bool
-- tal que (especial n) reconozca si n es especial. Por ejemplo,
--    especial 4116743  == True
--    especial 41167435 == False
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

especial1 :: Integer -> Bool
especial1 n =
  n > 9 && and [primo (x+y) | (x,y) <- zip xs (tail xs)]
  where xs = digitos n

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


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

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

-- 2ª definición
```

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

especial2 :: Integer -> Bool
especial2 n = n > 9 && all primo (zipWith (+) xs (tail xs))
  where xs = digitos n

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    acumula :: Num a => [a] -> [a]
-- tal que (acumula xs) es la lista obtenida sumando a cada elemento de
-- xs todos los posteriores. Por ejemplo:
--    acumula [1..5]    == [15,14,12,9,5]
--    acumula [3,5,1,2] == [11,8,3,2]
--    acumula [-1,0]    == [-1,0]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
acumula :: Num a => [a] -> [a]
acumula []     = []
acumula (x:xs) = (x + sum xs): acumula xs

-- 2ª definición (por comprensión):
acumula2 :: Num a => [a] -> [a]
acumula2 xs = [sum (drop k xs) | k <- [0..length xs - 1]]

-- 3ª definición (por composición):
acumula3 :: Num a => [a] -> [a]
acumula3 = init . map sum . tails
```

4.2. Examen 2 (1 de diciembre de 2016)

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

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    menorFactorialDiv :: Integer -> Integer
-- tal que (menorFactorialDiv n) es el menor m tal que n divide a m!
-- Por ejemplo,
```

```haskell
--    menorFactorialDiv 10    == 5
--    menorFactorialDiv 25    == 10
--    menorFactorialDiv 10000 == 20
--    menorFactorialDiv1 1993 == 1993
--
-- ¿Cuáles son los números n tales que (menorFactorialDiv n) == n?
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

menorFactorialDiv1 :: Integer -> Integer
menorFactorialDiv1 n =
  head [x | x <- [1..], product [1..x] `mod` n == 0]

-- Los números n tales que
--    (menorFactorialDiv n) == n
-- son el 1, el 4 y los números primos, lo que permite la segunda
-- definición.

-- 2ª solución
-- ===========

menorFactorialDiv2 :: Integer -> Integer
menorFactorialDiv2 n
  | isPrime n = n
  | otherwise = head [x | x <- [1..], product [1..x] `mod` n == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Dado un entero positivo n, consideremos la suma de los
-- cuadrados de sus divisores. Por ejemplo,
--    f(10) = 1 + 4 + 25 + 100 = 130
--    f(42) = 1 + 4 +  9 +  36 + 49 + 196 + 441 + 1764 = 2500
-- Decimos que n es un número "sumaCuadradoPerfecto" si f(n) es un
-- cuadrado perfecto. En los ejemplos anteriores, 42 es
-- sumaCuadradoPerfecto y 10 no lo es.
--
-- Definir la función
--    sumaCuadradoPerfecto:: Int -> Bool
-- tal que (sumaCuadradoPerfecto x) se verifica si x es un número es
```

```haskell
-- sumaCuadradoPerfecto. Por ejemplo,
--    sumaCuadradoPerfecto 42  ==  True
--    sumaCuadradoPerfecto 10  ==  False
--    sumaCuadradoPerfecto 246 ==  True
--
-- Calcular todos los números sumaCuadradoPerfectos de tres cifras.
-- ----------------------------------------------------------------------

-- 1ª definición
-- =============

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

-- 2ª definición
-- =============

sumaCuadradoPerfecto2 :: Int -> Bool
sumaCuadradoPerfecto2 =
  esCuadrado . sum . map (^2) . divisores

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

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

-- El cálculo es
--    ghci> filter sumaCuadradoPerfecto [100..999]
--    [246,287,728]

-- ----------------------------------------------------------------------
-- Ejercicio 3.1. Un lista de números enteros se llama alternada si sus
```

```haskell
-- elementos son alternativamente par/impar o impar/par.
--
-- Definir la función
--    alternada :: [Int] -> Bool
-- tal que (alternada xs) se verifica si xs es una lista alternada. Por
-- ejemplo,
--    alternada [1,2,3]     == True
--    alternada [1,2,3,4]   == True
--    alternada [8,1,2,3,4] == True
--    alternada [8,1,2,3]   == True
--    alternada [8]         == True
--    alternada [7]         == True
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

alternada1 :: [Int] -> Bool
alternada1 (x:y:xs)
  | even x    = odd y  && alternada1 (y:xs)
  | otherwise = even y && alternada1 (y:xs)
alternada1 _ = True

-- 2ª solución
-- ===========

alternada2 :: [Int] -> Bool
alternada2 xs = all odd (zipWith (+) xs (tail xs))

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Generalizando, una lista es alternada respecto de un
-- predicado p si sus elementos verifican alternativamente el predicado
-- p.
--
-- Definir la función
--    alternadaG :: (a -> Bool) -> [a] -> Bool
-- tal que (alternadaG p xs) compruebe se xs es una lista alternada
-- respecto de p. Por ejemplo,
--    alternadaG (>0) [-2,1,3,-9,2]  == False
--    alternadaG (>0) [-2,1,-3,9,-2] == True
```

```
--    alternadaG (<0) [-2,1,-3,9,-2] == True
--    alternadaG even [8,1,2,3]      == True
-- ---------------------------------------------------------------------

alternadaG :: (a -> Bool) -> [a] -> Bool
alternadaG p (x:y:xs)
  | p x       = not (p y) && alternadaG p (y:xs)
  | otherwise = p y && alternadaG p (y:xs)
alternadaG _ _ = True

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Redefinir la función alternada usando alternadaG y
-- comprobar con QuickCheck que ambas definiciones coinciden.
-- ---------------------------------------------------------------------

alternada3 :: [Int] -> Bool
alternada3 = alternadaG even

-- La propiedad es
propAlternada :: [Int] -> Bool
propAlternada xs = alternada1 xs == alternada3 xs

-- Su comprobación es
--    ghci> quickCheck propAlternada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4. Una lista de listas es engarzada si el último elemento
-- de cada lista coincide con el primero de la siguiente.
--
-- Definir la función
--    engarzada :: Eq a => [[a]] -> Bool
-- tal que (engarzada xss) se verifica si xss es una lista engarzada.
-- Por ejemplo,
--    engarzada [[1,2,3], [3,5], [5,9,0]] == True
--    engarzada [[1,4,5], [5,0], [1,0]]   == False
--    engarzada [[1,4,5], [], [1,0]]      == False
--    engarzada [[2]]                     == True
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución:
engarzada :: Eq a => [[a]] -> Bool
engarzada (xs:ys:xss) =
     not (null xs) && not (null ys) && last xs == head ys
  && engarzada (ys:xss)
engarzada _ = True

-- 2ª solución:
engarzada2 :: Eq a => [[a]] -> Bool
engarzada2 (xs:ys:xss) =
  and [ not (null xs)
      , not (null ys)
      , last xs == head ys
      , engarzada2 (ys:xss) ]
engarzada2 _ = True

-- 3ª solución:
engarzada3 :: Eq a => [[a]] -> Bool
engarzada3 xss =
  and [ not (null xs) && not (null ys) && last xs == head ys
      | (xs,ys) <- zip xss (tail xss)]

-- 4ª solución:
engarzada4 :: Eq a => [[a]] -> Bool
engarzada4 xss =
  all engarzados (zip xss (tail xss))
  where engarzados (xs,ys) =
          not (null xs) && not (null ys) && last xs == head ys

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. La sucesión de números de Pell se construye de la
-- forma siguiente:
--    P(0) = 0
--    P(1) = 1
--    P(n) = 2*P(n-1) + P(n-2)
-- Sus primeros términos son
--    0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, 33461,...
--
-- Definir la constante
--   sucPell :: [Integer]
```

```
-- tal que sucPell es la lista formada por los términos de la
-- sucesión. Por ejemplo,
--    ghci> take 15 sucPell
--    [0,1,2,5,12,29,70,169,408,985,2378,5741,13860,33461,80782]
-- ---------------------------------------------------------------------

sucPell :: [Integer]
sucPell = 0 : 1 : zipWith (+) sucPell (map (2*) (tail sucPell))

-- ---------------------------------------------------------------------
-- Ejercicio 5.2.  Definir la función
--    pell :: Int -> Integer
-- tal que (pell n) es el n-ésimo número de Pell. Por ejemplo,
--    pell 0   == 0
--    pell 10  == 2378
--    pell 100 == 6699209205055163766343890671318231377 2
-- ---------------------------------------------------------------------

pell :: Int -> Integer
pell n = sucPell !! n

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Los números de Pell verifican que los cocientes
--    ((P(m-1) + P(m))/P(m)
-- proporcionan una aproximación de la raiz cuadrada de 2.
--
-- Definir la función
--    aproxima :: Int -> [Double]
-- tal que (aproxima n) es la lista de las sucesivas aproximaciones
-- a la raíz de 2, desde m = 1 hasta n. Por ejemplo,
--    aproxima 3  == [1.0,1.5,1.4]
--    aproxima 4  == [1.0,1.5,1.4,1.4166666666666667]
--    aproxima 10 ==  [1.0,1.5,1.4,
--                     1.4166666666666667,1.4137931034482758,
--                     1.4142857142857144,1.4142011834319526,
--                     1.4142156862745099,1.4142131979695431,
--                     1.4142136248948696]
-- ---------------------------------------------------------------------

aproxima :: Int -> [Double]
```

```haskell
aproxima n = map aprox2 [1..n]
  where aprox2 n =
          fromIntegral (pell (n-1) + pell  n) / fromIntegral (pell n)
```

4.3. Examen 3 (31 de enero de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 3º examen de evaluación continua (31 de enero de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. [2.5 puntos] Definir la lista
--    ternasCoprimas :: [(Integer,Integer,Integer)]
-- cuyos elementos son ternas de primos relativos (a,b,c) tales que
-- a < b y a + b = c. Por ejemplo,
--    take 7 ternasCoprimas
--    [(1,2,3),(1,3,4),(2,3,5),(1,4,5),(3,4,7),(1,5,6),(2,5,7)]
--    ternasCoprimas !! 300000
--    (830,993,1823)
-- ---------------------------------------------------------------------

-- 1ª solución
ternasCoprimas :: [(Integer,Integer,Integer)]
ternasCoprimas = [(x,y,z) | y <- [1..]
                          , x <-[1..y-1]
                          , gcd x y == 1
                          , let z = x+y
                          , gcd x z == 1
                          , gcd y z == 1]

-- 2ª solución (Teniendo en cuenta que es suficiente que gcd x y == 1).
ternasCoprimas2 :: [(Integer,Integer,Integer)]
ternasCoprimas2 =
```

```
  [(x,y,x+y) | y <- [1..]
             , x <- [1..y-1]
             , gcd x y == 1]

-- ---------------------------------------------------------------------
-- Ejercicio 2. [2.5 puntos] Un entero positivo n es pandigital en base
-- b si su expresión en base b contiene todos los dígitos de 0 a b-1 al
-- menos una vez. Por ejemplo,
--    el 2 es pandigital en base 2 porque 2 en base 2 es 10,
--    el 11 es pandigital en base 3 porque 11 en base 3 es 102 y
--    el 75 es pandigital en base 4 porque 75 en base 4 es 1023.
--
-- Un número n es super pandigital de orden m si es pandigital en todas
-- las bases desde 2 hasta m. Por ejemplo, 978 es super pandigital de
-- orden 5 pues
--    en base 2 es: 1111010010
--    en base 3 es: 1100020
--    en base 4 es: 33102
--    en base 5 es: 12403
--
-- Definir la función
--    superPandigitales :: Integer -> [Integer]
-- tal que (superPandigitales m) es la lista de los números super
-- pandigitales de orden m. Por ejemplo,
--    take 3 (superPandigitales 3) == [11,19,21]
--    take 3 (superPandigitales 4) == [75,99,114]
--    take 3 (superPandigitales 5) == [978,1070,1138]
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

superPandigitales :: Integer -> [Integer]
superPandigitales m =
  [n | n <- [1..]
     , and [pandigitalBase b n | b <- [2..m]]]

-- (pandigitalBase b n) se verifica si n es pandigital en base la base
-- b. Por ejemplo,
--    pandigitalBase 4 75  ==  True
```

```haskell
--    pandigitalBase 4 76  ==  False
pandigitalBase :: Integer -> Integer -> Bool
pandigitalBase b n = [0..b-1] `esSubconjunto` enBase b n

-- (enBase b n) es la lista de los dígitos de n en base b. Por ejemplo,
--    enBase 4 75  ==  [3,2,0,1]
--    enBase 4 76  ==  [0,3,0,1]
enBase :: Integer -> Integer -> [Integer]
enBase b n | n < b     = [n]
           | otherwise = n `mod` b : enBase b (n `div` b)

-- (esSubconjunto xs ys) se verifica si xs es un subconjunto de ys. Por
-- ejemplo,
--    esSubconjunto [1,5] [5,2,1]  ==  True
--    esSubconjunto [1,5] [5,2,3]  ==  False
esSubconjunto :: Eq a => [a] -> [a] -> Bool
esSubconjunto xs ys = all (`elem` ys) xs

-- 2ª definición
-- =============

superPandigitales2 :: Integer -> [Integer]
superPandigitales2 a = foldl' f ys [a-1,a-2..2]
  where ys = filter (pandigitalBase a) [1..]
        f xs b = filter (pandigitalBase b) xs

pandigitalBase2 :: Integer -> Integer -> Bool
pandigitalBase2 b n = sort (nub xs) == [0..(b-1)]
  where xs = enBase b n

-- ---------------------------------------------------------------------
-- Ejercicio 3. [2.5 puntos] Definir la sucesión
--    sucFinalesFib :: [(Integer,Integer)]
-- cuyos elementos son los pares (n,x), donde x es el n-ésimo término de
-- la sucesión de Fibonacci, tales que la terminación de x es n. Por
-- ejemplo,
--    ghci> take 6 sucFinalesFib
--    [(0,0),(1,1),(5,5),(25,75025),(29,514229),(41,165580141)]
--    ghci> head [(n,x) | (n,x) <- sucFinalesFib, n > 200]
--    (245,71201125556981885592325792420049634380763282975024 5)
```

```
--    ghci> head [n | (n,_) <- sucFinalesFib, n > 10^4]
--    10945
-- ---------------------------------------------------------------------

sucFinalesFib :: [(Integer, Integer)]
sucFinalesFib =
  [(n, fib n) | n <- [0..]
              , show n `isSuffixOf` show (fib n)]

-- (fib n) es el n-ésimo término de la sucesión de Fibonacci.
fib :: Integer -> Integer
fib n = sucFib `genericIndex` n

-- sucFib es la sucesión de Fibonacci.
sucFib :: [Integer]
sucFib = 0 : 1 : zipWith (+) sucFib (tail sucFib)

-- 2ª definición de sucFib
sucFib2 :: [Integer]
sucFib2 = 0 : scanl (+) 1 sucFib2

-- ---------------------------------------------------------------------
-- Ejercicio 4. [2.5 puntos] 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
--     / \       / \           / \
--    8   3     5   3         5   3
--        |        /|\       /|\  |
--        4       4 7 6     4 7 6 7
-- se representan por
--    ej1, ej2, ej3 :: Arbol Int
--    ej1 = N 1 [N 8 [],N 3 [N 4 []]]
--    ej2 = N 1 [N 5 [], N 3 [N 4 [], N 7 [], N 6 []]]
--    ej3 = N 1 [N 5 [N 4 [], N 7 [], N 6 []], N 3 [N 7 []]]
--
-- Definir la función
--    minimaSuma :: Arbol Int -> Int
```

```haskell
-- tal que (minimaSuma a) es el mínimo de las sumas de las ramas del
-- árbol a. Por ejemplo,
--    minimaSuma ej1  ==  8
--    minimaSuma ej2  ==  6
--    minimaSuma ej3  ==  10
-- ---------------------------------------------------------------------

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 5 [], N 3 [N 4 [], N 7 [], N 6 []]]
ej3 = N 1 [N 5 [N 4 [], N 7 [], N 6 []], N 3 [N 7 []]]

-- 1ª definición
-- =============

minimaSuma :: Arbol Int -> Int
minimaSuma a = minimum [sum xs | xs <- ramas a]

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

-- 2ª definición (Como composición de funciones):
-- ==============================================

minimaSuma2 :: Arbol Int -> Int
minimaSuma2 = minimum . map sum . ramas

-- 3ª definición
-- =============

minimaSuma3 :: Arbol Int -> Int
minimaSuma3 (N x []) = x
minimaSuma3 (N x as) = x + minimum [minimaSuma3 a | a <- as]
```

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

minimaSuma4 :: Arbol Int -> Int
minimaSuma4 (N x []) = x
minimaSuma4 (N x as) = x + minimum (map minimaSuma4 as)
```

4.4. Examen 4 (14 de marzo de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (14 de marzo de 2017)
-- ---------------------------------------------------------------------

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

import Graphics.Gnuplot.Simple
import Data.Numbers.Primes
import Data.Matrix

-- ---------------------------------------------------------------------
-- Ejercicio 1. Decimos que una lista de números enteros es un camino
-- primo si todos sus elementos son primos y cada uno se diferencia del
-- anterior en un único dígito. Por ejemplo,
--    [1033, 1733, 3733, 3739, 3779, 8779, 8179] es un camino primo, y
--    [1033, 1733, 3733, 3739, 3793, 4793] no lo es.
--
-- Definir la función
--    caminoPrimo:: [Integer] -> Bool
-- tal que (caminoPrimo xs) se verifica si xs es un camino primo. Por
-- ejemplo,
--    caminoPrimo [1033, 1733, 3733, 3739, 3779, 8779, 8179] == True
--    caminoPrimo [1033, 1733, 3733, 3739, 3793, 4793]       == False
-- ---------------------------------------------------------------------

caminoPrimo :: [Integer] -> Bool
caminoPrimo ps = all pred (zip ps (tail ps))
```

```haskell
  where pred (x,y) = isPrime x && isPrime y && unDigitoDistinto x y

unDigitoDistinto :: Integer-> Integer -> Bool
unDigitoDistinto n m = aux (show n) (show m)
  where aux (x:xs) (y:ys) | x /= y    = xs == ys
                          | otherwise = aux xs ys
        aux _ _ = False

-- ---------------------------------------------------------------------
-- Ejercicio 2. Consideramos el siguiente tipo algebraico de los árboles
-- binarios:
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--       deriving (Show,Eq)
-- y el árbol
--    a1:: Arbol Int
--    a1 = N 9 (N 3 (H 2) (N 4 (H 1) (H 5))) (H 8)
--
--                     9
--                   /   \
--                  3     8
--                 / \
--                2   4
--                   / \
--                  1   5
--
-- En este árbol hay una rama en la que todos sus elementos son mayores
-- que 7, pero no hay ninguna rama en la que todos sus elementos sean
-- pares.
--
-- Definir la función
--    propExisteTodos :: (a -> Bool) -> Arbol a -> Bool
-- tal que (propExisteTodos p a) se verifica si hay una rama en la que
-- todos sus nodos (internos u hoja) cumple la propiedad p. Por ejemplo
--    propExisteTodos even a1  ==  False
--    propExisteTodos (>7) a1  ==  True
--    propExisteTodos (<=9) a1 ==  True
-- ---------------------------------------------------------------------

data Arbol a = H a
```

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

a1 :: Arbol Int
a1 = N 9 (N 3 (H 2) (N 4 (H 1) (H 5))) (H 8)

propiedadEA :: (a -> Bool) -> Arbol a -> Bool
propiedadEA p a = any (all p) (ramas a)

ramas :: Arbol a -> [[a]]
ramas (H x)     = [[x]]
ramas (N x i d) = [x:ys | ys <- ramas i ++ ramas d]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Representamos un tablero del juego de los barcos
-- mediante una matriz de 0 y 1, en la que cada barco está formado por
-- uno o varios 1 consecutivos, tanto en horizontal como en vertical.
-- Por ejemplo, las  siguientes matrices representan distintos tableros:
--    ej1, ej2, ej3, ej4 :: Matrix Int
--    ej1 = fromLists [[0,0,1,1],
--                     [1,0,1,0],
--                     [0,0,0,1],
--                     [1,1,0,1]]
--    ej2 = fromLists [[0,0,0,1],
--                     [1,0,0,0],
--                     [0,0,1,1],
--                     [1,0,0,1]]
--    ej3 = fromLists [[1,1,0,0],
--                     [0,1,0,1],
--                     [1,1,1,0],
--                     [0,0,1,0]]
--    ej4 = joinBlocks (ej1,ej3,ej2,ej3)
--
-- Definir la función
--    posicionesComunes :: Matrix Int -> Matrix Int -> [(Int,Int)]
-- tal que (posicionesComunes p q) es la lista con las posiciones
-- comunes ocupadas por algún barco en ambas matrices. Por ejemplo,
--    posicionesComunes ej1 ej2 == [(1,4),(2,1),(3,4),(4,1),(4,4)]
--    posicionesComunes ej1 ej3 == []
--    posicionesComunes ej2 ej3 == [(3,3)]
```

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

ej1, ej2, ej3, ej4 :: Matrix Int
ej1 = fromLists [[0,0,1,1],
                 [1,0,1,0],
                 [0,0,0,1],
                 [1,1,0,1]]
ej2 = fromLists [[0,0,0,1],
                 [1,0,0,0],
                 [0,0,1,1],
                 [1,0,0,1]]
ej3 = fromLists [[1,1,0,0],
                 [0,1,0,1],
                 [1,1,1,0],
                 [0,0,1,0]]
ej4 = joinBlocks (ej1,ej3,ej2,ej3)

-- 1ª definición:
posicionesComunes :: Matrix Int -> Matrix Int -> [(Int,Int)]
posicionesComunes p q =
  [(i,j) | i <-[1..m], j <-[1..n], a !(i,j) == 2]
  where m = nrows p
        n = ncols p
        a = p + q

-- 2ª definición:
posicionesComunes2 :: Matrix Int -> Matrix Int -> [(Int,Int)]
posicionesComunes2 p q =
  [(i,j) | i <-[1..m], j <-[1..n], p !(i,j) == 1 && q!(i,j) == 1]
  where m = nrows p
        n = ncols p

-- ---------------------------------------------------------------------
-- Ejercicio 3.2 Definir una función
--    juego :: Matrix Int -> IO ()
-- que pregunte al usuario por una posición y devuelva "Tocado" o
-- "Agua" según lo que haya en esa posición en el tablero ej4. Una
-- posible sesión puede ser la siguiente.
--    ghci> juego ej4
--    Elije una fila:
```

```
--    3
--    Elije una columna:
--    3
--    Agua
-- ---------------------------------------------------------------------

juego :: Matrix Int -> IO ()
juego p =
    do putStrLn "Elije una fila: "
       ci <- getLine
       let i = read ci
       putStrLn "Elije una columna: "
       cj <- getLine
       let j = read cj
       if p ! (i,j) == 1
          then (putStrLn "Tocado")
          else (putStrLn "Agua")

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. La fracción continua determinada por las sucesiones
-- as = [a1,a2,a3,...] y bs = [b1,b2,b3,...] es la siguiente
--                     b1
--     a1 + ---------------------------------
--                          b2
--           a2 + --------------------------------
--                               b3
--                a3 + ---------------------------
--                                    b4
--                     a4 + ----------------------
--                                         b5
--                          a5 + ---------------
--                                        ...
--
-- Definir la función
--    aproxFracionContinua:: [Int] -> [Int] -> Int -> Double
-- tal que (aproxFracionContinua xs ys n) es la aproximación con n
-- términos de la fracción contínua determinada por xs e ys. Por
-- ejemplo,
--    aproxFracionContinua [1,3..] [2,4..] 100       == 1.5414940825367982
--    aproxFracionContinua (repeat 1) (repeat 3) 100 == 2.302775637731995
```

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

aproxFracionContinua :: [Int] -> [Int] -> Int -> Double
aproxFracionContinua xs ys n = foldl f (a + b) (zip as bs)
  where (a:as) = map fromIntegral (reverse (take n xs))
        (b:bs) = map fromIntegral (reverse (take n ys))
        f z (x,y) = x + y/z


-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Una aproximación de pi mediante fracciones continuas
-- es la siguiente:
--    4                   1^2
--    --  = 1 + ------------------------------------
--    pi                        2^2
--              3 + ---------------------------------
--                                  3^2
--                    5 + ---------------------------
--                                       4^2
--                          7 + ---------------------
--                                            5^2
--                               9 + ---------------
--                                        ...
--
--
-- Definir la función
--    aproximacionPi :: Int -> Double
-- tal que (aproximacionPi n) es la n-ésima aproximación de pi, mediante
-- la expresión anterior. Por ejemplo,
--    aproximacionPi 100 == 3.141592653589793
-- ---------------------------------------------------------------------

aproximacionPi :: Int -> Double
aproximacionPi n = 4/(aproxFracionContinua as bs n)
  where as = [1,3..]
        bs = map (^2) [1..]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    grafica :: [Int] -> IO ()
```

```
-- tal que (grafica ns) dibuja la gráfica de las k-ésimas aproximaciones
-- de pi donde k toma los valores de la lista ns.
-- ---------------------------------------------------------------------

grafica :: [Int] -> IO ()
grafica ns = plotList [] $ map aproximacionPi ns
```

4.5. Examen 5 (21 de abril de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (21 de abril de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import I1M.PolOperaciones
import I1M.Pila
import Graphics.Gnuplot.Simple
import qualified Data.Matrix as M
import qualified Data.Set    as S
import qualified Data.Vector as V

-- ---------------------------------------------------------------------
-- Ejercicio 1. Los números 181 y 1690961 son ambos palíndromos y suma
-- de cuadrados consecutivos, pues 181 = 9^2 + 10^2 y
-- 1690961 = 919^2 + 920^2.
--
-- Definir la constante
--    sucesion :: [Integer]
-- tal que sucesion es la lista de los números que son suma de cuadrados
-- consecutivos y palíndromos. Por ejemplo,
--    take 10 sucesion
--    [1,5,181,313,545,1690961,3162613,3187813,5258525,5824285]
-- ---------------------------------------------------------------------

sucesion :: [Integer]
sucesion = filter palindromo sucSumaCuadradosConsecutivos
```

```haskell
palindromo :: Integer -> Bool
palindromo n = xs == reverse xs
  where xs = show n

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

-- 2ª definición
sucSumaCuadradosConsecutivos2 :: [Integer]
sucSumaCuadradosConsecutivos2 =
  [x^2 + (x+1)^2 | x <- [0..]]

-- 3ª definición
sucSumaCuadradosConsecutivos3 :: [Integer]
sucSumaCuadradosConsecutivos3 =
  zipWith (+) cuadrados (tail cuadrados)

cuadrados :: [Integer]
cuadrados = map (^2) [0..]

-- Comparación de eficiencia
--    ghci> maximum (take (10^6) sucSumaCuadradosConsecutivos)
--    1999998000001
--    (1.47 secs, 912,694,568 bytes)
--    ghci> maximum (take (10^6) sucSumaCuadradosConsecutivos2)
--    1999998000001
--    (1.79 secs, 1,507,639,560 bytes)
--    ghci> maximum (take (10^6) sucSumaCuadradosConsecutivos3)
--    1999998000001
--    (1.29 secs, 840,488,376 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se define la relación de orden entre las pilas como el
-- orden lexicográfico. Es decir, la pila p1 es "menor" que p2 si la
-- cima de p1 es menor que la cima de p2, o si son iguales, la pila que
```

```
-- resulta de desapilar p1 es "menor" que la pila que resulta de
-- desapilar p2.
--
-- Definir la función
--    menorPila :: Ord a => Pila a -> Pila a -> Bool
-- tal que (menorPila 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]
--    p3 = foldr apila vacia [3..10]
--    p4 = foldr apila vacia [4,-1,7,3,8,10,0,3,3,4]
-- se verifica que
--    menorPila p1 p2    == True
--    menorPila p2 p1    == False
--    menorPila p3 p4    == True
--    menorPila vacia p1 == True
--    menorPila p1 vacia == False
-- ---------------------------------------------------------------------

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

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]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    polM:: M.Matrix Int -> Polinomio Int
-- tal que (polM m) es el polinomio cuyas raices son los elementos de la
```

```haskell
-- diagonal de la matriz m. Por ejemplo, para la matriz definida por
--    m1 :: M.Matrix Int
--    m1 = M.fromLists [[1, 2, 3,4],
--                      [0,-1, 0,5],
--                      [0, 0,-2,9],
--                      [0, 0, 0,3]]
-- se tiene
--    ghci> polM m1
--    x^4 + -1*x^3 + -7*x^2 + 1*x + 6
-- ---------------------------------------------------------------------

m1 :: M.Matrix Int
m1 = M.fromLists [[1, 2, 3,4],
                  [0,-1, 0,5],
                  [0, 0,-2,9],
                  [0, 0, 0,3]]

polM :: M.Matrix Int -> Polinomio Int
polM m = foldr multPol polUnidad ps
  where ds = V.toList (M.getDiag m)
        ps = [consPol 1 1 (consPol 0 (-d) polCero) | d <- ds]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    dibPol :: M.Matrix Int -> [Int] -> IO ()
-- tal que (dibPol m xs) dibuje la gráfica del polinomio (polM m)
-- tomando los valores en la lista xs. Evaluar (dibPol m1 [-10..10]).
-- ---------------------------------------------------------------------

dibPol :: M.Matrix Int -> [Int] -> IO ()
dibPol m xs =
  plotList [Key Nothing]
           (zip xs (map (valor (polM m)) xs))

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. La sucesión de Recamán está definida como sigue:
--    a(0) = 0
--    a(n) = a(n-1) - n, si a(n-1) > n y no figura ya en la sucesión
--    a(n) = a(n-1) + n, en caso contrario.
--
```

```haskell
-- Definir la constante
--    sucRecaman :: [Int]
-- tal que sucRecaman es la sucesión anterior. Por ejemplo,
--    ghci> take 21 sucRecaman
--    [0,1,3,6,2,7,13,20,12,21,11,22,10,23,9,24,8,25,43,62,42]
--    ghci> sucRecaman !! (10^3)
--    3686
--    ghci> sucRecaman !! (10^4)
--    18658
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

sucRecaman1:: [Int]
sucRecaman1 = map suc1 [0..]

suc1 :: Int -> Int
suc1 0 = 0
suc1 n | y > n && y - n `notElem` ys = y - n
       | otherwise                   = y + n
  where y  = suc1 (n - 1)
        ys = [suc1 k | k <- [0..n - 1]]

-- 2ª solución (ev. perezosa)
-- ==========================

sucRecaman2 :: [Int]
sucRecaman2 = 0:zipWith3 f sucRecaman [1..] (repeat sucRecaman)
  where f y n ys | y > n && y - n `notElem` take n ys = y - n
                 | otherwise                         = y + n

-- 3ª solución (ev. perezosa y conjuntos)
-- ======================================

sucRecaman3 :: [Int]
sucRecaman3 = 0 : recaman (S.singleton 0) 1 0

recaman :: S.Set Int -> Int -> Int -> [Int]
recaman s n x
```

```
  | x > n && (x-n) `S.notMember` s =
    (x-n) : recaman (S.insert (x-n) s) (n+1) (x-n)
  | otherwise =
    (x+n):recaman (S.insert (x+n) s) (n+1) (x+n)

-- Comparación de eficiencia:
--    ghci> sucRecaman1 !! 25
--    17
--    (3.76 secs, 2,394,593,952 bytes)
--    ghci> sucRecaman2 !! 25
--    17
--    (0.00 secs, 0 bytes)
--    ghci> sucRecaman3 !! 25
--    17
--    (0.00 secs, 0 bytes)
--
--    ghci> sucRecaman2 !! (2*10^4)
--    14358
--    (2.69 secs, 6,927,559,784 bytes)
--    ghci> sucRecaman3 !! (2*10^4)
--    14358
--    (0.04 secs, 0 bytes)

-- En lo que sigue se usará la 3ª definición.
sucRecaman :: [Int]
sucRecaman = sucRecaman3

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Se ha conjeturado que cualquier entero positivo
-- aparece en la sucesión de Recaman.
--
-- Definir la función
--    conjetura_Recaman :: Int -> Bool
-- tal que (conjetura_Recaman n) comprueba la conjetura para x <= n. Por
-- ejemplo,
--    conjeturaRecaman 30  == True
--    conjeturaRecaman 100 == True
-- ---------------------------------------------------------------------

-- 1ª definición
```

```haskell
conjeturaRecaman :: Int -> Bool
conjeturaRecaman n =
  and [not $ null $ filter (==x) sucRecaman | x <- [1..n]]

-- 3ª definición
conjeturaRecaman2 :: Int -> Bool
conjeturaRecaman2 n =
  all (`elem` sucRecaman3) [1..n]

-- Comparación de eficiencia
--    ghci> conjeturaRecaman 100 == True
--    True
--    (0.44 secs, 249,218,152 bytes)
--    ghci> conjeturaRecaman2 100 == True
--    True
--    (0.02 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función
--    invRecaman :: Int -> Int
-- tal que (invRecaman n) es la posición de n en la sucesión de
-- Recaman. Por ejemplo,
--    invRecaman 10  == 12
--    invRecaman 100 == 387
-- ---------------------------------------------------------------------

-- 1ª definición
invRecaman :: Int -> Int
invRecaman n = head [m | m <- [0..], sucRecaman !! m == n]

-- 2ª definición
invRecaman2 :: Int -> Int
invRecaman2 n =
  length (takeWhile (/=n) sucRecaman)

-- Comparación de eficiencia
--    ghci> invRecaman 10400
--    50719
--    (3.13 secs, 42,679,336 bytes)
--    ghci> invRecaman2 10400
```

```
--    50719
--    (0.00 secs, 0 bytes)
```

4.6. Examen 6 (12 de junio de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- 6º examen de evaluación continua (12 de junio de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Array
import Data.Numbers.Primes
import Test.QuickCheck
import Graphics.Gnuplot.Simple

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La función de Möebius,  está definida para todos
-- los enteros positivos ncomo sigue:
--    mu(n) =  1 si n es libre de cuadrados y tiene un número par de
--               factores primos distintos.
--    mu(n) = -1 si n es libre de cuadrados y tiene un número impar de
--               factores primos distintos.
--    mu(n) =  0 si n es divisible por algún cuadrado.
--
-- Definir la función
--    mu :: Int -> Int
-- tal que (mu n) es el valor mu(n). Por ejemplo:
--    mu 1    == 1
--    mu 1000 == 0
--    mu 3426 == -1
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

mu1 :: Int -> Int
```

```haskell
mu1 n | divisibleAlgunCuadrado n = 0
      | even k                   = 1
      | otherwise                = -1
  where k = length (map fst (factorizacion n))

factorizacion :: (Integral t) => t -> [(t,t)]
factorizacion n = [(head ps, genericLength ps) | ps <- pss]
  where pss = group (primeFactors n)

divisibleAlgunCuadrado :: Int -> Bool
divisibleAlgunCuadrado = any (>1) . map snd . factorizacion

libreDeCuadrados :: Int -> Bool
libreDeCuadrados = not . divisibleAlgunCuadrado

-- 2ª definición
-- =============

mu2 :: Int -> Int
mu2 n | any (>1) es = 0
      | even k      = 1
      | otherwise   = -1
  where ps = factorizacion n
        k = length (map fst ps)
        es = map snd ps

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que se verifica la siguiente
-- propiedad: la suma de mu(d), siendo d los divisores positivos de n, es
-- cero excepto cuando n = 1.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mu :: Int -> Property
prop_mu n = abs n /= 1 ==> sum (map mu2 (divisores n)) == 0
  where divisores x = [y | y <- [1..abs x], x `mod` y == 0]

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

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    graficaMu :: [Int] -> IO ()
-- tal que (graficaMu ns) dibuje la gráfica de la función tomando los
-- valores en la lista ns.
-- ---------------------------------------------------------------------

graficaMu :: [Int] -> IO ()
graficaMu ns = plotList [] $ map mu2 ns

-- ---------------------------------------------------------------------
-- Ejercicio 2. Se dice que un número es generable si se puede escribir
-- como una sucesión (quizá vacía) de multiplicaciones por 3 y sumas de
-- 5 al número 1.
--
-- Definir las siguientes funciones
--    generables     :: [Integer]
--    generable      :: Integer -> Bool
-- tales que
-- + generables es la sucesión de los números generables. Por ejemplo,
--      ghci> take 20 generables
--      [1,3,6,8,9,11,13,14,16,18,19,21,23,24,26,27,28,29,31,32]
--      ghci> generables !! (10^6)
--      1250008
-- + (generable x) se verifica si x es generable. Por ejemplo,
--      generable 23       ==  True
--      generable 77       ==  True
--      generable 15       ==  False
--      generable 1250008  ==  True
--      generable 1250010  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

generables :: [Integer]
generables = 1 : mezcla [3 * x | x <- generables]
                        [5 + x | x <- generables]
```

```haskell
-- (mezcla xs ys) es la lista ordenada obtenida mezclando las dos listas
-- ordenadas xs e ys, suponiendo que ambas son infinitas. Por ejemplo,
--    take 10 (mezcla [2,12..] [5,15..])  ==  [2,5,12,15,22,25,32,35,42,45]
--    take 10 (mezcla [2,22..] [5,15..])  ==  [2,5,15,22,25,35,42,45,55,62]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla us@(x:xs) vs@(y:ys)
  | x < y     = x : mezcla xs vs
  | x == y    = x : mezcla xs ys
  | otherwise = y : mezcla us ys

generable :: Integer -> Bool
generable x =
  x == head (dropWhile (<x) generables)

-- 2ª definición
-- =============

generable2 :: Integer -> Bool
generable2 1 = True
generable2 x =    (x `mod` 3 == 0 && generable2 (x `div` 3))
               || (x > 5 && generable2 (x - 5))

generables2 :: [Integer]
generables2 = filter generable2 [1..]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideramos el siguiente tipo algebraico de los árboles
--    data Arbol a = N a [Arbol a]
--                   deriving Show
-- Por ejemplo, el árbol
--                      1
--                    /  |  \
--                   4   5   2
--                  / \      |
--                 3   7     6
-- se define por
--    ej1:: Arbol Int
--    ej1 = N 1 [N 4 [N 3 [], N 7 []],
--               N 5 [],
--               N 2 [N 6 []]]
```

```haskell
--
-- A partir de él, podemos construir otro árbol en el que, para cada
-- nodo de ej1, calculamos el número de nodos del subárbol que lo tiene
-- como raíz. Obtenemos el árbol siguiente:
--
--                     7
--                   /  |  \
--                  3   1   2
--                 / \      |
--                1   1     1
--
-- Definir la función
--    arbolNN :: Arbol a -> Arbol Int
-- tal que (arbolNN x) es un árbol con la misma estructura que x, de
-- forma que en cada nodo de (arbolNN x) aparece el número de nodos del
-- subárbol correspondiente en x. Por ejemplo,
--    arbolNN ej1 == N 7 [N 3 [N 1 [],N 1 []],
--                        N 1 [],
--                        N 2 [N 1 []]]
-- ---------------------------------------------------------------------

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

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

raiz :: Arbol a -> a
raiz (N r _) = r

arbolNN :: Arbol a -> Arbol Int
arbolNN (N r []) = N 1 []
arbolNN (N r as) = N (s+1) bs
  where bs = map arbolNN as
        rs = map raiz bs
        s = sum rs

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

```
-- Ejercicio 4. El procedimiento de codificación matricial se puede
-- entender siguiendo la codificación del mensaje "todoparanada" como se
-- muestra a continuación:
--     Se calcula la longitud L del mensaje. En el ejemplo es L es 12.
--     Se calcula el menor entero positivo N cuyo cuadrado es mayor o
--     igual que L. En el ejemplo N es 4.  Se extiende el mensaje con
--     N²-L asteriscos. En el ejemplo, el mensaje extendido es
--     "todoparanada****" Con el mensaje extendido se forma una matriz
--     cuadrada NxN. En el ejemplo la matriz es
--      | t o d o |
--      | p a r a |
--      | n a d a |
--      | * * * * |
--
--     Se rota 90º la matriz del mensaje extendido. En el ejemplo, la
--     matriz rotada es
--
--      | * n p t |
--      | * a a o |
--      | * d r d |
--      | * a a o |
--
--     Se calculan los elementos de la matriz rotada. En el ejemplo, los
--     elementos son "*npt*aap*drd*aao" El mensaje codificado se obtiene
--     eliminando los asteriscos de los elementos de la matriz
--     rotada. En el ejemplo, "nptaapdrdaao".
--
-- Definir la función
--    codificado :: String -> String
-- tal que (codificado cs) es el mensaje obtenido aplicando la
-- codificación matricial al mensaje cs. Por ejemplo,
--    codificado "todoparanada"    ==  "nptaaodrdaao"
--    codificado "nptaaodrdaao"    ==  "danaopadtora"
--    codificado "danaopadtora"    ==  "todoparanada"
--    codificado "entodolamedida"  ==  "dmdeaeondltiao"
-- ----------------------------------------------------------------------

codificado :: String -> String
codificado cs =
  filter (/='*') (elems (rota p))
```

```
  where n = ceiling (sqrt (genericLength cs))
        p = listArray ((1,1),(n,n)) (cs ++ repeat '*')

rota :: Array (Int,Int) Char -> Array (Int,Int) Char
rota p = array d [((i,j),p!(n+1-j,i)) | (i,j) <- indices p]
  where d = bounds p
        n = fst (snd d)
```

4.7. Examen 7 (29 de junio de 2017)

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

4.8. Examen 8 (8 de septiembre de 2017)

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

4.9. Examen 9 (21 de noviembre de 2017)

El examen es común con el del grupo 1 (ver página 65).

5

Exámenes del grupo 5

Andrés Cordón y Antonia M. Chávez

5.1. Examen 1 (26 de octubre de 2016)

```
-- Informática (1º del Grado en Matemáticas, Grupo 5)
-- 1º examen de evaluación continua (26 de octubre de 2016)
-- ---------------------------------------------------------------------

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

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. La sombra de un número x es el que se obtiene borrando
-- las cifras de x que ocupan lugares impares. 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 4736  ==  43
--    sombra 473   ==  43
--    sombra 47    ==  4
--    sombra 4     ==  4
```

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

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

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

-- 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

-- 3ª definición (por recursión y composición)
-- ===========================================

sombra3 :: Int -> Int
sombra3 = read . elementosEnPares . show

prop_sombra :: Int -> Property
prop_sombra x =
  x >= 0 ==>
  nDigitos (sombra x) == ceiling (fromIntegral (nDigitos x) / 2)

nDigitos n = length (show n)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
```

```
--    esSombra :: Int -> Int -> Bool
-- tal que (esSombra x y) se verifica si y es sombra de x. Por ejemplo,
--    esSombra 72941 791  ==  True
--    esSombra 72941 741  ==  False
-- ---------------------------------------------------------------------

esSombra :: Int -> Int -> Bool
esSombra x y = sombra x == y

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    conSombra :: Int -> [Int]
-- tal que (conSombra x n) es la lista de números con el menor número
-- posible de cifras cuya sombra es x. Por ejemplo,
--    ghci> conSombra 2
--    [2]
--    ghci> conSombra 23
--    [203,213,223,233,243,253,263,273,283,293]
--    ghci> conSombra 234
--    [20304,20314,20324,20334,20344,20354,20364,20374,20384,20394,
--     21304,21314,21324,21334,21344,21354,21364,21374,21384,21394,
--     22304,22314,22324,22334,22344,22354,22364,22374,22384,22394,
--     23304,23314,23324,23334,23344,23354,23364,23374,23384,23394,
--     24304,24314,24324,24334,24344,24354,24364,24374,24384,24394,
--     25304,25314,25324,25334,25344,25354,25364,25374,25384,25394,
--     26304,26314,26324,26334,26344,26354,26364,26374,26384,26394,
--     27304,27314,27324,27334,27344,27354,27364,27374,27384,27394,
--     28304,28314,28324,28334,28344,28354,28364,28374,28384,28394,
--     29304,29314,29324,29334,29344,29354,29364,29374,29384,29394]
-- ---------------------------------------------------------------------

conSombra :: Int -> [Int]
conSombra x =
  [read ys | ys <- intercala (show x) ['0'..'9']]

-- (intercala xs ys) es la lista obtenida intercalando los elementos de
-- xs entre los de ys. Por ejemplo,
--    ghci> intercala "79" "15"
--    ["719","759"]
--    ghci> intercala "79" "154"
```

```
--    ["719","759","749"]
--    ghci> intercala "796" "15"
--    ["71916","71956","75916","75956"]
--    ghci> intercala "796" "154"
--    ["71916","71956","71946",
--     "75916","75956","75946",
--     "74916","74956","74946"]
intercala :: [a] -> [a] -> [[a]]
intercala []     _  = []
intercala [x]    _  = [[x]]
intercala (x:xs) ys = [x:y:zs | y <- ys
                              , zs <- intercala xs ys]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    mezcla :: [a] -> [a] -> [a]
-- tal que (mezcla xs ys) es la lista obtenida alternando cada uno de
-- los elementos de xs con los de ys. Por ejemplo,
--    mezcla [1,2,3] [4,7,3,2,4]        == [1,4,2,7,3,3,2,4]
--    mezcla [4,7,3,2,4] [1,2,3]        == [4,1,7,2,3,3,2,4]
--    mezcla [1,2,3] [4,7]              == [1,4,2,7,3]
--    mezcla "E er eSnRqe" "lprod a ou" == "El perro de San Roque"
--    mezcla "et" "so sobra"            == "esto sobra"
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
mezcla :: [a] -> [a] -> [a]
mezcla xs ys =
  concat [[x,y] | (x,y) <- zip xs ys]
  ++ drop (length xs) ys
  ++ drop (length ys) xs

-- 2ª definición (por recursión)
mezcla2 :: [a] -> [a] -> [a]
mezcla2 [] ys = ys
mezcla2 xs [] = xs
mezcla2 (x:xs) (y:ys) = x : y : mezcla2 xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 3. Un elemento de una lista es una cresta si es mayor que
```

```haskell
-- todos los anteriores a ese elemento en la lista.
--
-- Definir la función
--    crestas :: Ord a => [a] -> [a]
-- tal que (crestas xs) es la lista de las crestas de xs. Por
-- ejemplo,
--    crestas [80,1,7,8,4]  ==  [80]
--    crestas [1,7,8,4]     ==  [1,7,8]
--    crestas [3,2,6,1,5,9] ==  [3,6,9]
-- ---------------------------------------------------------------------

-- 1ª definición
-- =============

crestas :: Ord a => [a] -> [a]
crestas xs = [x | (x,n) <- zip xs [1 ..]
                , esMayor x (take (n-1) xs)]

esMayor :: Ord a => a -> [a] -> Bool
esMayor x xs = and [x > y | y <- xs]

-- 2ª definición
-- =============

crestas2 :: Ord a => [a] -> [a]
crestas2 xs = [x | (x,ys) <- zip xs (inits xs)
                 , esMayor x ys]

-- 3ª definición
-- =============

crestas3 :: Ord a => [a] -> [a]
crestas3 xs = aux xs []
  where aux [] _ = []
        aux (x:xs) ys | esMayor x ys = x : aux xs (x:ys)
                      | otherwise    = aux xs (x:ys)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    posicion :: (Float,Float) -> Float -> String
```

```haskell
-- tal que (posicion p z) es la posición del punto p respecto del
-- cuadrado de lado l y centro (0,0). Por ejemplo,
--    posicion (-1, 1) 6 == "Interior"
--    posicion (-1, 2) 6 == "Interior"
--    posicion ( 0,-3) 6 == "Borde"
--    posicion ( 3,-3) 6 == "Borde"
--    posicion ( 3, 1) 6 == "Borde"
--    posicion (-1, 7) 6 == "Exterior"
--    posicion (-1, 4) 6 == "Exterior"
-- ---------------------------------------------------------------------

posicion :: (Float,Float) -> Float -> String
posicion (x,y) z
    | abs x < z/2 && abs y < z/2 = "Interior"
    | abs x > z/2 || abs y > z/2 = "Exterior"
    | otherwise                  = "Borde"
```

5.2. Examen 2 (30 de noviembre de 2016)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 2º examen de evaluación continua (30 de noviembre de 2016)
-- ---------------------------------------------------------------------

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

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    cuadrado :: [Integer] -> Integer
-- tal que (cuadrado xs) es el número obtenido uniendo los cuadrados de
-- los números de la lista xs. Por ejemplo,
--    cuadrado [2,6,9] == 43681
--    cuadrado [10]    == 100
-- ---------------------------------------------------------------------

-- 1ª definición (usando recursión)
cuadrado1 :: [Integer] -> Integer
```

```haskell
cuadrado1 xs = read (aux xs)
  where aux [] = ""
        aux (y:ys) = show (y^2) ++ aux ys

-- 2ª definición (plegado a la derecha)
cuadrado2 :: [Integer] -> Integer
cuadrado2 = read . aux
  where aux = foldr f ""
        f x y = show (x^2) ++ y

-- 3ª definición (por comprensión)
cuadrado3 :: [Integer] -> Integer
cuadrado3 ys = read (concat [show (y^2) | y <- ys])

-- 4ª definición (con orden superior)
cuadrado4 :: [Integer] -> Integer
cuadrado4 = read . concatMap (show . (^2))

-- 5ª definición (con acumulador)
cuadrado5 :: [Integer] -> Integer
cuadrado5 xs = read (aux xs "")
  where aux [] ys     = ys
        aux (x:xs) ys = aux xs (ys ++ show (x^2))

-- 6ª definición (con plegado a la izquierda)
cuadrado6 :: [Integer] -> Integer
cuadrado6 xs = read (foldl g "" xs)
  where g ys x = ys ++ show (x^2)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Un elemento de una lista es una cima si es mayor que los
-- que tiene más cerca, su antecesor y su sucesor.
--
-- Definir la función
--    cimas:: [Int] -> [Int]
-- tal que (cimas xs) es la lista de las cimas de xs. Por ejemplo,
--    cimas [80,1,7,5,9,1]  ==  [7,9]
--    cimas [1,7,8,4]       ==  [8]
--    cimas [3,2,6,1,5,4]   ==  [6,5]
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª definición (por comprensión)
cimasC ::  [Int] -> [Int]
cimasC xs = [x | (y,x,z) <- trozos xs, y < x, x > z]

trozos :: [a] -> [(a,a,a)]
trozos (x:y:z:zs) = (x,y,z) : trozos (y:z:zs)
trozos _          = []

-- 2ª definición (orden superior)
cimasS ::  [Int] -> [Int]
cimasS xs = concatMap f (trozos xs)
  where f (a,b,c) | a < b && b > c = [b]
                  | otherwise      = []

-- 3ª definición (por recursión)
cimasR ::  [Int] -> [Int]
cimasR (x:y:z:xs) | x < y && y > z = y : cimasR (y:z:xs)
                  | otherwise      = cimasR (y:z:xs)
cimasR _ = []

-- 4ª definición (plegado a la derecha)
cimasPR ::  [Int] -> [Int]
cimasPR xs = concat (foldr f [] (trozos xs))
    where f (a,b,c) ys | a < b && b > c = [b] : ys
                       | otherwise      = []  : ys

-- 5ª definición (con acumuladores)
cimasA ::  [Int] -> [Int]
cimasA xs = concat (aux (trozos xs) [[]])
  where aux [] ac = ac
        aux ((a,b,c):ts) ac
          | a < b && b > c = aux ts (ac ++ [[b]])
          | otherwise      = aux ts (ac ++ [[]])

-- 6ª definición (por plegado a la izquierda)
cimasPL ::  [Int] -> [Int]
cimasPL xs = concat (foldl g [] (trozos xs))
    where g acum (a,b,c)
            | a < b && b > c = acum ++ [[b]]
```

```
            | otherwise      = acum ++ [[]]
```

5.3. Examen 3 (31 de enero de 2017)

El examen es común con el del grupo 4 (ver página 117).

5.4. Examen 4 (15 de marzo de 2017)

```
-- Informática (1º del Grado en Matemáticas)
-- 4º examen de evaluación continua (15 de marzo de 2017)
-- ---------------------------------------------------------------------

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

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

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir el predicado
--    relacionados :: Int -> Int -> Bool
-- tal que (relacionados x y) se verifica si los enteros positivos x e y
-- contienen los mismos dígitos (sin importar el orden o la repetición
-- de los mismos). Por ejemplo:
--    relacionados 12 1121   == True
--    relacionados 12 123    == False
-- ---------------------------------------------------------------------

relacionados :: Int -> Int -> Bool
relacionados x y =
  sort (nub (show x)) == sort (nub (show y))

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la lista infinita
--    paresRel :: [(Int,Int)]
-- cuyo elementos son los pares de enteros positivos (a,b), con
-- 1 <= a < b, tales que a y b están relacionados. Por ejemplo,
```

```haskell
--    ghci> take 8 paresRel
--    [(1,11),(12,21),(2,22),(13,31),(23,32),(3,33),(14,41),(24,42)]
-- ---------------------------------------------------------------------

paresRel :: [(Int,Int)]
paresRel = [(a,b) | b <- [2..]
                  , a <- [1..b-1]
                  , relacionados a b]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    lugar :: Int -> Int -> Maybe Int
-- tal que (lugar x y) es el lugar que ocupa el par (x,y) en la lista
-- infinita paresRel o bien Nothing si dicho par no está en la lista.
-- Por ejemplo,
--    lugar 4 44  == Just 10
--    lugar 5 115 == Nothing
-- ---------------------------------------------------------------------

lugar :: Int -> Int -> Maybe Int
lugar x y | relacionados x y = Just z
          | otherwise        = Nothing
   where z = 1 + length (takeWhile (/=(x,y)) paresRel)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los árboles binarios con elementos en las
-- hojas y en los nodos mediante el tipo de dato
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving Show
-- Por ejemplo,
--    ej1 :: Arbol Int
--    ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))
--
-- Definir la función
--    ramasCon :: Eq a => Arbol a -> a -> [[a]]
-- tal que (ramasCon a x) es la lista de las ramas del árbol a en las
-- que aparece el elemento x. Por ejemplo,
--   ramasCon ej1 2 ==  [[5,2,1],[5,2,2],[5,3,2]]
-- ---------------------------------------------------------------------
```

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

ej1 :: Arbol Int
ej1 = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))

ramasCon :: Eq a => Arbol a -> a -> [[a]]
ramasCon a x = filter (x `elem`) (ramas a)

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Representamos las matrices mediante el tipo de dato
--    type Matriz a = Array (Int,Int) a
-- Por ejemplo,
--    ejm :: matriz int
--    ejm = listarray ((1,1),(3,4)) [1,2,3,0,4,5,6,7,7,5,1,11]
-- representa la matriz
--    |1 2 3 0 |
--    |4 5 6 7 |
--    |7 5 1 11|
--
-- definir la función
--   cruz :: (eq a,num a) => matriz a -> int -> int -> matriz a
-- tal que (cruz p i j) es la matriz obtenida anulando todas las
-- posiciones de p excepto las de la fila i y la columna j. por ejmplo,
--    ghci> cruz ejM 2 3
--    array ((1,1),(3,4)) [((1,1),0),((1,2),0),((1,3),3),((1,4),0),
--                         ((2,1),4),((2,2),5),((2,3),6),((2,4),7),
--                         ((3,1),0),((3,2),0),((3,3),1),((3,4),0)]
--    ghci> elems (cruz ejM 2 3)
--    [0,0,3,0,
--     4,5,6,7,
--     0,0,1,0]
-- ---------------------------------------------------------------------
```

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

ejM :: Matriz Int
ejM = listArray ((1,1),(3,4)) [1,2,3,0,4,5,6,7,7,5,1,11]

cruz :: Num a => Matriz a -> Int -> Int -> Matriz a
cruz p fil col =
  array (bounds p) [((i,j), f i j) | (i,j) <- indices p]
  where f i j | i == fil || j == col = p!(i,j)
              | otherwise = 0

-- 2ª definición
cruz2 :: Num a => Matriz a -> Int -> Int -> Matriz a
cruz2 p fil col =
  p // [((i,j),0) | (i,j) <- indices p, i /= fil, j /= col]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Una matriz está ordenada por columnas si cada una de
-- sus columnas está ordenada en orden creciente.
--
-- Definir la función
--    ordenadaCol :: Ord a => Matriz a -> Bool
-- tal que (ordenadaCol p) se verifica si la matriz p está ordenada por
-- columnas. Por ejemplo,
--    ghci> ordenadaCol (listArray ((1,1),(3,4)) [1..12])
--    True
--    ghci> ordenadaCol (listArray ((1,1),(3,4)) [1,2,3,0,4,5,6,7,7,5,1,11])
--    False
-- ---------------------------------------------------------------------

ordenadaCol :: Ord a => Matriz a -> Bool
ordenadaCol p = and [xs == sort xs | xs <- columnas p]

columnas :: Matriz a -> [[a]]
columnas p = [[p!(i,j) | i <- [1..n]] | j <- [1..m]]
   where (n,m) = snd (bounds p)

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Representamos los pesos de un conjunto de objetos
-- mediante una lista de asociación (objeto,peso). Por ejemplo,
```

```
--    ejLista :: [(Char,Int)]
--    ejLista = [('a',10),('e',5),('i',7),('l',2),('s',1),('v',4)]
--
-- Definir la función
--    peso :: Eq a => [(a,Int)] -> [a] -> Int
-- tal que (peso xs as) es el peso del conjunto xs de acuerdo con la
-- lista de asociación as. Por ejemplo:
--    peso ejLista "sevilla" == 31
-- ---------------------------------------------------------------------

ejLista :: [(Char,Int)]
ejLista = [('a',10),('e',5),('i',7),('l',2),('s',1),('v',4)]

peso :: Eq a => [(a,Int)] -> [a] -> Int
peso as xs = sum [busca x as | x <- xs]
  where busca x as = head [z | (y,z) <- as, y == x]

-- 2ª definición
peso2 :: Eq a => [(a,Int)] -> [a] -> Int
peso2 as xs = sum [fromJust (lookup x as) | x <- xs]

-- 3ª definición
peso3 :: Eq a => [(a,Int)] -> [a] -> Int
peso3 as = sum . map (fromJust . (`lookup` as))

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    conPeso :: Eq a => Int -> [(a,Int)] -> [a] -> [[a]]
-- tal que (conPeso x as xs) es la lista de los subconjuntos de xs con
-- peso x de acuerdo con la asignación as. Por ejemplo:
--    conPeso 10 ejLista "sevilla" ==  ["sev","sell","sil","sil","a"]
-- ---------------------------------------------------------------------

conPeso :: Eq a => Int -> [(a,Int)] -> [a] -> [[a]]
conPeso x as xs =
  [ys | ys <- subsequences xs, peso as ys == x]
```

5.5. Examen 5 (26 de abril de 2017)

```haskell
-- Informática (1º del Grado en Matemáticas)
-- 5º examen de evaluación continua (26 de abril de 2017)
-- ---------------------------------------------------------------------

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

import Data.List
import Data.Matrix
import qualified Data.Vector as V
import I1M.Pila
import I1M.PolOperaciones

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Un número natural x es un cuadrado perfecto si x = b^2
-- para algún número natural b <= x. Por ejemplo, 25 es un cuadrado
-- perfecto y 24 no lo es.
--
-- Un número natural se dirá equilibrado si contiene una cantidad par de
-- cifras pares y una cantidad impar de cifras impares. Por ejemplo,
-- 124, 333 y 20179 son equilibrados, mientras que 1246, 2333 y 2017 no
-- lo son.
--
-- Definir la lista infinita
--    equilibradosPerf :: [Integer]
-- de todos los cuadrados perfectos que son equilibrados. Por ejemplo
--    ghci> take 15 equilibradosPerf
--    [1,9,100,144,225,256,289,324,441,625,676,784,841,900,10000]
-- ---------------------------------------------------------------------

equilibradosPerf :: [Integer]
equilibradosPerf = filter esEquilibrado cuadrados

-- cuadrados es la lista de los cuadrados perfectos. Por ejemplo,
--    take 10 cuadrados  ==  [0,1,4,9,16,25,36,49,64,81]
cuadrados :: [Integer]
cuadrados = [x^2 | x <- [0..]]
```

```haskell
-- (esEquilibrado x) se verifica si x es equilibrado. Por ejemplo,
--    esEquilibrado 124   ==  True
--    esEquilibrado 1246  ==  False
esEquilibrado :: Integer -> Bool
esEquilibrado x = even a && odd (n-a) where
  cs = show x
  n  = length cs
  a  = length (filter (`elem` "02468") cs)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Calcular el mayor cuadrado perfecto equilibrado de 9
-- cifras.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> last (takeWhile (<=999999999) equilibradosPerf)
--    999950884

-- ---------------------------------------------------------------------
-- Ejercicio 2. Representamos los árboles binarios con elementos en las
-- hojas y en los nodos mediante el tipo de dato
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo,
--    ejArbol :: Arbol Int
--    ejArbol = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))
--
-- Definir la función
--    todasHojas :: (a -> Bool)->  Arbol a -> Bool
-- tal que (todasHojas p a) se verifica si todas las hojas del árbol a
-- satisfacen el predicado p.  Por ejemplo,
--    todasHojas even ejArbol  ==  False
--    todasHojas (<5) ejArbol  ==  True
-- ---------------------------------------------------------------------

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

```haskell
ejArbol :: Arbol Int
ejArbol = N 5 (N 2 (H 1) (H 2)) (N 3 (H 4) (H 2))

todasHojas :: (a -> Bool) -> Arbol a -> Bool
todasHojas p (H x)     = p x
todasHojas p (N _ i d) = todasHojas p i && todasHojas p d

-- ---------------------------------------------------------------------
-- Ejercicio 3. Representamos las pilas mediante el TAD definido en la
-- librería I1M.Pila. Por ejemplo,
--
-- Definir la función
--    elemPila :: Int -> Pila a -> Maybe a
-- tal que (elemPila k p) es el k-ésimo elemento de la pila p, si un tal
-- elemento existe y Nothing, en caso contrario. Por ejemplo, para la
-- pila definida por
--    ejPila :: Pila Int
--    ejPila = foldr apila vacia [2,4,6,8,10]
-- se tiene
--    elemPila 2 ejPila  ==  Just 4
--    elemPila 5 ejPila  ==  Just 10
--    elemPila 7 ejPila  ==  Nothing
-- ---------------------------------------------------------------------

ejPila :: Pila Int
ejPila = foldr apila vacia [2,4,6,8,10]

elemPila :: Int -> Pila a -> Maybe a
elemPila k p
  | esVacia p = Nothing
  | k <= 0    = Nothing
  | k == 1    = Just (cima p)
  | otherwise = elemPila (k-1) (desapila p)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Representamos las matrices mediante la librería
-- Data.Matrix. Por ejemplo,
--    ejMat :: Matrix Int
--    ejMat = fromLists [[1,2,3,0],[4,2,6,7],[7,5,1,11]]
```

```
-- representa la matriz
--    |1 2 3 0 |
--    |4 2 6 7 |
--    |7 5 1 11|
--
-- Definir la función
--   ampliada :: Ord a => Matrix a -> Matrix a
-- tal que (ampliada p) es la matriz obtenida al ampliar cada fila de p
-- con sus elementos mínimo y máximo, respectivamente. Por ejemplo,
--    > ampliada ejMat
--    ( 1 2 3  0 0  3 )
--    ( 4 2 6  7 2  7 )
--    ( 7 5 1 11 1 11 )
-- ------------------------------------------------------------------

ejMat :: Matrix Int
ejMat = fromLists [[1,2,3,0],[4,2,6,7],[7,5,1,11]]

ampliada :: Ord a => Matrix a -> Matrix a
ampliada p = matrix n (m+2) f where
  n = nrows p
  m = ncols p
  f (i,j) | j <= m   = p ! (i,j)
          | j == m+1 = minimum (fila i p)
          | j == m+2 = maximum (fila i p)
  fila i p = [p ! (i,k) | k <- [1..m]]

-- 2ª definición
-- =============

ampliada2 :: Ord a => Matrix a -> Matrix a
ampliada2 p = p <|> minMax p

-- (minMax p) es la matriz cuyas filas son los mínimos y los máximos de
-- la matriz p. Por ejemplo,
--    ghci> minMax ejMat
--    (  0  3 )
--    (  2  7 )
--    (  1 11 )
minMax :: Ord a => Matrix a -> Matrix a
```

```haskell
minMax p =
  fromLists [[V.minimum f, V.maximum f] | i <- [1..nrows p]
                                        , let f = getRow i p]
-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Representamos los polinomios mediante el TAD definido
-- en la librería I1M.Pol. Por ejemplo,
--    ejPol :: Polinomio Int
--    ejPol = consPol 4 5 (consPol 3 6 (consPol 2 (-1) (consPol 0 7 polCero)))
--
-- Definir la función
--    partePar :: (Eq a,Num a) => Polinomio a -> Polinomio a
-- tal que (partePar p) es el polinomio formado por los términos de
-- grado par del polinomio p. Por ejemplo,
--   ghci> partePar ejPol
--   5*x^4 + -1*x^2 + 7
-- ---------------------------------------------------------------------

partePar :: (Eq a,Num a) => Polinomio a -> Polinomio a
partePar p
  | esPolCero p = polCero
  | even n      = consPol n a (partePar r)
  | otherwise   = partePar r
  where n = grado p
        a = coefLider p
        r = restoPol p

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la función
--    generaPol :: [Int] -> Polinomio Int
-- tal que (generaPol xs) es el polinomio de coeficientes enteros de
-- grado la longitud de xs y cuyas raíces son los elementos de xs. Por
-- ejemplo,
--    ghci> generaPol [1,-1]
--    x^2 + -1
--    ghci> generaPol [0,1,1]
--    x^3 + -2*x^2 + 1*x
--    ghci> generaPol [1,1,3,-4,5]
--    x^5 + -6*x^4 + -8*x^3 + 90*x^2 + -137*x + 60
-- ---------------------------------------------------------------------
```

```haskell
-- 1ª solución
generaPol ::  [Int] -> Polinomio Int
generaPol []     = polUnidad
generaPol (r:rs) = multPol (creaFactor r) (generaPol rs)
  where creaFactor r = consPol 1 1 (consPol 0 (-r) polCero)

-- 2ª solución
generaPol2 ::  [Int] -> Polinomio Int
generaPol2 xs =
  foldr multPol polUnidad ps
  where ps = [consPol 1 1 (consPol 0 (-x) polCero) | x <- xs]
```

5.6. Examen 6 (12 de junio de 2017)

El examen es común con el del grupo 4 (ver página 136).

5.7. Examen 7 (29 de junio de 2017)

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

5.8. Examen 8 (8 de septiembre de 2017)

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

5.9. Examen 9 (21 de noviembre de 2017)

El examen es común con el del grupo 1 (ver página 65).

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.