

# Exercitium

## Ejercicios de programación funcional con Haskell

Volumen 1: curso 2013-14

José A. Alonso Jiménez

---

Grupo de Lógica Computacional  
Dpto. de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla  
Sevilla, 11 de diciembre de 2018

Esta obra está bajo una licencia Reconocimiento-NoComercial-CompartirIgual 2.5 Spain de Creative Commons.

**Se permite:**

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

**Bajo las condiciones siguientes:**

**Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor.



**No comercial.** No puede utilizar esta obra para fines comerciales.



**Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

*Para Guiomar*



# Índice general

1	Iguals al siguiente	9
2	Ordenación por el máximo	13
3	La bandera tricolor	15
4	Elementos minimales	19
5	Mastermind	21
6	Primos consecutivos con media capicúa	25
7	Anagramas	27
8	Primos equidistantes	31
9	Suma si todos los valores son justos	35
10	Matrices de Toeplitz	39
11	Máximos locales	41
12	Lista cuadrada	43
13	Segmentos de elementos consecutivos	47
14	Valor de un polinomio mediante vectores	49
15	Ramas de un árbol	51
16	Alfabeto comenzado en un carácter	55
17	Numeración de ternas de naturales	59

---

<b>18 Ordenación de estructuras</b>	<b>61</b>
<b>19 Emparejamiento binario</b>	<b>63</b>
<b>20 Ampliación de columnas de una matriz</b>	<b>65</b>
<b>21 Regiones en el plano</b>	<b>67</b>
<b>22 Elemento más repetido</b>	<b>69</b>
<b>23 Número de pares de elementos adyacentes iguales</b>	<b>71</b>
<b>24 Mayor producto de las ramas de un árbol</b>	<b>75</b>
<b>25 Biparticiones de una lista</b>	<b>81</b>
<b>26 Trenzado de listas</b>	<b>83</b>
<b>27 Números triangulares con n cifras distintas</b>	<b>85</b>
<b>28 Enumeración de árboles binarios</b>	<b>89</b>
<b>29 Elementos con algún vecino menor</b>	<b>91</b>

# Introducción

*"The chief goal of my work as an educator and author is to help people learn to write beautiful programs."*

(Donald Knuth en [Computer programming as an art](#))

Este libro es una recopilación de las soluciones de los ejercicios propuestos en el blog [Exercitium](#)<sup>1</sup> durante el curso 2013-14.

El principal objetivo de Exercitium es servir de complemento a la asignatura de [Informática](#)<sup>2</sup> de 1º del Grado en Matemáticas de la Universidad de Sevilla.

Con los problemas de Exercitium, a diferencia de los de las [relaciones](#)<sup>3</sup>, se pretende practicar con los conocimientos adquiridos durante todo el curso, mientras que con las relaciones están orientadas a los nuevos conocimientos.

Habitualmente de cada ejercicio se muestra distintas soluciones y se compara sus eficiencias.

La dinámica del blog es la siguiente: cada día, de lunes a viernes, se propone un ejercicio para que los alumnos escriban distintas soluciones en los comentarios. Pasado 7 días de la propuesta de cada ejercicio, se cierra los comentarios y se publica una selección de sus soluciones.

Para conocer la cronología de los temas explicados se puede consultar el [diario de clase](#)<sup>4</sup>.

El código del libro se encuentra en [GitHub](#)<sup>5</sup>

---

<sup>1</sup><https://www.glc.us.es/~jalonso/exercitium>

<sup>2</sup><https://www.cs.us.es/~jalonso/cursos/ilm-13>

<sup>3</sup><https://www.cs.us.es/~jalonso/cursos/ilm-13/ejercicios/ejercicios-I1M-2013.pdf>

<sup>4</sup><https://www.glc.us.es/~jalonso/vestigium/category/curso/ilm/ilm2013>

<sup>5</sup><https://github.com/jaalonso/Exercitium2013>





# Ejercicio 1

## Iguales al siguiente

### Enunciado

Definir la función

---

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

---

tal que (igualesAlSiguiente xs) es la lista de los elementos de xs que son iguales a su siguiente. Por ejemplo,

---

```
igualesAlSiguiente [1,2,2,2,3,3,4] == [2,2,3]
igualesAlSiguiente [1..10]         == []
```

---

### Soluciones

```
import Data.List (group)
import Test.QuickCheck

-- 1ª definición (por comprensión):
igualesAlSiguiente :: Eq a => [a] -> [a]
igualesAlSiguiente xs =
  [x | (x,y) <- zip xs (tail xs), x == y]

-- 2ª definición (por recursión):
igualesAlSiguiente2 :: Eq a => [a] -> [a]
```

```
igualesAlSiguiente2 (x:y:zs)
  | x == y      = x : igualesAlSiguiente2 (y:zs)
  | otherwise   = igualesAlSiguiente2 (y:zs)
igualesAlSiguiente2 _ = []

-- 3ª definición (con concat y comprensión):
igualesAlSiguiente3 :: Eq a => [a] -> [a]
igualesAlSiguiente3 xs = concat [ys | (_,ys) <- group xs]

-- 4ª definición (con concat y map):
igualesAlSiguiente4 :: Eq a => [a] -> [a]
igualesAlSiguiente4 xs = concat (map tail (group xs))

-- 5ª definición (con concatMap):
igualesAlSiguiente5 :: Eq a => [a] -> [a]
igualesAlSiguiente5 xs = concatMap tail (group xs)

-- 6ª definición (con concatMap y sin argumentos):
igualesAlSiguiente6 :: Eq a => [a] -> [a]
igualesAlSiguiente6 = concatMap tail . group

-- Equivalencia
-- =====

-- La propiedad es
prop_igualesAlSiguiente_equiv :: [Int] -> Bool
prop_igualesAlSiguiente_equiv xs =
  igualesAlSiguiente xs == igualesAlSiguiente2 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente3 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente4 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente5 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente6 xs

verifica_igualesAlSiguiente_equiv :: IO ()
verifica_igualesAlSiguiente_equiv =
  quickCheck prop_igualesAlSiguiente_equiv

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

-- (0.07 secs, 9,911,528 bytes)



## Ejercicio 2

# Ordenación por el máximo

### Enunciado

Definir la función

---

```
ordenadosPorMaximo :: Ord a => [[a]] -> [[a]]
```

---

tal que (ordenadosPorMaximo xss) es la lista de los elementos de xss ordenada por sus máximos. Por ejemplo,

---

```
ghci> ordenadosPorMaximo [[3,2],[6,7,5],[1,4]]
[[3,2],[1,4],[6,7,5]]
ghci> ordenadosPorMaximo ["este","es","el","primero"]
["el","primero","es","este"]
```

---

### Soluciones

```
import Data.List (sort)
import GHC.Exts  (sortWith)
import Test.QuickCheck

-- 1ª definición
ordenadosPorMaximo :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo xss =
  map snd (sort [(maximum xs,xs) | xs <- xss])
```

```

-- 2ª definición
ordenadosPorMaximo2 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo2 xss =
  [xs | (_,xs) <- sort [(maximum xs,xs) | xs <- xss]]

-- 3ª definición:
ordenadosPorMaximo3 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo3 = sortWith maximum

-- Equivalencia
-- =====

verificaOrdenadosPorMaximo :: IO ()
verificaOrdenadosPorMaximo =
  quickCheck prop_ordenadosPorMaximo

prop_ordenadosPorMaximo :: [[Int]] -> Bool
prop_ordenadosPorMaximo xs =
  ordenadosPorMaximo ys == ordenadosPorMaximo2 ys
  where ys = filter (not . null) xs

-- Comprobación
--    λ> verificaOrdenadosPorMaximo
--    +++ OK, passed 100 tests.

```

## Ejercicio 3

# La bandera tricolor

### Enunciado

El problema de la bandera tricolor consiste en lo siguiente: Dada una lista de objetos `xs` que pueden ser rojos, amarillos o morados, se pide devolver una lista `ys` que contiene los elementos de `xs`, primero los rojos, luego los amarillos y por último los morados.

Se pide definir el tipo de dato `Color` para representar los colores con los constructores `R`, `A` y `M` correspondientes al rojo, amarillo y morado y la función

---

```
banderaTricolor :: [Color] -> [Color]
```

---

tal que `(banderaTricolor xs)` es la bandera tricolor formada con los elementos de `xs`. Por ejemplo,

---

```
bandera [M,R,A,A,R,R,A,M,M] == [R,R,R,A,A,A,M,M,M]
bandera [M,R,A,R,R,A]       == [R,R,R,A,A,M]
```

---

### Soluciones

```
import Data.List (sort)
import Test.QuickCheck

data Color = R | A | M
```

```

deriving (Show, Eq, Ord, Enum)

-- 1ª definición (con sort):
banderaTricolor :: [Color] -> [Color]
banderaTricolor = sort

-- 2ª definición (por comprensión):
banderaTricolor2 :: [Color] -> [Color]
banderaTricolor2 xs =
  [x | x <- xs, x == R] ++ [x | x <- xs, x == A] ++ [x | x <- xs, x == M]

-- 3ª definición (por comprensión y concat):
banderaTricolor3 :: [Color] -> [Color]
banderaTricolor3 xs =
  concat [[x | x <- xs, x == c] | c <- [R,A,M]]

-- 4ª definición (por recursión):
banderaTricolor4 :: [Color] -> [Color]
banderaTricolor4 xs = aux xs ([],[],[ ])
  where aux []      (rs,as,ms) = rs ++ as ++ ms
        aux (R:ys) (rs,as,ms) = aux ys (R:rs, as, ms)
        aux (A:ys) (rs,as,ms) = aux ys ( rs, A:as, ms)
        aux (M:ys) (rs,as,ms) = aux ys ( rs, as, M:ms)

-- 5ª definición (por recursión):
banderaTricolor5 :: [Color] -> [Color]
banderaTricolor5 xs = aux xs (0,0,0)
  where aux []      (as,rs,ms) = replicate rs R ++
                                replicate as A ++
                                replicate ms M
        aux (A:ys) (as,rs,ms) = aux ys (1+as, rs, ms)
        aux (R:ys) (as,rs,ms) = aux ys ( as, 1+rs, ms)
        aux (M:ys) (as,rs,ms) = aux ys ( as, rs, 1+ms)

-- Equivalencia
-- =====

instance Arbitrary Color where
  arbitrary = elements [R,A,M]

```



```
prop_banderaTricolor :: [Color] -> Bool
prop_banderaTricolor xs =
  all (== banderaTricolor xs)
    [f xs | f <- [ banderaTricolor2
                  , banderaTricolor3
                  , banderaTricolor4
                  , banderaTricolor5]]

verifica_banderaTricolor :: IO ()
verifica_banderaTricolor =
  quickCheck prop_banderaTricolor

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

-- Comparaciones:
--   ghci> bandera n = concat [replicate n c | c <- [M,R,A]]
--
--   ghci> length (banderaTricolor (bandera 1000000))
--   3000000
--   (2.65 secs, 312128100 bytes)
--
--   ghci> length (banderaTricolor2 (bandera 1000000))
--   3000000
--   (7.78 secs, 512387912 bytes)
--
--   ghci> length (banderaTricolor3 (bandera 1000000))
--   3000000
--   (7.84 secs, 576080444 bytes)
--
--   ghci> length (banderaTricolor4 (bandera 1000000))
--   3000000
--   (3.76 secs, 476484220 bytes)
--
--   ghci> length (banderaTricolor5 (bandera 1000000))
--   3000000
--   (4.45 secs, 622205356 bytes)
```



## Ejercicio 4

# Elementos minimales

### Enunciado

Definir la función

---

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

---

tal que (minimales xss) es la lista de los elementos de xss que no están contenidos en otros elementos de xss. Por ejemplo,

---

```
minimales [[1,3],[2,3,1],[3,2,5]]      == [[2,3,1],[3,2,5]]
minimales [[1,3],[2,3,1],[3,2,5],[3,1]] == [[2,3,1],[3,2,5]]
```

---

### Soluciones

```
import Data.List (delete, nub)
```

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

```
minimales xss =
```

```
  [xs | xs <- xss, [ys | ys <- xss, subconjuntoPropio xs ys] == []]
```

```
-- (subconjuntoPropio xs ys) se verifica si xs es un subconjunto propio
-- de ys. Por ejemplo,
```

```
-- subconjuntoPropio [1,3] [3,1,3]      == False
```

```
-- subconjuntoPropio [1,3,1] [3,1,2]    == True
```

```
subconjuntoPropio :: Eq a => [a] -> [a] -> Bool
subconjuntoPropio xs ys = subconjuntoPropio' (nub xs) (nub ys)
  where
    subconjuntoPropio' _ [] = False
    subconjuntoPropio' [] _ = True
    subconjuntoPropio' (x:xs') ys' =
      x `elem` ys' && subconjuntoPropio xs' (delete x ys')
```

# Ejercicio 5

## Mastermind

### Enunciado

El Mastermind es un juego que consiste en deducir un código numérico formado por una lista de números distintos. Cada vez que se empieza una partida, el programa debe elegir un código, que será lo que el jugador debe adivinar en la menor cantidad de intentos posibles. Cada intento consiste en una propuesta de un código posible que propone el jugador, y una respuesta del programa. Las respuestas le darán pistas al jugador para que pueda deducir el código.

Estas pistas indican cuán cerca estuvo el número propuesto de la solución a través de dos valores: la cantidad de aciertos es la cantidad de dígitos que propuso el jugador que también están en el código en la misma posición. La cantidad de coincidencias es la cantidad de dígitos que propuso el jugador que también están en el código pero en una posición distinta.

Por ejemplo, si el código que eligió el programa es el [2,6,0,7], y el jugador propone el [1,4,0,6], el programa le debe responder un acierto (el 0, que está en el código original en el mismo lugar, el tercero), y una coincidencia (el 6, que también está en el código original, pero en la segunda posición, no en el cuarto como fue propuesto). Si el jugador hubiera propuesto el [3,5,9,1], habría obtenido como respuesta ningún acierto y ninguna coincidencia, ya que no hay números en común con el código original, y si se obtienen cuatro aciertos es porque el jugador adivinó el código y ganó el juego.

Definir la función

---

```
mastermind :: [Int] -> [Int] -> (Int,Int)
```

---

tal que (mastermind xs ys) es el par formado por los números de aciertos y de coincidencias entre xs e ys. Por ejemplo,

---

```

mastermind [2,6,0,7] [1,4,0,6] == (1,1)
mastermind [2,6,0,7] [3,5,9,1] == (0,0)
mastermind [2,6,0,7] [1,6,0,4] == (2,0)
mastermind [2,6,0,7] [2,6,0,7] == (4,0)

```

---

## Soluciones

```

import Test.QuickCheck
import Data.List (nub)

```

```

-- 1ª solución (por comprensión):

```

```

mastermind :: [Int] -> [Int] -> (Int,Int)

```

```

mastermind xs ys =
    (length (aciertos xs ys), length (coincidencias xs ys))

```

```

-- (aciertos xs ys) es la lista de aciertos entre xs e ys. Por ejemplo,

```

```

--   aciertos [2,6,0,7] [1,4,0,6] == [0]

```

```

aciertos :: Eq a => [a] -> [a] -> [a]

```

```

aciertos xs ys = [x | (x,y) <- zip xs ys, x == y]

```

```

-- (coincidencia xs ys) es la lista de coincidencias entre xs e ys. Por
-- ejemplo,

```

```

--   coincidencias [2,6,0,7] [1,4,0,6] == [6]

```

```

coincidencias :: Eq a => [a] -> [a] -> [a]

```

```

coincidencias xs ys =

```

```

    [x | x <- xs, x `elem` ys, x `notElem` zs]

```

```

    where zs = aciertos xs ys

```

```

-- 2ª solución (por recursión):

```

```

mastermind2 :: [Int] -> [Int] -> (Int,Int)

```

```

mastermind2 xs ys = aux xs ys

```

```

    where aux [] [] = (0,0)

```

```

          aux (x:xs') (z:zs)

```

```

              | x == z      = (a+1,b)

```

```

              | x `elem` ys = (a,b+1)

```

```

              | otherwise   = (a,b)

```

```

        where (a,b) = aux xs' zs
        aux _ _ = error "Imposible"

-- 3ª solución:
mastermind3 :: [Int] -> [Int] -> (Int,Int)
mastermind3 xs ys = (nAciertos,nCoincidencias)
    where nAciertos = length [(x,y) | (x,y) <- zip xs ys, x == y]
          nCoincidencias = length (xs++ys) - length (nub (xs++ys)) - nAciertos

-- Equivalencia
-- =====

prop_mastermind :: [Int] -> [Int] -> Bool
prop_mastermind xs ys =
    all (== mastermind cs ds)
        [f cs ds | f <- [ mastermind2
                          , mastermind3]]
    where as = nub xs
          bs = nub ys
          n   = min (length as) (length bs)
          cs = take n as
          ds = take n bs

verifica_mastermind :: IO ()
verifica_mastermind =
    quickCheck prop_mastermind

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

```





## Ejercicio 6

# Primos consecutivos con media capicúa

### Enunciado

La pasada semana, Antonio Roldán publicó en [Twitter](#) la siguiente observación:

Los pares de primos consecutivos (97,101) y (109,113) son los más pequeños, con promedio capicúa con más de una cifra:  $(97+101)/2=99$  y  $(109+113)/2=111$ .

A partir de ella, se propone el ejercicio de hoy.

Definir la constante

---

```
primosConsecutivosConMediaCapicua :: [(Int,Int,Int)]
```

---

tal que `primosConsecutivosConMediaCapicua` es la lista de las ternas (x,y,z) tales que x e y son primos consecutivos tales que su media, z, es capicúa. Por ejemplo,

---

```
ghci> take 5 primosConsecutivosConMediaCapicua  
[(3,5,4),(5,7,6),(7,11,9),(97,101,99),(109,113,111)]
```

---

Calcular cuántos hay anteriores a 2014.

## Soluciones

```
import Data.Numbers.Primes (primes)

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

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

-- El cálculo es
-- λ> length (takeWhile (\(_,y,_) -> y < 2014) primosConsecutivosConMediaCapicua)
-- 20
```

# Ejercicio 7

## Anagramas

### Enunciado

Una palabra es una anagrama de otra si se puede obtener permutando sus letras. Por ejemplo, mora y roma son anagramas de amor.

Definir la función

---

```
anagramas :: String -> [String] -> [String]
```

---

tal que (anagramas x ys) es la lista de los elementos de ys que son anagramas de x. Por ejemplo,

---

```
ghci> anagramas "amor" ["Roma","mola","loma","moRa", "rama"]
["Roma","moRa"]
ghci> anagramas "rama" ["aMar","amaRa","roMa","marr","aRma"]
["aMar","aRma"]
```

---

### Soluciones

```
import Data.List      (sort)
import Data.Char      (toLower)
import Data.Function  (on)

-- 1ª definición (por recursión)
```

```

-- =====

anagramas :: String -> [String] -> [String]
anagramas _ [] = []
anagramas x (y:ys)
  | sonAnagramas x y = y : anagramas x ys
  | otherwise       = anagramas x ys

-- (sonAnagramas xs ys) se verifica si xs e ys son anagramas. Por
-- ejemplo,
--   sonAnagramas "amor" "Roma" == True
--   sonAnagramas "amor" "mola" == False
sonAnagramas :: String -> String -> Bool
sonAnagramas xs ys =
  sort (map toLower xs) == sort (map toLower ys)

-- 2ª definición de sonAnagramas
sonAnagramas2 :: String -> String -> Bool
sonAnagramas2 xs ys =
  (sort . map toLower) xs == (sort . map toLower) ys

-- 3ª definición de sonAnagramas (con on)
sonAnagramas3 :: String -> String -> Bool
sonAnagramas3 = (==) `on` (sort . map toLower)

-- Nota. En la definición anterior se usa la función on ya que
--   (f `on` g) x y
-- es equivalente a
--   f (g x) (g y)
-- Por ejemplo,
--   ghci> ((*) `on` (+2)) 3 4
--   30

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

anagramas2 :: String -> [String] -> [String]
anagramas2 x ys = [y | y <- ys, sonAnagramas x y]

-- 3ª definición (con filter y sin el 2ª argumento)

```

```
-- =====  
  
anagramas3 :: String -> [String] -> [String]  
anagramas3 x = filter (`sonAnagramas` x)  
  
-- 4ª definición (sin sonAnagramas ni el 2º argumento)  
-- =====  
  
anagramas4 :: String -> [String] -> [String]  
anagramas4 x = filter (((==) `on` (sort . map toLower)) x)
```



# Ejercicio 8

## Primos equidistantes

### Enunciado

Definir la función

---

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

---

tal que (primosEquidistantes k) es la lista de los pares de primos cuya diferencia es k. Por ejemplo,

---

```
take 3 (primosEquidistantes 2) == [(3,5),(5,7),(11,13)]
take 3 (primosEquidistantes 4) == [(7,11),(13,17),(19,23)]
take 3 (primosEquidistantes 6) == [(23,29),(31,37),(47,53)]
take 3 (primosEquidistantes 8) == [(89,97),(359,367),(389,397)]
```

---

### Soluciones

```
import Data.Numbers.Primes (primes)
```

```
-- 1ª solución
```

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

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

```

        | otherwise = aux (y:ps)

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

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

-- 2ª solución
-- =====

primosEquidistantes2 :: Integer -> [(Integer,Integer)]
primosEquidistantes2 k = aux primes
  where aux (x:y:ps) | y - x == k = (x,y) : aux (y:ps)
        | otherwise = aux (y:ps)

-- 3ª solución
-- =====

primosEquidistantes3 :: Integer -> [(Integer,Integer)]
primosEquidistantes3 k =
  [(x,y) | (x,y) <- zip primes (tail primes)
    , y - x == k]

-- Comparación de eficiencia
-- =====

--   λ> primosEquidistantes 2 !! 200
--   (9677,9679)
--   (6.30 secs, 896,925,344 bytes)
--   λ> primosEquidistantes2 2 !! 200
--   (9677,9679)
--   (0.02 secs, 3,622,808 bytes)
--   λ> primosEquidistantes3 2 !! 200
--   (9677,9679)

```



```
--      (0.03 secs, 6,398,168 bytes)
--
--      λ> primosEquidistantes2 2 !! 20000
--      (2840447,2840449)
--      (2.46 secs, 1,142,499,552 bytes)
--      λ> primosEquidistantes3 2 !! 20000
--      (2840447,2840449)
--      (4.32 secs, 2,207,828,128 bytes)
```



## Ejercicio 9

# Suma si todos los valores son justos

### Enunciado

Definir la función

---

```
sumaSiTodosJustos :: (Num a, Eq a) => [Maybe a] -> Maybe a
```

---

tal que (sumaSiTodosJustos xs) es justo la suma de todos los elementos de xs si todos son justos (es decir, si Nothing no pertenece a xs) y Nothing en caso contrario. Por ejemplo,

---

```
sumaSiTodosJustos [Just 2, Just 5]           == Just 7
sumaSiTodosJustos [Just 2, Just 5, Nothing] == Nothing
```

---

### Soluciones

```
import Data.Maybe
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
sumaSiTodosJustos :: (Num a, Eq a) => [Maybe a] -> Maybe a
```

```

sumaSiTodosJustos xs
  | todosJustos xs = Just (sum [x | (Just x) <- xs])
  | otherwise      = Nothing

-- (todosJustos xs) se verifica si todos los elementos de xs son justos
-- (es decir, si Nothing no pertenece a xs) y Nothing en caso
-- contrario. Por ejemplo,
--   todosJustos [Just 2, Just 5]           == True
--   todosJustos [Just 2, Just 5, Nothing] == False
todosJustos :: Eq a => [Maybe a] -> Bool
todosJustos = notElem Nothing

-- 2ª solución
-- =====

sumaSiTodosJustos2 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos2 xs
  | todosJustos2 xs = Just (sum [x | (Just x) <- xs])
  | otherwise       = Nothing

todosJustos2 :: Eq a => [Maybe a] -> Bool
todosJustos2 = all isJust

-- 3ª solución
-- =====

sumaSiTodosJustos3 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos3 xs
  | todosJustos xs = Just (sum [fromJust x | x <- xs])
  | otherwise      = Nothing

-- 4ª solución
-- =====

sumaSiTodosJustos4 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos4 xs
  | todosJustos xs = Just (sum (catMaybes xs))
  | otherwise      = Nothing

-- 5ª solución

```

```

-- =====

sumaSiTodosJustos5 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos5 xs = suma (sequence xs)
  where suma Nothing  = Nothing
        suma (Just ys) = Just (sum ys)

-- Nota. En la solución anterior se usa la función
--   sequence :: Monad m => [m a] -> m [a]
-- tal que (sequence xs) es la mónada obtenida evaluando cada una de las
-- de xs de izquierda a derecha. Por ejemplo,
--   sequence [Just 2, Just 5]    == Just [2,5]
--   sequence [Just 2, Nothing]   == Nothing
--   sequence [[2,4],[5,7]]      == [[2,5],[2,7],[4,5],[4,7]]
--   sequence [[2,4],[5,7],[6]]  == [[2,5,6],[2,7,6],[4,5,6],[4,7,6]]
--   sequence [[2,4],[5,7],[]]   == []

-- 6ª solución
-- =====

sumaSiTodosJustos6 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos6 xs = fmap sum (sequence xs)

-- 7ª solución
-- =====

sumaSiTodosJustos7 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos7 = fmap sum . sequence

-- Equivalencia
-- =====

-- La propiedad es
prop_sumaSiTodosJustos :: [Maybe Int] -> Bool
prop_sumaSiTodosJustos xs =
  all (== sumaSiTodosJustos xs)
    [f xs | f <- [ sumaSiTodosJustos2
                  , sumaSiTodosJustos3
                  , sumaSiTodosJustos4
                  , sumaSiTodosJustos5

```

```
, sumaSiTodosJustos6  
, sumaSiTodosJustos7  
]]
```

```
-- La comprobación es  
--  $\lambda > \text{quickCheck prop\_sumaSiTodosJustos}$   
-- +++ OK, passed 100 tests.
```

# Ejercicio 10

## Matrices de Toeplitz

### Enunciado

Una matriz de Toeplitz es una matriz cuadrada que es constante a lo largo de las diagonales paralelas a la diagonal principal. Por ejemplo,

---

2 5 1 6	2 5 1 6
4 2 5 1	4 2 6 1
7 4 2 5	7 4 2 5
9 7 4 2	9 7 4 2

---

la primera es una matriz de Toeplitz y la segunda no lo es.

Las anteriores matrices se pueden definir por

---

```
ej1, ej2 :: Array (Int,Int) Int
ej1 = listArray ((1,1),(4,4)) [2,5,1,6,4,2,5,1,7,4,2,5,9,7,4,2]
ej2 = listArray ((1,1),(4,4)) [2,5,1,6,4,2,6,1,7,4,2,5,9,7,4,2]
```

---

Definir la función

---

```
esToeplitz :: Eq a => Array (Int,Int) a -> Bool
```

---

tal que (esToeplitz p) se verifica si la matriz p es de Toeplitz. Por ejemplo,

---

```
esToeplitz ej1 == True
esToeplitz ej2 == False
```

---

## Soluciones

```
import Data.Array
```

```
ej1, ej2 :: Array (Int,Int) Int
```

```
ej1 = listArray ((1,1),(4,4)) [2,5,1,6,4,2,5,1,7,4,2,5,9,7,4,2]
```

```
ej2 = listArray ((1,1),(4,4)) [2,5,1,6,4,2,6,1,7,4,2,5,9,7,4,2]
```

```
esToeplitz :: Eq a => Array (Int,Int) a -> Bool
```

```
esToeplitz p = m == n &&  
    and [ p!(i,j) == p!(i+1,j+1)  
        | i <- [1..n-1]  
        , j <- [1..n-1]]
```

```
where (_,(m,n)) = bounds p
```



# Ejercicio 11

## Máximos locales

### Enunciado

Un máximo local de una lista es un elemento de la lista que es mayor que su predecesor y que su sucesor en la lista. Por ejemplo, 5 es un máximo local de [3,2,5,3,7,7,1,6,2] ya que es mayor que 2 (su predecesor) y que 3 (su sucesor).

Definir la función

---

```
maximosLocales :: Ord a => [a] -> [a]
```

---

tal que (maximosLocales xs) es la lista de los máximos locales de la lista xs. Por ejemplo,

---

```
maximosLocales [3,2,5,3,7,7,1,6,2] == [5,6]
maximosLocales [1..100]             == []
maximosLocales "adbpmqexyz"         == "dpq"
```

---

### Soluciones

```
import Test.QuickCheck
```

```
-- 1ª solución
```

```
maximosLocales :: Ord a => [a] -> [a]
```

```

maximosLocales (x:y:z:xs) | y > x && y > z = y : maximosLocales (z:xs)
                        | otherwise      = maximosLocales (y:z:xs)
maximosLocales _                = []

```

-- 2ª solución

```
maximosLocales2 :: Ord a => [a] -> [a]
```

```
maximosLocales2 xs =
```

```
  [y | (x,y,z) <- zip3 xs (tail xs) (drop 2 xs), y > x, y > z]
```

-- Equivalencia

-- =====

-- La propiedad es

```
prop_maximosLocales :: [Int] -> Bool
```

```
prop_maximosLocales xs =
```

```
  maximosLocales xs == maximosLocales2 xs
```

-- La comprobación es

```
-- λ> quickCheck prop_maximosLocales
```

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

-- Comparación de eficiencia

-- =====

```
-- λ> maximosLocales [1..10^6]
```

```
-- []
```

```
-- (2.50 secs, 464,396,256 bytes)
```

```
-- λ> maximosLocales2 [1..10^6]
```

```
-- []
```

```
-- (1.36 secs, 280,395,256 bytes)
```

```
--
```

```
-- λ> maximosLocales (concat (replicate 10 [1..10^5]))
```

```
-- [100000,100000,100000,100000,100000,100000,100000,100000,100000]
```

```
-- (2.55 secs, 455,647,560 bytes)
```

```
-- λ> maximosLocales2 (concat (replicate 10 [1..10^5]))
```

```
-- [100000,100000,100000,100000,100000,100000,100000,100000,100000]
```

```
-- (1.34 secs, 271,650,096 bytes)
```

# Ejercicio 12

## Lista cuadrada

### Enunciado

Definir la función

---

```
listaCuadrada :: Int -> a -> [a] -> [[a]]
```

---

tal que (listaCuadrada n x xs) es una lista de n listas de longitud n formadas con los elementos de xs completada con x, si no xs no tiene suficientes elementos. Por ejemplo,

---

```
listaCuadrada 3 7 [0,3,5,2,4] == [[0,3,5],[2,4,7],[7,7,7]]
listaCuadrada 3 7 [0..]       == [[0,1,2],[3,4,5],[6,7,8]]
listaCuadrada 2 'p' "eva"     == ["ev","ap"]
listaCuadrada 2 'p' ['a'..]   == ["ab","cd"]
```

---

### Soluciones

```
import Test.QuickCheck
```

```
-- 1ª solución
```

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

```
listaCuadrada :: Int -> a -> [a] -> [[a]]
```

```
listaCuadrada n x xs =
```

```

take n (grupos n (xs ++ repeat x))

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

-- 2ª solución
-- =====

listaCuadrada2 :: Int -> a -> [a] -> [[a]]
listaCuadrada2 n x xs =
  take n [take n (drop m xs ++ repeat x)
         | m <- [0,n..n*n]]

-- 3ª solución
-- =====

listaCuadrada3 :: Int -> a -> [a] -> [[a]]
listaCuadrada3 n x xs =
  take n [take n ys | ys <- iterate (drop n) (xs ++ repeat x)]

-- 4ª solución
-- =====

listaCuadrada4 :: Int -> a -> [a] -> [[a]]
listaCuadrada4 n x =
  take n . map (take n) . iterate (drop n) . (++ repeat x)

-- Equivalencia
-- =====

-- La propiedad es
prop_listaCuadrada :: NonNegative Int -> Int -> [Int] -> Bool
prop_listaCuadrada (NonNegative n) x xs =
  all (== listaCuadrada n x xs)

```

```
[f n x xs | f <- [ listaCuadrada2
                  , listaCuadrada3
                  , listaCuadrada4
                  ]]

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



## Ejercicio 13

# Segmentos de elementos consecutivos

### Enunciado

Definir la función

---

```
segmentos :: (Enum a, Eq a) => [a] -> [[a]]
```

---

tal que (segmentos xss) es la lista de los segmentos 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 "abbccddeeabc" == ["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'
```

---

### Soluciones

```
import Test.QuickCheck
```

```

-- 1ª solución
-- =====

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

-- 2ª solución
-- =====

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,_) <- takeWhile (\(u,v) -> u == v) (zip (x:xs) [x..])]

```



## Ejercicio 14

# Valor de un polinomio mediante vectores

### Enunciado

Los polinomios se pueden representar mediante vectores usando la librería Data.Array. En primer lugar, se define el tipo de los polinomios (con coeficientes de tipo a) mediante

---

```
type Polinomio a = Array Int a
```

---

Como ejemplos, definimos el polinomio

---

```
ej_pol1 :: Array Int Int
ej_pol1 = array (0,4) [(1,2),(2,-5),(4,7),(0,6),(3,0)]
```

---

que representa a  $2x - 5x^2 + 7x^4 + 6$  y el polinomio

---

```
ej_pol2 :: Array Int Double
ej_pol2 = array (0,4) [(1,2),(2,-5.2),(4,7),(0,6.5),(3,0)]
```

---

que representa a  $2x - 5,2x^2 + 7x^4 + 6,5$

Definir la función

---

```
valor :: Num a => Polinomio a -> a -> a
```

---

tal que (valor p b) es el valor del polinomio p en el punto b. Por ejemplo,

---

```
valor ej_pol1 0 == 6
valor ej_pol1 1 == 10
valor ej_pol1 2 == 102
valor ej_pol2 0 == 6.5
valor ej_pol2 1 == 10.3
valor ej_pol2 3 == 532.7
```

---

## Soluciones

```
import Data.Array
```

```
type Polinomio a = Array Int a
```

```
ej_pol1, ej_pol2 :: Array Int Double
```

```
ej_pol1 = array (0,4) [(1,2),(2,-5),(4,7),(0,6),(3,0)]
```

```
ej_pol2 = array (0,4) [(1,2),(2,-5.2),(4,7),(0,6.5),(3,0)]
```

```
-- 1ª solución
```

```
valor :: Num a => Polinomio a -> a -> a
```

```
valor p b = sum [(p!i)*b^i | i <- [0..n]]
```

```
  where (_,n) = bounds p
```

```
-- 2ª solución
```

```
valor2 :: Num a => Polinomio a -> a -> a
```

```
valor2 p b = sum [v*b^i | (i,v) <- assocs p]
```

## Ejercicio 15

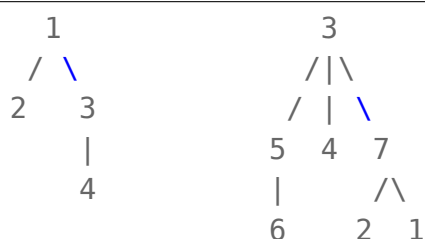
### Ramas de un árbol

#### Enunciado

Los árboles se pueden representar mediante el siguiente tipo de datos

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

Por ejemplo, los árboles



se representan por

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

Definir la función

```
ramas :: Arbol b -> [[b]]
```

tal que (ramas a) es la lista de las ramas del árbol a. Por ejemplo,

---

```
ramas ej1 == [[1,2],[1,3,4]]
ramas ej2 == [[3,5,6],[3,4],[3,7,2],[3,7,1]]
```

---

## Soluciones

```
import Test.QuickCheck
```

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

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

```
-- 1ª solución
```

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

```
-- 2ª solución
```

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

```
-- 3ª solución
```

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

```
-- 4ª solución
```

```
ramas4 :: Arbol b -> [[b]]
ramas4 (N x []) = [[x]]
ramas4 (N x as) = concatMap (map (x :) . ramas4) as
```

```
-- 5ª solución
```

```
ramas5 :: Arbol a -> [[a]]
ramas5 (N x []) = [[x]]
```

```

ramas5 (N x xs) = map ramas5 xs >=> map (x :)

-- Equivalencia
-- =====

-- Generador de árboles.
-- > sample ((gen_Arbol 5) :: Gen (Arbol Int))
-- N 0 [N 0 []]
-- N (-2) []
-- N 4 []
-- N 2 [N 4 []]
-- N 8 []
-- N (-2) [N (-9) [], N 7 []]
-- N 11 []
-- N (-11) [N 4 [], N 14 []]
-- N 10 [N (-3) [], N 13 []]
-- N 12 [N 11 []]
-- N 20 [N (-18) [], N (-13) []]
gen_Arbol :: Arbitrary a => Int -> Gen (Arbol a)
gen_Arbol m = do
  x <- arbitrary
  n <- choose (0, m `div` 2)
  xs <- vectorOf n (gen_Arbol (m `div` 4))
  return (N x xs)

-- Incluye los árboles en Arbitrary.
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary =
    sized gen_Arbol

-- La propiedad es
prop_ramas :: Arbol Int -> Bool
prop_ramas a =
  all (== ramas a)
    [f a | f <- [ ramas2
                  , ramas3
                  , ramas4
                  , ramas5
                ]]

```

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

## Ejercicio 16

# Alfabeto comenzado en un carácter

### Enunciado

Definir la función

---

```
alfabetoDesde :: Char -> String
```

---

tal que (alfabetoDesde c) es el alfabeto, en minúscula, comenzando en el carácter c, si c es una letra minúscula y comenzando en 'a', en caso contrario. Por ejemplo,

---

```
alfabetoDesde 'e' == "efghijklmnopqrstuvwxyzabcd"
alfabetoDesde 'a' == "abcdefghijklmnopqrstuvwxyz"
alfabetoDesde '7' == "abcdefghijklmnopqrstuvwxyz"
alfabetoDesde '{' == "abcdefghijklmnopqrstuvwxyz"
alfabetoDesde 'B' == "abcdefghijklmnopqrstuvwxyz"
```

---

### Soluciones

```
import Data.Char (isAsciiLower, ord)
import Data.Tuple (swap)
import Test.QuickCheck
```

-- 1ª solución

**alfabetoDesde :: Char -> String**

**alfabetoDesde** c =  
     dropWhile (<c) ['a'..'z'] ++ takeWhile (<c) ['a'..'z']

-- 2ª solución

**alfabetoDesde2 :: Char -> String**

**alfabetoDesde2** c = ys ++ xs  
     **where** (xs,ys) = span (<c) ['a'..'z']

-- 3ª solución

**alfabetoDesde3 :: Char -> String**

**alfabetoDesde3** c = ys ++ xs  
     **where** (xs,ys) = break (==c) ['a'..'z']

-- 4ª solución

**alfabetoDesde4 :: Char -> String**

**alfabetoDesde4** = uncurry (++) . swap . flip span ['a'..'z'] . (>)

-- Ejemplo de cálculo:

```
-- alfabetoDesde4 'e'
-- = (uncurry (++) . swap . flip span ['a'..'z'] . (>)) 'e'
-- = (uncurry (++) . swap) ("abcd","efghijklmnopqrstuvwxyz")
-- = uncurry (++) ("efghijklmnopqrstuvwxyz","abcd")
-- = (++) "efghijklmnopqrstuvwxyz" "abcd"
-- = "efghijklmnopqrstuvwxyzabcd"
```

-- 5ª solución

**alfabetoDesde5 :: Char -> String**

**alfabetoDesde5** = uncurry (flip (++) . (`break` ['a'..'z'])) . (==)

-- Ejemplo de cálculo:

```
-- alfabetoDesde5 'e'
-- = (uncurry (flip (++) . (`break` ['a'..'z'])) . (==)) 'e'
-- = uncurry (flip (++) . (`break` ['a'..'z'])) ("abcd","efghijklmnopqrstuvwxyz")
-- = flip (++) "abcd" "efghijklmnopqrstuvwxyz"
-- = "efghijklmnopqrstuvwxyz" ++ "abcd"
-- = "efghijklmnopqrstuvwxyzabcd"
```

-- 6ª solución



```
alfabetoDesde6 :: Char -> String
alfabetoDesde6 c
  | c >= 'a' && c <= 'z' = [c..'z'] ++ ['a'..pred c]
  | otherwise            = ['a'..'z']
```

```
-- 7ª solución
```

```
alfabetoDesde7 :: Char -> String
alfabetoDesde7 c
  | isAsciiLower c = [c..'z'] ++ ['a'..pred c]
  | otherwise      = ['a'..'z']
```

```
-- Equivalencia
```

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

```
-- La propiedad es
```

```
prop_alfabetoDesde :: Char -> Bool
```

```
prop_alfabetoDesde c =
  all (== alfabetoDesde c)
    [f c | f <- [ alfabetoDesde2
                  , alfabetoDesde3
                  , alfabetoDesde4
                  , alfabetoDesde5
                  , alfabetoDesde6
                  , alfabetoDesde7
                  ]]
```

```
-- La comprobación es
```

```
-- λ> quickCheck prop_alfabetoDesde
```

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



## Ejercicio 17

# Numeración de ternas de naturales

### Enunciado

Las ternas de números naturales se pueden ordenar como sigue

---

```
(0,0,0),  
(0,0,1), (0,1,0), (1,0,0),  
(0,0,2), (0,1,1), (0,2,0), (1,0,1), (1,1,0), (2,0,0),  
(0,0,3), (0,1,2), (0,2,1), (0,3,0), (1,0,2), (1,1,1), (1,2,0), (2,0,1), (2,1,0), (3,0,0),  
...
```

---

Definir la función

---

```
posicion :: (Int,Int,Int) -> Int
```

---

tal que (posicion (x,y,z)) es la posición de la terna de números naturales (x,y,z) en la ordenación anterior. Por ejemplo,

---

```
posicion (0,1,0) == 2  
posicion (0,0,2) == 4  
posicion (0,1,1) == 5
```

---

## Soluciones

```
posicion :: (Int,Int,Int) -> Int
posicion (x,y,z) = length (takeWhile (/= (x,y,z)) ternas)

-- ternas es la lista ordenada de las ternas de números naturales. Por ejemplo,
--     ghci> take 10 ternas
--     [(0,0,0),(0,0,1),(0,1,0),(1,0,0),(0,0,2),(0,1,1),(0,2,0),(1,0,1),(1,1,0),(2,0,0)]
ternas :: [(Int,Int,Int)]
ternas = [(x,y,n-x-y) | n <- [0..], x <- [0..n], y <- [0..n-x]]
```

# Ejercicio 18

## Ordenación de estructuras

### Enunciado

Las notas de los dos primeros exámenes se pueden representar mediante el siguiente tipo de dato

---

```
data Notas = Notas String Int Int
    deriving (Read, Show, Eq)
```

---

Por ejemplo, (Notas "Juan" 6 5) representar las notas de un alumno cuyo nombre es Juan, la nota del primer examen es 6 y la del segundo es 5.

Definir la función

---

```
ordenadas :: [Notas] -> [Notas]
```

---

tal que (ordenadas ns) es la lista de las notas ns ordenadas considerando primero la nota del examen 2, a continuación la del examen 1 y finalmente el nombre. Por ejemplo,

---

```
ghci> ordenadas [Notas "Juan" 6 5, Notas "Luis" 3 7]
[Notas "Juan" 6 5, Notas "Luis" 3 7]
ghci> ordenadas [Notas "Juan" 6 5, Notas "Luis" 3 4]
[Notas "Luis" 3 4, Notas "Juan" 6 5]
ghci> ordenadas [Notas "Juan" 6 5, Notas "Luis" 7 4]
[Notas "Luis" 7 4, Notas "Juan" 6 5]
ghci> ordenadas [Notas "Juan" 6 4, Notas "Luis" 7 4]
[Notas "Juan" 6 4, Notas "Luis" 7 4]
```

---

```
ghci> ordenadas [Notas "Juan" 6 4, Notas "Luis" 5 4]
[Notas "Luis" 5 4, Notas "Juan" 6 4]
ghci> ordenadas [Notas "Juan" 5 4, Notas "Luis" 5 4]
[Notas "Juan" 5 4, Notas "Luis" 5 4]
ghci> ordenadas [Notas "Juan" 5 4, Notas "Eva" 5 4]
[Notas "Eva" 5 4, Notas "Juan" 5 4]
```

---

## Soluciones

```
import Data.List (sort)
```

```
data Notas = Notas String Int Int
  deriving (Read, Show, Eq)
```

```
ordenadas :: [Notas] -> [Notas]
```

```
ordenadas ns =
```

```
  [Notas n x y | (y,x,n) <- sort [(y1,x1,n1) | (Notas n1 x1 y1) <- ns]]
```

# Ejercicio 19

## Emparejamiento binario

### Enunciado

Definir la función

---

```
zipBinario :: [a -> b -> c] -> [a] -> [b] -> [c]
```

---

tal que (zipBinario fs xs ys) es la lista obtenida aplicando cada una de las operaciones binarias de fs a los correspondientes elementos de xs e ys. Por ejemplo,

---

```
zipBinario [(+), (*), (*)] [2,2,2] [4,4,4]    == [6,8,8]
zipBinario [(+)] [2,2,2] [4,4,4]              == [6]
zipBinario (cycle [(+), (*)]) [1..4] [2..5]   == [3,6,7,20]
```

---

### Soluciones

-- 1ª definición

```
zipBinario :: [a -> b -> c] -> [a] -> [b] -> [c]
zipBinario (f:fs) (x:xs) (y:ys) = f x y : zipBinario fs xs ys
zipBinario _ _ _                = []
```

-- 2ª definición

```
zipBinario2 :: [a -> b -> c] -> [a] -> [b] -> [c]
zipBinario2 fs xs ys = [f x y | (f,(x,y)) <- zip fs (zip xs ys)]
```

```
-- 3ª definición
zipBinario3 :: [a -> b -> c] -> [a] -> [b] -> [c]
zipBinario3 fs xs ys = [f x y | (f,x,y) <- zip3 fs xs ys]

-- 4ª definición
zipBinario4 :: [a -> b -> c] -> [a] -> [b] -> [c]
zipBinario4 = zipWith3 id
```



## Ejercicio 20

# Ampliación de columnas de una matriz

### Enunciado

Las matrices enteras se pueden representar mediante tablas con índices enteros:

---

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

---

Definir la función

---

```
ampliaColumnas :: Matriz -> Matriz -> Matriz
```

---

tal que `(ampliaColumnas p q)` es la matriz construida añadiendo las columnas de la matriz `q` a continuación de las de `p` (se supone que tienen el mismo número de filas). Por ejemplo, si `p` y `q` representa las dos primeras matrices, entonces `(ampliaColumnas p q)` es la tercera

---

0 1	4 5 6	0 1 4 5 6
2 3	7 8 9	2 3 7 8 9

---

En Haskell,

---

```
ghci> :{  
*Main| ampliaColumnas (listArray ((1,1),(2,2)) [0..3])  
*Main|                (listArray ((1,1),(2,3)) [4..9])
```

```
*Main| :}  
array ((1,1),(2,5))  
  [((1,1),0),((1,2),1),((1,3),4),((1,4),5),((1,5),6),  
    ((2,1),2),((2,2),3),((2,3),7),((2,4),8),((2,5),9)]
```

---

## Soluciones

```
import Data.Array
```

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

```
ampliaColumnas :: Matriz -> Matriz -> Matriz
```

```
ampliaColumnas p1 p2 =
```

```
  array ((1,1),(m,n1+n2)) [((i,j), f i j) | i <- [1..m], j <- [1..n1+n2]]
```

```
  where ((_,_), (m,n1)) = bounds p1
```

```
        ((_,_), (n2)) = bounds p2
```

```
        f i j | j <= n1    = p1!(i,j)
```

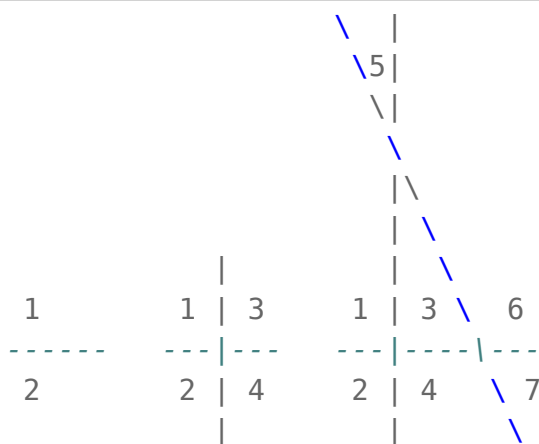
```
              | otherwise = p2!(i,j-n1)
```

# Ejercicio 21

## Regiones en el plano

### Enunciado

En los siguientes dibujos se observa que el número máximo de regiones en el plano generadas con 1, 2 ó 3 líneas son 2, 4 ó 7, respectivamente.



Definir la función

```
regiones :: Integer -> Integer
```

tal que (regiones n) es el número máximo de regiones en el plano generadas con n líneas. Por ejemplo,

```
regiones 3 == 7
regiones 100 == 5051
```

## Soluciones

```
import Test.QuickCheck

-- 1ª definición
regiones :: Integer -> Integer
regiones 0 = 1
regiones n = regiones (n-1) + n

-- 2ª definición
regiones2 :: Integer -> Integer
regiones2 n = n*(n+1) `div` 2 + 1

-- Equivalencia
-- =====

-- La propiedad es
prop_regiones :: Positive Integer -> Bool
prop_regiones (Positive n) =
    regiones n == regiones2 n

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

# Ejercicio 22

## Elemento más repetido

### Enunciado

Definir la función

---

```
masRepetido :: Ord a => [a] -> (a,Int)
```

---

tal que (masRepetido xs) es el elemento de xs que aparece más veces de manera consecutiva en la lista junto con el número de sus apariciones consecutivas; en caso de empate, se devuelve el último de dichos elementos. Por ejemplo,

---

```
masRepetido [1,1,4,4,1] == (4,2)
masRepetido "aadda"    == ('d',2)
```

---

### Soluciones

```
import Data.List
import Control.Arrow ((&&))           -- Para la definición sin argumentos
import Data.Function (on)
import Data.Tuple       -- Para 6'

-- 1ª definición
masRepetido1 :: Ord a => [a] -> (a,Int)
masRepetido1 [x] = (x,1)
```

```

masRepetido1 (x:y:zs) | m > n      = (x,m)
                  | otherwise = (u,n)
  where (u,n) = masRepetido1 (y:zs)
        m     = length (takeWhile (==x) (x:y:zs))

-- 2ª definición
masRepetido2 :: Ord a => [a] -> (a,Int)
masRepetido2 xs = (n,z)
  where (z,n) = maximum [(length ys,y) | (y:ys) <- group xs]

-- 3ª definición
masRepetido3 :: Ord a => [a] -> (a,Int)
masRepetido3 xs =
  swap (maximum [(length ys,y) | (y:ys) <- group xs])

-- 4ª definición
masRepetido4 :: Ord a => [a] -> (a,Int)
masRepetido4 xs =
  maximumBy compara [(y,length ys) | (y:ys) <- group xs]
  where compara (u,n) (v,m) = compare n m

-- 5ª definición
masRepetido5 :: Ord a => [a] -> (a,Int)
masRepetido5 =
  maximumBy (compare `on` snd) . map (head &&& length) . group

```

## Ejercicio 23

# Número de pares de elementos adyacentes iguales

### Enunciado

Definir la función

---

```
numeroParesAdyacentesIguales :: Eq a => [[a]] -> Int
```

---

tal que (numeroParesAdyacentesIguales xss) es el número de pares de elementos consecutivos (en la misma fila o columna) iguales de la matriz xss. Por ejemplo,

---

numeroParesAdyacentesIguales	[[0,1],[0,2]]	==	1
numeroParesAdyacentesIguales	[[0,0],[1,2]]	==	1
numeroParesAdyacentesIguales	[[0,1],[0,0]]	==	2
numeroParesAdyacentesIguales	[[1,2],[1,4],[4,4]]	==	3
numeroParesAdyacentesIguales	["ab","aa"]	==	2
numeroParesAdyacentesIguales	[[0,0,0],[0,0,0],[0,0,0]]	==	12
numeroParesAdyacentesIguales	[[0,0,0],[0,1,0],[0,0,0]]	==	8

---

### Soluciones

```
import Data.List (group,transpose)
import Data.Array
```

```

-- 1ª solución
-- =====

numeroParesAdyacentesIguales1 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIguales1 xss =
    numeroParesAdyacentesIgualesFilas xss +
    numeroParesAdyacentesIgualesFilas (transpose xss)

-- (numeroParesAdyacentesIgualesFilas xss) es el número de pares de
-- elementos consecutivos (en la misma fila) iguales de la matriz
-- xss. Por ejemplo,
--     ghci> numeroParesAdyacentesIgualesFilas [[0,0,1,0],[0,1,1,0],[0,1,0,1]]
--     2
--     ghci> numeroParesAdyacentesIgualesFilas ["0010","0110","0101"]
--     2
numeroParesAdyacentesIgualesFilas :: Eq a => [[a]] -> Int
numeroParesAdyacentesIgualesFilas xss =
    sum [numeroParesAdyacentesIgualesFila xs | xs <- xss]

-- La función anterior se puede definir con map
numeroParesAdyacentesIgualesFilas2 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIgualesFilas2 xss =
    sum (map numeroParesAdyacentesIgualesFila xss)

-- y también se puede definir sin argumentos:
numeroParesAdyacentesIgualesFilas3 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIgualesFilas3 =
    sum . map numeroParesAdyacentesIgualesFila

-- (numeroParesAdyacentesIgualesFila xs) es el número de pares de
-- elementos consecutivos de la lista xs. Por ejemplo,
numeroParesAdyacentesIgualesFila :: Eq a => [a] -> Int
numeroParesAdyacentesIgualesFila xs =
    length [(x,y) | (x,y) <- zip xs (tail xs), x == y]

-- 2ª solución
-- =====

-- numeroParesAdyacentesIguales2 :: Eq a => [[a]] -> Int

```



```
numeroParesAdyacentesIguales2 xss =
    length (concatMap tail (concatMap group (xss ++ transpose xss)))

-- 3ª solución
-- =====

numeroParesAdyacentesIguales3 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIguales3 xss =
    length [(i,j) | i <- [1..n-1], j <- [1..m], p!(i,j) == p!(i+1,j)] +
    length [(i,j) | i <- [1..n], j <- [1..m-1], p!(i,j) == p!(i,j+1)]
    where m = length xss
          n = length (head xss)
          p = listArray ((1,1),(m,n)) (concat xss)

-- 4ª solución
-- =====

numeroParesAdyacentesIguales4 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIguales4 =
    length . (tail ==<) . (group ==<) . ((++) ==< transpose)
```



## Ejercicio 24

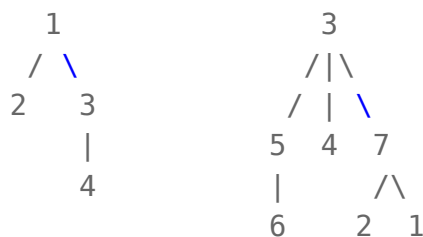
# Mayor producto de las ramas de un árbol

### Enunciado

Los árboles se pueden representar mediante el siguiente tipo de datos

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

Por ejemplo, los árboles



se representan por

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

Definir la función

```
mayorProducto :: (Ord a, Num a) => Arbol a -> a
```

tal que (`mayorProducto a`) es el mayor producto de las ramas del árbol `a`. Por ejemplo,

---

```
ghci> mayorProducto (N 1 [N 2 [], N 3 []])
3
ghci> mayorProducto (N 1 [N 8 [], N 4 [N 3 []]])
12
ghci> mayorProducto (N 1 [N 2 [], N 3 [N 4 []]])
12
ghci> mayorProducto (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
90
```

---

## Soluciones

```
import Test.QuickCheck
```

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

```
-- 1ª solución
-- =====
```

```
mayorProducto :: (Ord a, Num a) => Arbol a -> a
mayorProducto a = maximum (productosRamas a)
```

```
-- (productosRamas a) es la lista de los productos de las ramas
-- del árbol a. Por ejemplo,
-- ghci> productosRamas (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
-- [90,12,42,21]
```

```
productosRamas :: (Ord a, Num a) => Arbol a -> [a]
productosRamas (N x []) = [x]
productosRamas (N x xs) = [x * y | a <- xs, y <- productosRamas a]
```

```
-- 2ª solución
-- =====
```

```
mayorProducto2 :: (Ord a, Num a) => Arbol a -> a
mayorProducto2 (N x []) = x
mayorProducto2 (N x xs) =
```

```

| x > 0      = x * maximum (map mayorProducto2 xs)
| x == 0     = 0
| otherwise  = x * minimum (map menorProducto xs)

-- (menorProducto a) es el menor producto de las ramas del árbol
-- a. Por ejemplo,
--     ghci> menorProducto (N 1 [N 2 [], N 3 []])
--     2
--     ghci> menorProducto (N 1 [N 8 [], N 4 [N 3 []]])
--     8
--     ghci> menorProducto (N 1 [N 2 [], N 3 [N 4 []]])
--     2
--     ghci> menorProducto (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
--     12
menorProducto :: (Ord a, Num a) => Arbol a -> a
menorProducto (N x []) = x
menorProducto (N x xs)
  | x > 0      = x * minimum (map menorProducto xs)
  | x == 0     = 0
  | otherwise  = x * maximum (map mayorProducto2 xs)

-- 3ª solución
-- =====

mayorProducto3 :: (Ord a, Num a) => Arbol a -> a
mayorProducto3 a = maximum [product xs | xs <- ramas a]

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

-- 4ª solución
-- =====

mayorProducto4 :: (Ord a, Num a) => Arbol a -> a
mayorProducto4 a = maximum (map product (ramas a))

```

```

-- 5ª solución
-- =====

mayorProducto5 :: (Ord a, Num a) => Arbol a -> a
mayorProducto5 = maximum . map product . ramas

-- Equivalencia
-- =====

-- Generador de árboles.
--     ghci> sample ((gen_Arbol 5) :: Gen (Arbol Int))
--     N 0 [N 0 []]
--     N (-2) []
--     N 4 []
--     N 2 [N 4 []]
--     N 8 []
--     N (-2) [N (-9) [],N 7 []]
--     N 11 []
--     N (-11) [N 4 [],N 14 []]
--     N 10 [N (-3) [],N 13 []]
--     N 12 [N 11 []]
--     N 20 [N (-18) [],N (-13) []]
gen_Arbol :: Arbitrary a => Int -> Gen (Arbol a)
gen_Arbol m = do
  x <- arbitrary
  n <- choose (0, m `div` 2)
  xs <- vectorOf n (gen_Arbol (m `div` 4))
  return (N x xs)

-- Incluye los árboles en Arbitrary.
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized gen_Arbol

-- La propiedad es
prop_mayorProducto :: Arbol Int -> Bool
prop_mayorProducto a =
  all (== mayorProducto a)
    [f a | f <- [ mayorProducto2
                  , mayorProducto3
                  , mayorProducto4

```

```
, mayorProducto5  
]]
```

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





# Ejercicio 25

## Biparticiones de una lista

### Enunciado

Definir la función

---

```
biparticiones :: [a] -> [[a],[a]]
```

---

tal que (biparticiones xs) es la lista de pares formados por un prefijo de xs y el resto de xs. Por ejemplo,

---

```
ghci> biparticiones [3,2,5]
[([],[3,2,5]),([3],[2,5]),([3,2],[5]),([3,2,5],[])]
ghci> biparticiones "Roma"
[("", "Roma"), ("R", "oma"), ("Ro", "ma"), ("Rom", "a"), ("Roma", "")]
```

---

### Soluciones

```
import Data.List (inits, tails)
import Control.Applicative (liftA2)

-- 1ª solución
biparticiones1 :: [a] -> [[a],[a]]
biparticiones1 xs = [splitAt i xs | i <- [0..length xs]]

-- 2ª solución
```

```
biparticiones2 :: [a] -> [[a],[a]]
biparticiones2 xs = zip (inits xs) (tails xs)

-- 3ª solución
biparticiones3 :: [a] -> [[a],[a]]
biparticiones3 [] = ([],[])
biparticiones3 (x:xs) =
  ([],(x:xs)) : [(x:ys,zs) | (ys,zs) <- biparticiones3 xs]

-- 4ª solución
biparticiones4 :: [a] -> [[a],[a]]
biparticiones4 = liftA2 zip inits tails

-- 5ª solución (sin argumentos):
biparticiones5 :: [a] -> [[a],[a]]
biparticiones5 = zip <$> inits <*> tails
```

# Ejercicio 26

## Trenzado de listas

### Enunciado

Definir la función

---

```
trenza :: [a] -> [a] -> [a]
```

---

tal que (trenza xs ys) es la lista obtenida intercalando los elementos de xs e ys. Por ejemplo,

---

trenza [5,1] [2,7,4]	==	[5,2,1,7]
trenza [5,1,7] [2..]	==	[5,2,1,3,7,4]
trenza [2..] [5,1,7]	==	[2,5,3,1,4,7]
take 8 (trenza [2,4..] [1,5..])	==	[2,1,4,5,6,9,8,13]

---

### Soluciones

-- 1ª solución

```
trenza1 :: [a] -> [a] -> [a]
trenza1 xs ys = concat [[x,y] | (x,y) <- zip xs ys]
```

-- 2ª solución

```
trenza2 :: [a] -> [a] -> [a]
trenza2 xs ys = concat (zipWith par xs ys)
  where par x y = [x,y]
```

-- 3ª solución

```
trenza3 :: [a] -> [a] -> [a]
trenza3 = (concat .) . zipWith par
  where par x y = [x,y]
```

-- 4ª solución

```
trenza4 :: [a] -> [a] -> [a]
trenza4 (x:xs) (y:ys) = x : y : trenza4 xs ys
trenza4 _ _ = []
```

## Ejercicio 27

# Números triangulares con n cifras distintas

### Enunciado

Los números triangulares se forman como sigue

---

*	*	*
	* *	* *
		* * *
1	3	6

---

La sucesión de los números triangulares se obtiene sumando los números naturales. Así, los 5 primeros números triangulares son

---

1	=	1
3	=	1+2
6	=	1+2+3
10	=	1+2+3+4
15	=	1+2+3+4+5

---

Definir la función

---

```
triangularesConCifras :: Int -> [Integer]
```

---

tal que (triangulares n) es la lista de los números triangulares con n cifras distintas. Por ejemplo,

---

```

take 6 (triangularesConCifras 1) == [1,3,6,55,66,666]
take 6 (triangularesConCifras 2) == [10,15,21,28,36,45]
take 6 (triangularesConCifras 3) == [105,120,136,153,190,210]
take 5 (triangularesConCifras 4) == [1035,1275,1326,1378,1485]
take 2 (triangularesConCifras 10) == [1062489753,1239845706]

```

---

## Soluciones

```

import Data.List (nub)

triangularesConCifras1 :: Int -> [Integer]
triangularesConCifras1 n =
    [x | x <- triangulares, nCifras x == n]

-- 1ª definición de triangulares
-- =====
triangulares1 :: [Integer]
triangulares1 = 1 : [x+y | (x,y) <- zip [2..] triangulares1]

-- 2ª definición de triangulares
-- =====
triangulares2 :: [Integer]
triangulares2 = scanl (+) 1 [2..]

-- 3ª definición de triangulares
-- =====
triangulares3 :: [Integer]
triangulares3 = [(n*(n+1)) `div` 2 | n <- [1..]]

-- Comparación de eficiencia
-- =====

--      λ> triangulares1 !! (10^6)
--      500001500001
--      (2.03 secs, 330,448,496 bytes)
--      λ> triangulares2 !! (10^6)
--      500001500001

```

```
-- (1.09 secs, 225,631,536 bytes)
-- λ> triangulares3 !! (10^6)
-- 500001500001
-- (0.92 secs, 184,405,920 bytes)

-- Usaremos como triangulares la 2ª definición:
triangulares :: [Integer]
triangulares = triangulares3

-- (nCifras x) es el número de cifras distintas del número x. Por
-- ejemplo,
--   nCifras 325275 == 4
nCifras :: Integer -> Int
nCifras = length . nub . show
```





## Ejercicio 28

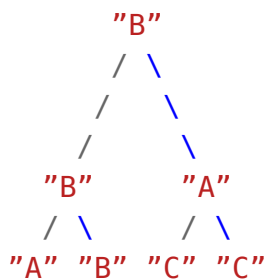
# Enumeración de árboles binarios

### Enunciado

Los árboles binarios se pueden representar mediante el tipo `Arbol` definido por

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

Por ejemplo, el árbol



se puede definir por

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

Definir la función

```
enumeraArbol :: Arbol t -> Arbol Int
```

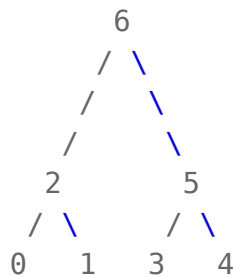
tal que `(enumeraArbol a)` es el árbol obtenido numerando las hojas y los nodos de `a` desde la hoja izquierda hasta la raíz. Por ejemplo,

---

```
ghci> enumeraArbol ej1
N 6 (N 2 (H 0) (H 1)) (N 5 (H 3) (H 4))
```

---

Gráficamente,



## Soluciones

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

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

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

## Ejercicio 29

# Elementos con algún vecino menor

### Enunciado

Las matrices puede representarse mediante tablas cuyos índices son pares de números naturales:

---

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

---

Definir la función

---

```
algunMenor :: Matriz -> [Int]
```

---

tal que (algunMenor p) es la lista de los elementos de p que tienen algún vecino menor que él. Por ejemplo,

---

```
ghci> algunMenor (listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,4,2,5,4])  
[9,4,6,5,8,7,4,2,5,4]
```

---

pues sólo el 1 y el 3 no tienen ningún vecino menor en la matriz

---

```
| 9 4 6 5 |  
| 8 1 7 3 |  
| 4 2 5 4 |
```

---

## Soluciones

```
import Data.Array

type Matriz = Array (Int,Int) Int

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