

Exercitium (curso 2018–19)  
Ejercicios de programación funcional con Haskell  
(hasta el 28 de diciembre de 2018)

José A. Alonso Jiménez

---

Grupo de Lógica Computacional  
Dpto. de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla  
Sevilla, 5 de enero de 2019

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.

“Sorpresas tiene la vida,  
Guiomar, del alma y del cuerpo;  
que nadie guarde hasta el fin  
el nombre que le pusieron;  
nadie crea ser quien dicen  
que es, ni que pueda serlo.”

De Antonio Machado

*Para Guiomar*



# Índice general

1	Listas equidigitales	11
2	Distancia de Hamming	13
3	Último dígito no nulo del factorial	17
4	Diferencia simétrica	21
5	Números libres de cuadrados	23
6	Capicúas productos de dos números de dos dígitos	27
7	Números autodescriptivos	29
8	Número de parejas	31
9	Reconocimiento de particiones	35
10	Relación definida por una partición	39
11	Ceros finales del factorial	41
12	Números primos sumas de dos primos	45
13	Suma de inversos de potencias de cuatro	49
14	Elemento solitario	53
15	Números colinas	59
16	Raíz cúbica entera	63
17	Numeración de los árboles binarios completos	67

18 Posiciones en árboles binarios	69
19 Posiciones en árboles binarios completos	73
20 Elemento del árbol binario completo según su posición	79
21 Aproximación entre $\pi$ y $e$	83
22 Menor contenedor de primos	85
23 Árbol de computación de Fibonacci	87
24 Entre dos conjuntos	93
25 Expresiones aritméticas generales	99
26 Superación de límites	101
27 Intercambio de la primera y última columna de una matriz	103
28 Números primos de Pierpont	105
29 Grado exponencial	107
30 Divisores propios maximales	111
31 Árbol de divisores	115
32 Divisores compuestos	119
33 Número de divisores compuestos	125
34 Tablas de operaciones binarias	129
35 Reconocimiento de conmutatividad	133
36 Árbol de subconjuntos	139
37 El teorema de Navidad de Fermat	143
38 El 2019 es apocalíptico	149

# 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 2018-19.

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>.

En el libro se irán añadiendo semanalmente las soluciones de los ejercicios del curso.

---

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

<sup>2</sup><https://www.cs.us.es/~jalonso/cursos/ilm-18>

<sup>3</sup><https://www.cs.us.es/~jalonso/cursos/ilm-18/ejercicios/ejercicios-I1M-2018.pdf>

<sup>4</sup><https://www.glc.us.es/~jalonso/vestigium/category/curso/ilm/ilm2018>

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

## Cuaderno de bitácora

En esta sección se registran los cambios realizados en las sucesivas versiones del libro.

### Versión del 16 de diciembre de 2018

Se han añadido los ejercicios resueltos de la primera semana de diciembre:

- Numeración de los árboles binarios completos
- Posiciones en árboles binarios
- Posiciones en árboles binarios completos
- Elemento del árbol binario completo según su posición
- Aproximación entre  $\pi$  y  $e$

### Versión del 22 de diciembre de 2018

Se han añadido los ejercicios resueltos de la primera semana de diciembre:

- Menor contenedor de primos
- Árbol de computación de Fibonacci
- Entre dos conjuntos
- Expresiones aritméticas generales
- Superación de límites

---

<sup>5</sup><https://github.com/jaalonso/Exercitium2018>



## **Versión del 29 de diciembre de 2018**

Se han añadido los ejercicios resueltos de la primera semana de diciembre:

- Intercambio de la primera y última columna de una matriz
- Números primos de Pierpont
- Grado exponencial
- Divisores propios maximales
- Árbol de divisores

## **Versión del 29 de diciembre de 2018**

Se han añadido los ejercicios resueltos del 24 al 28 de diciembre:

- Divisores compuestos
- Número de divisores compuestos
- Tablas de operaciones binarias
- Reconocimiento de conmutatividad
- Árbol de subconjuntos



# Ejercicio 1

## Listas equidigitales

### Enunciado

Una lista de números naturales es equidigital si todos sus elementos tienen el mismo número de dígitos.

Definir la función

---

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

---

tal que (equidigital xs) se verifica si xs es una lista equidigital. Por ejemplo,

---

```
equidigital [343,225,777,943] == True
equidigital [343,225,777,94,3] == False
```

---

### Soluciones

```
-- 1ª definición
-- =====
```

```
equidigital :: [Int] -> Bool
equidigital xs = todosIguales (numerosDeDigitos xs)
```

```
-- (numerosDeDigitos xs) es la lista de los números de dígitos de
```

```

-- los elementos de xs. Por ejemplo,
--   numerosDeDigitos [343,225,777,943] == [3,3,3,3]
--   numerosDeDigitos [343,225,777,94,3] == [3,3,3,2,1]
numerosDeDigitos :: [Int] -> [Int]
numerosDeDigitos xs = [numeroDeDigitos x | x <- xs]

-- (numeroDeDigitos x) es el número de dígitos de x. Por ejemplo,
--   numeroDeDigitos 475 == 3
numeroDeDigitos :: Int -> Int
numeroDeDigitos x = length (show x)

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--   todosIguales [3,3,3,3] == True
--   todosIguales [3,3,3,2,1] == False
todosIguales :: Eq a => [a] -> Bool
todosIguales (x:y:zs) = x == y && todosIguales (y:zs)
todosIguales _        = True

-- 2ª definición
-- =====

equidigital2 :: [Int] -> Bool
equidigital2 []      = True
equidigital2 (x:xs) = and [numeroDeDigitos y == n | y <- xs]
                    where n = numeroDeDigitos x

-- 3ª definición
-- =====

equidigital3 :: [Int] -> Bool
equidigital3 (x:y:zs) = numeroDeDigitos x == numeroDeDigitos y &&
                        equidigital3 (y:zs)
equidigital3 _        = True

```

## Ejercicio 2

# Distancia de Hamming

### Enunciado

La distancia de Hamming entre dos listas es el número de posiciones en que los correspondientes elementos son distintos. Por ejemplo, la distancia de Hamming entre `romaz` y `loba` es 2 (porque hay 2 posiciones en las que los elementos correspondientes son distintos: la 1ª y la 3ª).

Definir la función

---

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

---

tal que `(distancia xs ys)` es la distancia de Hamming entre `xs` e `ys`. Por ejemplo,

---

```
distancia "romano" "comino" == 2
distancia "romano" "camino" == 3
distancia "roma"   "comino" == 2
distancia "roma"   "camino" == 3
distancia "romano" "ron"     == 1
distancia "romano" "cama"    == 2
distancia "romano" "rama"    == 1
```

---

Comprobar con QuickCheck si la distancia de Hamming tiene la siguiente propiedad:  $\text{distancia}(xs,ys) = 0$  si, y sólo si,  $xs = ys$  y, en el caso de que no se verifique, modificar ligeramente la propiedad para obtener una condición necesaria y suficiente de  $\text{distancia}(xs,ys) = 0$ .

## Soluciones

```
import Test.QuickCheck
```

```
-- 1ª definición:
```

```
distancia :: Eq a => [a] -> [a] -> Int
distancia xs ys = length [(x,y) | (x,y) <- zip xs ys, x /= y]
```

```
-- 2ª definición:
```

```
distancia2 :: Eq a => [a] -> [a] -> Int
distancia2 [] _ = 0
distancia2 _ [] = 0
distancia2 (x:xs) (y:ys) | x /= y = 1 + distancia2 xs ys
                        | otherwise = distancia2 xs ys
```

```
-- La propiedad es
```

```
prop_distancial :: [Int] -> [Int] -> Bool
prop_distancial xs ys =
  (distancia xs ys == 0) == (xs == ys)
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_distancial
-- *** Failed! Falsifiable (after 2 tests and 1 shrink):
-- []
-- [1]
--
```

```
-- En efecto,
```

```
-- ghci> distancia [] [1] == 0
-- True
-- ghci> [] == [1]
-- False
--
```

```
-- La primera modificación es restringir la propiedad a lista de igual
-- longitud:
```

```
prop_distancia2 :: [Int] -> [Int] -> Property
prop_distancia2 xs ys =
  length xs == length ys ==>
  (distancia xs ys == 0) == (xs == ys)
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_distancia2
-- *** Gave up! Passed only 33 tests.

-- Nota. La propiedad se verifica, pero al ser la condición demasiado
-- restringida sólo 33 de los casos la cumple.

-- La segunda restricción es limitar las listas a la longitud de la más
-- corta:
prop_distancia3 :: [Int] -> [Int] -> Bool
prop_distancia3 xs ys =
  (distancia xs ys == 0) == (take n xs == take n ys)
  where n = min (length xs) (length ys)

-- La comprobación es
-- ghci> quickCheck prop_distancia3
-- +++ OK, passed 100 tests.
```





## Ejercicio 3

# Último dígito no nulo del factorial

### Enunciado

El factorial de 7 es  $7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040$ . Por tanto, el último dígito no nulo del factorial de 7 es 4.

Definir la función

---

```
ultimoNoNuloFactorial :: Integer -> Integer
```

---

tal que (ultimoNoNuloFactorial n) es el último dígito no nulo del factorial de n. Por ejemplo,

---

```
ultimoNoNuloFactorial 7  == 4
ultimoNoNuloFactorial 10 == 8
ultimoNoNuloFactorial 12 == 6
ultimoNoNuloFactorial 97 == 2
ultimoNoNuloFactorial 0  == 1
```

---

Comprobar con QuickCheck que si n es mayor que 4, entonces el último dígito no nulo del factorial de n es par.

## Solución

```

import Test.QuickCheck

-- 1ª definición
-- =====

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

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

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

-- 2ª definición
-- =====

ultimoNoNuloFactorial2 :: Integer -> Integer
ultimoNoNuloFactorial2 n = ultimoNoNulo2 (factorial n)

-- (ultimoNoNulo2 n) es el último dígito no nulo de n. Por ejemplo,
--   ultimoNoNulo2 5040 == 4
ultimoNoNulo2 :: Integer -> Integer
ultimoNoNulo2 n = read [head (dropWhile (=='0') (reverse (show n)))]

-- Comprobación
-- =====

-- La propiedad es
prop_ultimoNoNuloFactorial :: Integer -> Property
prop_ultimoNoNuloFactorial n =

```

---

```
n > 4 ==> even (ultimoNoNuloFactorial n)

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



## Ejercicio 4

# Diferencia simétrica

### Enunciado

La **diferencia simétrica** de dos conjuntos es el conjunto cuyos elementos son aquellos que pertenecen a alguno de los conjuntos iniciales, sin pertenecer a ambos a la vez. Por ejemplo, la diferencia simétrica de 2,5,3 y 4,2,3,7 es 5,4,7.

Definir la función

---

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

---

tal que (diferenciaSimetrica xs ys) es la diferencia simétrica de xs e ys. Por ejemplo,

---

```
diferenciaSimetrica [2,5,3] [4,2,3,7] == [4,5,7]
diferenciaSimetrica [2,5,3] [5,2,3]   == []
diferenciaSimetrica [2,5,2] [4,2,3,7] == [3,4,5,7]
diferenciaSimetrica [2,5,2] [4,2,4,7] == [4,5,7]
diferenciaSimetrica [2,5,2,4] [4,2,4,7] == [5,7]
```

---

### Soluciones

```
import Data.List
```

```
-- 1ª definición
```

```
diferenciaSimetrica :: Ord a => [a] -> [a] -> [a]
diferenciaSimetrica xs ys =
  sort (nub ([x | x <- xs, x `notElem` ys] ++ [y | y <- ys, y `notElem` xs]))

-- 2ª definición
diferenciaSimetrica2 :: Ord a => [a] -> [a] -> [a]
diferenciaSimetrica2 xs ys =
  sort (nub (union xs ys \\ intersect xs ys))

-- 3ª definición
diferenciaSimetrica3 :: Ord a => [a] -> [a] -> [a]
diferenciaSimetrica3 xs ys =
  [x | x <- sort (nub (xs ++ ys))
    , x `notElem` xs || x `notElem` ys]
```

## Ejercicio 5

# Números libres de cuadrados

### Enunciado

Un número entero positivo es libre de cuadrados si no es divisible por el cuadrado de ningún entero mayor que 1. Por ejemplo, 70 es libre de cuadrado porque sólo es divisible por 1, 2, 5, 7 y 70; en cambio, 40 no es libre de cuadrados porque es divisible por  $2^2$ .

Definir la función

---

```
libreDeCuadrados :: Integer -> Bool
```

---

tal que (libreDeCuadrados x) se verifica si x es libre de cuadrados. Por ejemplo,

---

libreDeCuadrados 70	==	True
libreDeCuadrados 40	==	False
libreDeCuadrados 510510	==	True
libreDeCuadrados (((10 <sup>10</sup> ) <sup>10</sup> ) <sup>10</sup> )	==	False

---

### Soluciones

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

```
-- 1ª definición
```

```

-- =====

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

-- (divisoresPrimos x) es la lista de los divisores primos de x. Por
-- ejemplo,
--     divisoresPrimos 40 == [2,5]
--     divisoresPrimos 70 == [2,5,7]
divisoresPrimos :: Integer -> [Integer]
divisoresPrimos x = [n | n <- divisores x, primo n]

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

-- (primo n) se verifica si n es primo. Por ejemplo,
--     primo 30 == False
--     primo 31 == True
primo :: Integer -> Bool
primo n = divisores n == [1, n]

-- 2ª definición
-- =====

libreDeCuadrados2 :: Integer -> Bool
libreDeCuadrados2 n =
    null [x | x <- [2..n], rem n (x^2) == 0]

-- 3ª definición
-- =====

libreDeCuadrados3 :: Integer -> Bool
libreDeCuadrados3 n =
    null [x | x <- [2..floor (sqrt (fromIntegral n))],
            rem n (x^2) == 0]

-- 4ª definición
-- =====

```



```
libreDeCuadrados4 :: Integer -> Bool
libreDeCuadrados4 x =
    factorizacion x == nub (factorizacion x)

-- (factorizacion n) es la lista de factores primos de n. Por ejemplo,
--   factorizacion 180 == [2,2,3,3,5]
factorizacion :: Integer -> [Integer]
factorizacion n | n == 1    = []
                | otherwise = x : factorizacion (div n x)
    where x = menorFactor n

-- (menorFactor n) es el menor divisor de n. Por ejemplo,
--   menorFactor 15 == 3
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- Comparación de eficiencia
-- =====

--   λ> libreDeCuadrados 510510
--   True
--   (0.76 secs, 89,522,360 bytes)
--   λ> libreDeCuadrados2 510510
--   True
--   (1.78 secs, 371,826,320 bytes)
--   λ> libreDeCuadrados3 510510
--   True
--   (0.01 secs, 0 bytes)
--   λ> libreDeCuadrados4 510510
--   True
--   (0.00 secs, 153,216 bytes)
```



## Ejercicio 6

# Capicúas productos de dos números de dos dígitos

### Enunciado

El número 9009 es capicúa y es producto de dos números de dos dígitos, pues  $9009 = 91 \times 99$ .

Definir la lista

---

```
capicuasP2N2D :: [Int]
```

---

cuyos elementos son los números capicúas que son producto de 2 números de dos dígitos. Por ejemplo,

---

```
take 5  capicuasP2N2D == [121,242,252,272,323]
length  capicuasP2N2D == 74
drop 70  capicuasP2N2D == [8008,8118,8448,9009]
```

---

### Soluciones

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

```
capicuasP2N2D :: [Int]
```

```
capicuasP2N2D = [x | x <- productos, esCapicua x]
```

```
-- productos es la lista de números que son productos de 2 números de
-- dos dígitos.
productos :: [Int]
productos = sort (nub [x*y | x <- [10..99], y <- [x..99]])

-- (esCapicua x) se verifica si x es capicúa.
esCapicua :: Int -> Bool
esCapicua x = xs == reverse xs
  where xs = show x
```

## Ejercicio 7

# Números autodescriptivos

### Enunciado

Un número  $n$  es autodescriptivo cuando para cada posición  $k$  de  $n$  (empezando a contar las posiciones a partir de 0), el dígito en la posición  $k$  es igual al número de veces que ocurre  $k$  en  $n$ . Por ejemplo, 1210 es autodescriptivo porque tiene 1 dígito igual a "0", 2 dígitos iguales a "1", 1 dígito igual a "2" y ningún dígito igual a "3".

Definir la función

---

```
autodescriptivo :: Integer -> Bool
```

---

tal que (autodescriptivo  $n$ ) se verifica si  $n$  es autodescriptivo. Por ejemplo,

---

```
λ> autodescriptivo 1210
True
λ> [x | x <- [1..100000], autodescriptivo x]
[1210,2020,21200]
λ> autodescriptivo 9210000001000
True
```

---

**Nota:** Se puede usar la función [genericLength](#).

## Soluciones

```
import Data.List (genericLength)

autodescriptivo :: Integer -> Bool
autodescriptivo n = autodescriptiva (digitos n)

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

autodescriptiva :: [Integer] -> Bool
autodescriptiva ns =
  and [x == ocurrencias k ns | (k,x) <- zip [0..] ns]

ocurrencias :: Integer -> [Integer] -> Integer
ocurrencias x ys = genericLength (filter (==x) ys)
```

## Ejercicio 8

# Número de parejas

### Enunciado

Definir la función

---

```
nParejas :: Ord a => [a] -> Int
```

---

tal que (nParejas xs) es el número de parejas de elementos iguales en xs. Por ejemplo,

---

```
nParejas [1,2,2,1,1,3,5,1,2] == 3
nParejas [1,2,1,2,1,3,2]    == 2
nParejas [1..2*10^6]        == 0
nParejas2 ([1..10^6] ++ [1..10^6]) == 1000000
```

---

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

Comprobar con QuickCheck que para toda lista de enteros xs, el número de parejas de xs es igual que el número de parejas de la inversa de xs.

### Soluciones

```
import Test.QuickCheck
import Data.List ((\\), group, sort)
```

```

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

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

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

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

-- Equivalencia
prop_equiv :: [Int] -> Bool
prop_equiv xs =
  nParejas xs == nParejas2 xs &&
  nParejas xs == nParejas3 xs &&
  nParejas xs == nParejas4 xs

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

-- Comparación de eficiencia
--   λ> nParejas [1..20000]
--   0
--   (2.54 secs, 4,442,808 bytes)
--   λ> nParejas2 [1..20000]
--   0
--   (0.03 secs, 12,942,232 bytes)
--   λ> nParejas3 [1..20000]
--   0

```



```
-- (0.02 secs, 13,099,904 bytes)
-- λ> nParejas4 [1..20000]
-- 0
-- (0.01 secs, 11,951,992 bytes)

-- Propiedad:
prop_nParejas :: [Int] -> Bool
prop_nParejas xs =
  nParejas xs == nParejas (reverse xs)

-- Comprobación
comprueba :: IO ()
comprueba = quickCheck prop_nParejas

-- Comprobación:
-- λ> comprueba
-- +++ OK, passed 100 tests.
```



## Ejercicio 9

# Reconocimiento de particiones

### Enunciado

Una **partición** de un conjunto es una división del mismo en subconjuntos disjuntos no vacíos.

Definir la función

---

```
esParticion :: Eq a => [[a]] -> Bool
```

---

tal que (esParticion xss) se verifica si xss es una partición; es decir sus elementos son listas no vacías disjuntas. Por ejemplo.

---

```
esParticion [[1,3],[2],[9,5,7]] == True
esParticion [[1,3],[2],[9,5,1]] == False
esParticion [[1,3],[],[9,5,7]]  == False
esParticion [[2,3,2],[4]]       == True
```

---

### Soluciones

```
import Data.List ((\\), intersect)

-- 1ª definición
-- =====
```

```

esParticion :: Eq a => [[a]] -> Bool
esParticion xss =
  [] `notElem` xss &&
  and [disjuntos xs ys | xs <- xss, ys <- xss \\ [xs]]

```

```

disjuntos :: Eq a => [a] -> [a] -> Bool
disjuntos xs ys = null (xs `intersect` ys)

```

```

-- 2ª definición
-- =====

```

```

esParticion2 :: Eq a => [[a]] -> Bool
esParticion2 [] = True
esParticion2 (xs:xss) =
  not (null xs) &&
  and [disjuntos xs ys | ys <- xss] &&
  esParticion2 xss

```

```

-- 3ª definición
-- =====

```

```

esParticion3 :: Eq a => [[a]] -> Bool
esParticion3 [] = True
esParticion3 (xs:xss) =
  not (null xs) &&
  all (disjuntos xs) xss &&
  esParticion3 xss

```

```

-- Equivalencia

```

```

prop_equiv :: [[Int]] -> Bool
prop_equiv xss =
  and [esParticion xss == f xss | f <- [ esParticion2
                                         , esParticion3]]

```

```

-- Comprobación

```

```

-- λ> quickCheck prop_equiv
-- +++ OK, passed 100 tests.

```

```

-- Comparación de eficiencia:

```

```

-- λ> esParticion [[x] | x <- [1..3000]]

```

```
-- True
-- (4.37 secs, 3,527,956,400 bytes)
-- λ> esParticion2 [[x] | x <- [1..3000]]
-- True
-- (1.26 secs, 1,045,792,552 bytes)
-- λ> esParticion3 [[x] | x <- [1..3000]]
-- True
-- (1.30 secs, 1,045,795,272 bytes)
-- λ> esParticion3 [[x] | x <- [1..3000]]
-- True
-- (1.30 secs, 1,045,795,272 bytes)
```



## Ejercicio 10

# Relación definida por una partición

### Enunciado

Dos elementos están **relacionados por una partición** xss si pertenecen al mismo elemento de xss.

Definir la función

---

```
relacionados :: Eq a => [[a]] -> a -> a -> Bool
```

---

tal que (relacionados xss y z) se verifica si los elementos y y z están relacionados por la partición xss. Por ejemplo,

---

```
relacionados [[1,3],[2],[9,5,7]] 7 9 == True
relacionados [[1,3],[2],[9,5,7]] 3 9 == False
relacionados [[1,3],[2],[9,5,7]] 4 9 == False
```

---

### Soluciones

```
-- 1ª definición
-- =====
```

```
relacionados :: Eq a => [[a]] -> a -> a -> Bool
```

```

relacionados [] _ _ = False
relacionados (xs:xss) y z
  | y `elem` xs = z `elem` xs
  | otherwise   = relacionados xss y z

-- 2ª definición
-- =====

relacionados2 :: Eq a => [[a]] -> a -> a -> Bool
relacionados2 xss y z =
  or [elem y xs && elem z xs | xs <- xss]

-- 3ª definición
-- =====

relacionados3 :: Eq a => [[a]] -> a -> a -> Bool
relacionados3 xss y z =
  or [[y,z] `subconjunto` xs | xs <- xss]

-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys; es
-- decir, si todos los elementos de xs pertenecen a ys. Por ejemplo,
--   subconjunto [3,2,3] [2,5,3,5] == True
--   subconjunto [3,2,3] [2,5,6,5] == False
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]

-- 4ª definición
-- =====

relacionados4 :: Eq a => [[a]] -> a -> a -> Bool
relacionados4 xss y z =
  any ([y,z] `subconjunto`) xss

```



# Ejercicio 11

## Ceros finales del factorial

### Enunciado

Definir la función

---

```
cerosDelFactorial :: Integer -> Integer
```

---

tal que (cerosDelFactorial n) es el número de ceros en que termina el factorial de n. Por ejemplo,

---

```
cerosDelFactorial 24      == 4
cerosDelFactorial 25      == 6
length (show (cerosDelFactorial (1234^5678))) == 17552
```

---

### Soluciones

```
import Data.List (genericLength)

-- 1ª definición
-- =====

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

-- (factorial n) es el factorial n. Por ejemplo,
```

```

--      factorial 3 == 6
factorial :: Integer -> Integer
factorial n = product [1..n]

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--      ceros 320000 == 4
ceros :: Integer -> Integer
ceros n | rem n 10 /= 0 = 0
        | otherwise    = 1 + ceros (div n 10)

-- 2ª definición
-- =====

cerosDelFactorial2 :: Integer -> Integer
cerosDelFactorial2 n = ceros2 (factorial n)

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--      ceros 320000 == 4
ceros2 :: Integer -> Integer
ceros2 n = genericLength (takeWhile (=='0') (reverse (show n)))

-- 3ª definición
-- =====

cerosDelFactorial3 :: Integer -> Integer
cerosDelFactorial3 n
  | n < 5      = 0
  | otherwise = m + cerosDelFactorial3 m
  where m = n `div` 5

-- Comparación de la eficiencia
--      λ> cerosDelFactorial1 (3*10^4)
--      7498
--      (3.96 secs, 1,252,876,376 bytes)
--      λ> cerosDelFactorial2 (3*10^4)
--      7498
--      (3.07 secs, 887,706,864 bytes)
--      λ> cerosDelFactorial3 (3*10^4)

```

```
-- 7498
-- (0.03 secs, 9,198,896 bytes)
```



## Ejercicio 12

# Números primos sumas de dos primos

### Enunciado

Definir las funciones

---

```
esPrimoSumaDeDosPrimos :: Integer -> Bool
primosSumaDeDosPrimos  :: [Integer]
```

---

tales que

- 
- (esPrimoSumaDeDosPrimos x) se verifica si x es un número primo que se puede escribir como la suma de dos números primos. Por ejemplo,

---

```
esPrimoSumaDeDosPrimos 19 == True
esPrimoSumaDeDosPrimos 20 == False
esPrimoSumaDeDosPrimos 23 == False
```

---

- primosSumaDeDosPrimos es la lista de los números primos que se pueden escribir como la suma de dos números primos. Por ejemplo,

---

```
λ> take 17 primosSumaDeDosPrimos
[5,7,13,19,31,43,61,73,103,109,139,151,181,193,199,229,241]
λ> primosSumaDeDosPrimos !! (10^5)
18409543
```

---

## Soluciones

```
import Data.Numbers.Primes (isPrime, primes)
import Test.QuickCheck

-- 1ª solución
-- =====

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

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

-- 2ª solución
-- =====

primosSumaDeDosPrimos2 :: [Integer]
primosSumaDeDosPrimos2 =
    [y | (x,y) <- zip primes (tail primes)
      , y == x + 2]

esPrimoSumaDeDosPrimos2 :: Integer -> Bool
esPrimoSumaDeDosPrimos2 x =
    x == head (dropWhile (<x) primosSumaDeDosPrimos2)

-- Equivalencias
-- =====

-- Equivalencia de esPrimoSumaDeDosPrimos
prop_esPrimoSumaDeDosPrimos_equiv :: Integer -> Property
prop_esPrimoSumaDeDosPrimos_equiv x =
    x > 0 ==>
    esPrimoSumaDeDosPrimos x == esPrimoSumaDeDosPrimos2 x

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

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

-- Equivalencia de primosSumaDeDosPrimos
prop_primosSumaDeDosPrimos_equiv :: Int -> Property
prop_primosSumaDeDosPrimos_equiv n =
  n >= 0 ==>
  primosSumaDeDosPrimos !! n == primosSumaDeDosPrimos2 !! n

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

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

--      λ> primosSumaDeDosPrimos !! (10^4)
--      1261081
--      (2.07 secs, 4,540,085,256 bytes)
--
-- Se recarga para evitar memorización
--      λ> primosSumaDeDosPrimos2 !! (10^4)
--      1261081
--      (0.49 secs, 910,718,408 bytes)
```

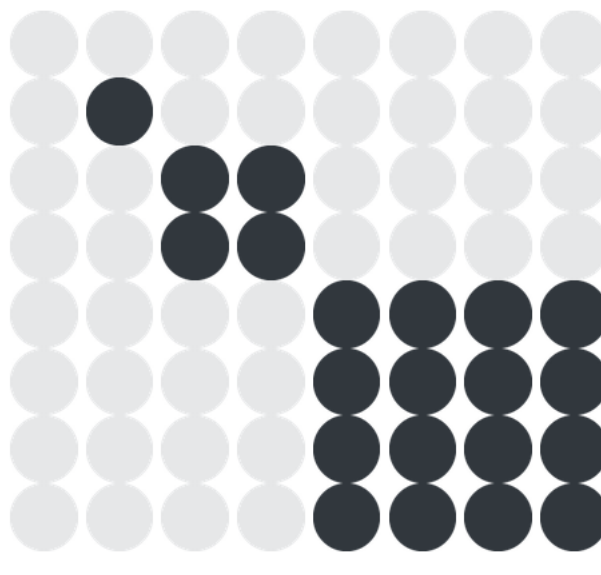




# Suma de inversos de potencias de cuatro

Esta semana se ha publicado en [Twitter](#) una demostración visual de que

como se muestra en la siguiente imagen



49

---

```
sumaInversosPotenciasDeCuatro :: [Double]
aproximacion :: Double -> Int
```

---

tales que

- sumaInversosPotenciasDeCuatro es la lista de la suma de la serie de los inversos de las potencias de cuatro. Por ejemplo,

---

```
λ> take 6 sumaInversosPotenciasDeCuatro
[0.25,0.3125,0.328125,0.33203125,0.3330078125,0.333251953125]
```

---

- (aproximacion e) es el menor número de términos de la serie anterior que hay que sumar para que el valor absoluto de su diferencia con  $1/3$  sea menor que e. Por ejemplo,

---

```
aproximacion 0.001 == 4
aproximacion 1e-3 == 4
aproximacion 1e-6 == 9
aproximacion 1e-20 == 26
sumaInversosPotenciasDeCuatro !! 26 == 0.3333333333333333
```

---

## Soluciones

-- 1ª definición

```
sumaInversosPotenciasDeCuatro :: [Double]
sumaInversosPotenciasDeCuatro =
  [sum [1 / (4^k) | k <- [1..n]] | n <- [1..]]
```

-- 2ª definición

```
sumaInversosPotenciasDeCuatro2 :: [Double]
sumaInversosPotenciasDeCuatro2 =
  [1/4*((1/4)^n-1)/(1/4-1) | n <- [1..]]
```

-- 3ª definición

```
sumaInversosPotenciasDeCuatro3 :: [Double]
sumaInversosPotenciasDeCuatro3 =
  [(1 - 0.25^n)/3 | n <- [1..]]
```

```
-- 1ª solución
aproximacion :: Double -> Int
aproximacion e =
    length (takeWhile (≥e) es)
    where es = [abs (1/3 - x) | x <- sumaInversosPotenciasDeCuatro2]

-- 2ª solución
aproximacion2 :: Double -> Int
aproximacion2 e =
    head [n | (x,n) <- zip es [0..]
           , x < e]
    where es = [abs (1/3 - x) | x <- sumaInversosPotenciasDeCuatro2]
```



# Ejercicio 14

## Elemento solitario

### Enunciado

Definir la función

---

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

---

tal que (solitario xs) es el único elemento que ocurre una vez en la lista xs (se supone que la lista xs tiene al menos 3 elementos y todos son iguales menos uno que es el solitario). Por ejemplo,

---

```
solitario [2,2,7,2] == 7
solitario [2,2,2,7] == 7
solitario [7,2,2,2] == 7
solitario (replicate (2*10^7) 1 ++ [2]) == 2
```

---

### Soluciones

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

-- 1ª definición
-- =====

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

```

solitario xs =
    head [x | x <- xs
           , cuenta xs x == 1]

cuenta :: Eq a => [a] -> a -> Int
cuenta xs x = length [y | y <- xs
                          , x == y]

-- 2ª definición
-- =====

solitario2 :: Ord a => [a] -> a
solitario2 xs = head (filter (\x -> cuenta2 xs x == 1) xs)

cuenta2 :: Eq a => [a] -> a -> Int
cuenta2 xs x = length (filter (==x) xs)

-- 3ª definición
-- =====

solitario3 :: Ord a => [a] -> a
solitario3 [x] = x
solitario3 (x1:x2:x3:xs)
    | x1 /= x2 && x2 == x3 = x1
    | x1 == x2 && x2 /= x3 = x3
    | otherwise           = solitario3 (x2:x3:xs)
solitario3 _ = error "Imposible"

-- 4ª definición
-- =====

solitario4 :: Ord a => [a] -> a
solitario4 xs
    | y1 == y2 = last ys
    | otherwise = y1
    where (y1:y2:ys) = sort xs

-- 5ª definición
-- =====

```

```

solitario5 :: Ord a => [a] -> a
solitario5 xs | null ys    = y
               | otherwise = z
  where [y:ys,z:_] = group (sort xs)

-- 6ª definición
-- =====

solitario6 :: Ord a => [a] -> a
solitario6 xs =
  head [x | x <- nub xs
          , cuenta xs x == 1]

-- 7ª definición
-- =====

solitario7 :: Ord a => [a] -> a
solitario7 (a:b:xs)
  | a == b      = solitario7 (b:xs)
  | elem a (b:xs) = b
  | elem b (a:xs) = a
solitario7 [_ ,b] = b
solitario7 _      = error "Imposible"

-- Equivalencia
-- =====

-- Propiedad de equivalencia
prop_solitario_equiv :: Property
prop_solitario_equiv =
  forAll listaSolitaria (\xs -> solitario xs == solitario2 xs &&
                                solitario xs == solitario3 xs &&
                                solitario xs == solitario4 xs &&
                                solitario xs == solitario5 xs &&
                                solitario xs == solitario6 xs &&
                                solitario xs == solitario7 xs)

-- Generador de listas con al menos 3 elementos y todos iguales menos
-- uno. Por ejemplo,
--   λ> sample listaSolitaria

```

```

--      [1,0,0,0,0]
--      [0,0,-1,0,0,0]
--      [4,1,1,1]
--      [6,6,4,6]
--      [8,8,8,8,8,-4,8,8,8,8,8,8]
--      ...
listaSolitaria :: Gen [Int]
listaSolitaria = do
  n <- arbitrary
  m <- arbitrary `suchThat` (\a -> n + a > 2)
  x <- arbitrary
  y <- arbitrary `suchThat` (\a -> a /= x)
  return (replicate n x ++ [y] ++ replicate m x)

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

-- Comparación de eficiencia:
--      λ> solitario (replicate (5*10^3) 1 ++ [2])
--      2
--      (5.47 secs, 3,202,688,152 bytes)
--      λ> solitario2 (replicate (5*10^3) 1 ++ [2])
--      2
--      (2.08 secs, 1,401,603,960 bytes)
--      λ> solitario3 (replicate (5*10^3) 1 ++ [2])
--      2
--      (0.04 secs, 3,842,240 bytes)
--      λ> solitario4 (replicate (5*10^3) 1 ++ [2])
--      2
--      (0.02 secs, 1,566,472 bytes)
--      λ> solitario5 (replicate (5*10^3) 1 ++ [2])
--      2
--      (0.01 secs, 927,064 bytes)
--      λ> solitario6 (replicate (5*10^3) 1 ++ [2])
--      2
--      (0.01 secs, 1,604,176 bytes)
--      λ> solitario7 (replicate (5*10^3) 1 ++ [2])
--      2
--      (0.01 secs, 1,923,440 bytes)

```



```
--  
-- λ> solitario3 (replicate (5*10^6) 1 ++ [2])  
-- 2  
-- (4.62 secs, 3,720,123,560 bytes)  
-- λ> solitario4 (replicate (5*10^6) 1 ++ [2])  
-- 2  
-- (1.48 secs, 1,440,124,240 bytes)  
-- λ> solitario5 (replicate (5*10^6) 1 ++ [2])  
-- 2  
-- (1.40 secs, 1,440,125,936 bytes)  
-- λ> solitario6 (replicate (5*10^6) 1 ++ [2])  
-- 2  
-- (2.65 secs, 1,480,125,032 bytes)  
-- λ> solitario7 (replicate (5*10^6) 1 ++ [2])  
-- 2  
-- (2.21 secs, 1,800,126,224 bytes)  
--  
-- λ> solitario5 (2 : replicate (5*10^6) 1)  
-- 2  
-- (1.38 secs, 1,520,127,864 bytes)  
-- λ> solitario6 (2 : replicate (5*10^6) 1)  
-- 2  
-- (1.18 secs, 560,127,664 bytes)  
-- λ> solitario7 (2 : replicate (5*10^6) 1)  
-- 2  
-- (0.29 secs, 280,126,888 bytes)
```



## Ejercicio 15

# Números colinas

### Enunciado

Se dice que un número natural  $n$  es una colina si su primer dígito es igual a su último dígito, los primeros dígitos son estrictamente creciente hasta llegar al máximo, el máximo se puede repetir y los dígitos desde el máximo al final son estrictamente decrecientes.

Definir la función

---

```
esColina :: Integer -> Bool
```

---

tal que (esColina  $n$ ) se verifica si  $n$  es un número colina. Por ejemplo,

---

```
esColina 12377731 == True
esColina 1237731  == True
esColina 123731   == True
esColina 12377730 == False
esColina 12377730 == False
esColina 10377731 == False
esColina 12377701 == False
esColina 33333333 == True
```

---

## Soluciones

```

import Data.Char (digitToInt)
import Test.QuickCheck

-- 1ª definición
-- =====

esColina :: Integer -> Bool
esColina n =
  head ds == last ds &&
  esCreciente xs &&
  esDecreciente ys
  where ds = digitos n
        m  = maximum ds
        xs = takeWhile (<m) ds
        ys = dropWhile (==m) (dropWhile (<m) ds)

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

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

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

-- 2ª definición

```

```
-- =====

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

-- Equivalencia
-- =====

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

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



# Ejercicio 16

## Raíz cúbica entera

### Enunciado

Un número  $x$  es un cubo si existe un  $y$  tal que  $x = y^3$ . Por ejemplo, 8 es un cubo porque  $8 = 2^3$ .

Definir la función

---

```
raizCubicaEntera :: Integer -> Maybe Integer.
```

---

tal que (raizCubicaEntera x n) es justo la raíz cúbica del número natural x, si x es un cubo y Nothing en caso contrario. Por ejemplo,

---

raizCubicaEntera 8	==	Just 2
raizCubicaEntera 9	==	Nothing
raizCubicaEntera 27	==	Just 3
raizCubicaEntera 64	==	Just 4
raizCubicaEntera (2^30)	==	Just 1024
raizCubicaEntera (10^9000)	==	Just (10^3000)
raizCubicaEntera (5 + 10^9000)	==	Nothing

---

### Soluciones

```
import Data.Numbers.Primes (primeFactors)
import Data.List           (group)
```

```
import Test.QuickCheck
```

```
-- 1ª definición
```

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

```
raizCubicaEntera :: Integer -> Maybe Integer
```

```
raizCubicaEntera x = aux 0
```

```
  where aux y | y^3 > x    = Nothing
              | y^3 == x   = Just y
              | otherwise = aux (y+1)
```

```
-- 2ª definición
```

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

```
raizCubicaEntera2 :: Integer -> Maybe Integer
```

```
raizCubicaEntera2 x
```

```
  | y^3 == x = Just y
  | otherwise = Nothing
  where (y:_) = dropWhile (\z -> z^3 < x) [0..]
```

```
-- 3ª definición
```

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

```
raizCubicaEntera3 :: Integer -> Maybe Integer
```

```
raizCubicaEntera3 1 = Just 1
```

```
raizCubicaEntera3 x = aux (0,x)
```

```
  where aux (a,b) | d == x    = Just c
                  | c == a    = Nothing
                  | d < x     = aux (c,b)
                  | otherwise = aux (a,c)
```

```
  where c = (a+b) `div` 2
        d = c^3
```

```
-- 4ª definición
```

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

```
raizCubicaEntera4 :: Integer -> Maybe Integer
```

```
raizCubicaEntera4 x
```

```
  | y^3 == x = Just y
  | otherwise = Nothing
```



```

where y = floor ((fromIntegral x)**(1 / 3))

-- Nota. La definición anterior falla para números grandes. Por ejemplo,
--   λ> raizCubicaEntera4 (2^30)
--   Nothing
--   λ> raizCubicaEntera (2^30)
--   Just 1024

-- 5ª definición
-- =====

raizCubicaEntera5 :: Integer -> Maybe Integer
raizCubicaEntera5 x
  | all (==0) [length as `mod` 3 | as <- ass] =
    Just (product [a^((1 + length as) `div` 3) | (a:as) <- ass])
  | otherwise = Nothing
where ass = group (primeFactors x)

-- Equivalencia
-- =====

-- La propiedad es
prop_raizCubicaEntera :: Integer -> Property
prop_raizCubicaEntera x =
  x >= 0 ==>
  and [raizCubicaEntera x == f x | f <- [ raizCubicaEntera2
                                           , raizCubicaEntera3]]

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

-- Comparación de eficiencia
-- =====

--   λ> raizCubicaEntera (10^18)
--   Just 1000000
--   (1.80 secs, 1,496,137,192 bytes)
--   λ> raizCubicaEntera2 (10^18)
--   Just 1000000

```

```
-- (0.71 secs, 712,134,128 bytes)
-- λ> raizCubicaEntera3 (10^18)
-- Just 1000000
-- (0.01 secs, 196,424 bytes)
--
-- λ> raizCubicaEntera2 (5^27)
-- Just 1953125
-- (1.42 secs, 1,390,760,920 bytes)
-- λ> raizCubicaEntera3 (5^27)
-- Just 1953125
-- (0.00 secs, 195,888 bytes)
--
-- λ> raizCubicaEntera3 (10^9000) == Just (10^3000)
-- True
-- (2.05 secs, 420,941,368 bytes)
-- λ> raizCubicaEntera3 (5 + 10^9000) == Nothing
-- True
-- (2.08 secs, 420,957,576 bytes)
-- λ> raizCubicaEntera5 (5 + 10^9000) == Nothing
-- True
-- (0.03 secs, 141,248 bytes)
```

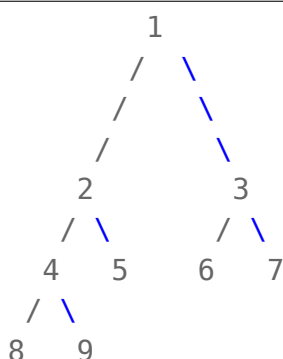
## Ejercicio 17

# Numeración de los árboles binarios completos

### Enunciado

Un **árbol binario completo** es un árbol binario que tiene todos los nodos posibles hasta el penúltimo nivel, y donde los elementos del último nivel están colocados de izquierda a derecha sin dejar huecos entre ellos.

La numeración de los árboles binarios completos se realiza a partir de la raíz, recorriendo los niveles de izquierda a derecha. Por ejemplo,



Los árboles binarios se puede representar mediante el siguiente tipo

---

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

---

Definir la función

---

```
arbolBinarioCompleto :: Int -> Arbol
```

---

tal que (arbolBinarioCompleto n) es el árbol binario completo con n nodos.  
Por ejemplo,

---

```
λ> arbolBinarioCompleto 4
N 1 (N 2 (N 4 H H) H) (N 3 H H)
λ> pPrint (arbolBinarioCompleto 9)
N 1
  (N 2
    (N 4
      (N 8 H H)
      (N 9 H H))
    (N 5 H H))
  (N 3
    (N 6 H H)
    (N 7 H H))
```

---

## Soluciones

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

arbolBinarioCompleto :: Int -> Arbol
arbolBinarioCompleto n = aux 1
  where aux i | i <= n    = N i (aux (2*i)) (aux (2*i+1))
            | otherwise = H
```

# Ejercicio 18

## Posiciones en árboles binarios

### Enunciado

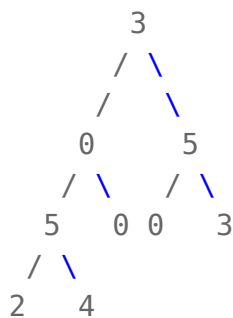
Los árboles binarios con datos en los nodos se definen por

---

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

---

Por ejemplo, el árbol



se representa por

---

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

---

```

      (N 4 H H))
    (N 0 H H))
  (N 5
    (N 0 H H)
    (N 3 H H))

```

---

Cada posición de un elemento de un árbol es una lista de movimientos hacia la izquierda o hacia la derecha. Por ejemplo, la posición de 4 en el árbol anterior es [I,I,D].

Los tipos de los movimientos y de las posiciones se definen por

```

data Movimiento = I | D deriving (Show, Eq)
type Posicion   = [Movimiento]

```

---

Definir la función

```

posiciones :: Eq b => b -> Arbol b -> [Posicion]
\end{solucion}
tal que (posiciones n a) es la lista de las posiciones del elemento n
en el árbol a. Por ejemplo,
\begin{descripcion}
posiciones 0 ejArbol == [[I],[I,D],[D,I]]
posiciones 2 ejArbol == [[I,I,I]]
posiciones 3 ejArbol == [[],[D,D]]
posiciones 4 ejArbol == [[I,I,D]]
posiciones 5 ejArbol == [[I,I],[D]]
posiciones 1 ejArbol == []

```

---

## Soluciones

```

import Data.List (nub)
import Test.QuickCheck

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

```

```
ejArbol :: Arbol Int
```

```
ejArbol = N 3
  (N 0
    (N 5
      (N 2 H H)
      (N 4 H H))
    (N 0 H H))
  (N 5
    (N 0 H H)
    (N 3 H H))
```

```
data Movimiento = I | D deriving (Show, Eq, Ord)
```

```
type Posicion = [Movimiento]
```

```
-- 1ª solución
```

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

```
posiciones :: Eq b => b -> Arbol b -> [Posicion]
```

```
posiciones n a = aux n a [[]]
```

```
  where aux _ H _ = []
        aux n' (N x i d) cs | x == n' = cs ++
          [I:xs | xs <- aux n' i cs] ++
          [D:xs | xs <- aux n' d cs]
        | otherwise = [I:xs | xs <- aux n' i cs] ++
          [D:xs | xs <- aux n' d cs]
```

```
-- 2ª solución
```

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

```
posiciones2 :: Eq b => b -> Arbol b -> [Posicion]
```

```
posiciones2 n a = aux n a [[]]
```

```
  where aux _ H _ = []
        aux n' (N x i d) cs | x == n' = cs ++ ps
        | otherwise = ps
        where ps = [I:xs | xs <- aux n' i cs] ++
          [D:xs | xs <- aux n' d cs]
```

```
-- 3ª solución
```

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

```

posiciones3 :: Eq b => b -> Arbol b -> [Posicion]
posiciones3 n a = aux n a [[]]
  where aux _ H _ = []
        aux n' (N x i d) cs | x == n' = cs ++ ps
                              | otherwise = ps
        where ps = map (I:) (aux n' i cs) ++
                  map (D:) (aux n' d cs)

-- Equivalencia
-- =====

-- Generador de árboles
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized genArbol

genArbol :: (Arbitrary a, Integral a1) => a1 -> Gen (Arbol a)
genArbol 0 = return H
genArbol n | n > 0 = N <$> arbitrary <*> subarbol <*> subarbol
  where subarbol = genArbol (div n 2)
genArbol _ = error "Imposible"

-- La propiedad es
prop_posiciones_equiv :: Arbol Int -> Bool
prop_posiciones_equiv a =
  and [posiciones n a == posiciones2 n a | n <- xs] &&
  and [posiciones n a == posiciones3 n a | n <- xs]
  where xs = take 3 (elementos a)

-- (elementos a) son los elementos del árbol a. Por ejemplo,
-- elementos ejArbol == [3,0,5,2,4]
elementos :: Eq b => Arbol b -> [b]
elementos H = []
elementos (N x i d) = nub (x : elementos i ++ elementos d)

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

```



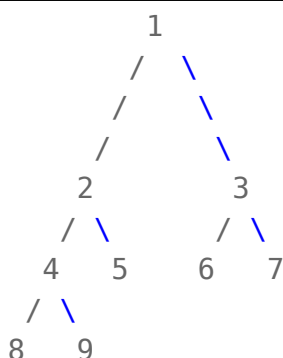
## Ejercicio 19

# Posiciones en árboles binarios completos

### Enunciado

Un **árbol binario completo** es un árbol binario que tiene todos los nodos posibles hasta el penúltimo nivel, y donde los elementos del último nivel están colocados de izquierda a derecha sin dejar huecos entre ellos.

La numeración de los árboles binarios completos se realiza a partir de la raíz, recorriendo los niveles de izquierda a derecha. Por ejemplo,



Los árboles binarios se puede representar mediante el siguiente tipo

---

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

---

Cada posición de un elemento de un árbol es una lista de movimientos hacia la izquierda o hacia la derecha. Por ejemplo, la posición de 9 en el árbol anterior es [I,I,D].

Los tipos de los movimientos y de las posiciones se definen por

---

```
data Movimiento = I | D deriving (Show, Eq)
type Posicion   = [Movimiento]
```

---

Definir la función

---

```
posicionDeElemento :: Integer -> Posicion
```

---

tal que (posicionDeElemento n) es la posición del elemento n en el árbol binario completo. Por ejemplo,

---

```
posicionDeElemento 6 == [D,I]
posicionDeElemento 7 == [D,D]
posicionDeElemento 9 == [I,I,D]
posicionDeElemento 1 == []

length (posicionDeElemento (10^50000)) == 166096
```

---

## Soluciones

```
import Test.QuickCheck

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

data Movimiento = I | D deriving (Show, Eq)

type Posicion = [Movimiento]

-- 1ª solución
-- =====

posicionDeElemento :: Integer -> Posicion
```

```

posicionDeElemento n =
  head (posiciones n (arbolBinarioCompleto n))

-- (arbolBinarioCompleto n) es el árbol binario completo con n
-- nodos. Por ejemplo,
--   λ> arbolBinarioCompleto 4
--   N 1 (N 2 (N 4 H H) H) (N 3 H H)
--   λ> pPrint (arbolBinarioCompleto 9)
--   N 1
--     (N 2
--       (N 4
--         (N 8 H H)
--         (N 9 H H))
--       (N 5 H H))
--     (N 3
--       (N 6 H H)
--       (N 7 H H))
arbolBinarioCompleto :: Integer -> Arbol
arbolBinarioCompleto n = aux 1
  where aux i | i <= n    = N i (aux (2*i)) (aux (2*i+1))
            | otherwise = H

-- (posiciones n a) es la lista de las posiciones del elemento n
-- en el árbol a. Por ejemplo,
--   posiciones 9 (arbolBinarioCompleto 9) == [[I,I,D]]
posiciones :: Integer -> Arbol -> [Posicion]
posiciones n a = aux n a [[]]
  where aux _ H _ = []
        aux n' (N x i d) cs | x == n' = cs ++ ps
                           | otherwise = ps
        where ps = map (I:) (aux n' i cs) ++
                  map (D:) (aux n' d cs)

-- 2ª solución
-- =====

posicionDeElemento2 :: Integer -> Posicion
posicionDeElemento2 1 = []
posicionDeElemento2 n
  | even n    = posicionDeElemento2 (n `div` 2) ++ [I]

```

```

    | otherwise = posicionDeElemento2 (n `div` 2) ++ [D]

-- 3ª solución
-- =====

posicionDeElemento3 :: Integer -> Posicion
posicionDeElemento3 = reverse . aux
  where aux 1 = []
        aux n | even n    = I : aux (n `div` 2)
              | otherwise = D : aux (n `div` 2)

-- 4ª solución
-- =====

posicionDeElemento4 :: Integer -> Posicion
posicionDeElemento4 n =
  [f x | x <- tail (reverse (binario n))]
  where f 0 = I
        f 1 = D
        f _ = error "Imposible"

-- (binario n) es la lista de los dígitos de la representación binaria
-- de n. Por ejemplo,
--   binario 11 == [1,1,0,1]
binario :: Integer -> [Integer]
binario n
  | n < 2    = [n]
  | otherwise = n `mod` 2 : binario (n `div` 2)

-- Equivalencia
-- =====

-- La propiedad es
prop_posicionDeElemento_equiv :: Positive Integer -> Bool
prop_posicionDeElemento_equiv (Positive n) =
  posicionDeElemento n == posicionDeElemento2 n &&
  posicionDeElemento n == posicionDeElemento3 n &&
  posicionDeElemento n == posicionDeElemento4 n

-- La comprobación es

```

```
--      λ> quickCheck prop_posicionDeElemento_equiv
--      +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

--      λ> posicionDeElemento (10^7)
--      [I,I,D,D,I,I,I,D,I,I,D,I,D,D,I,D,I,I,I,I,I,I,I]
--      (5.72 secs, 3,274,535,328 bytes)
--      λ> posicionDeElemento2 (10^7)
--      [I,I,D,D,I,I,I,D,I,I,D,I,D,D,I,D,I,I,I,I,I,I,I]
--      (0.01 secs, 189,560 bytes)
--      λ> posicionDeElemento3 (10^7)
--      [I,I,D,D,I,I,I,D,I,I,D,I,D,D,I,D,I,I,I,I,I,I,I]
--      (0.01 secs, 180,728 bytes)
--      λ> posicionDeElemento4 (10^7)
--      [I,I,D,D,I,I,I,D,I,I,D,I,D,D,I,D,I,I,I,I,I,I,I]
--      (0.01 secs, 184,224 bytes)
--
--      λ> length (posicionDeElemento2 (10^4000))
--      13287
--      (2.80 secs, 7,672,011,280 bytes)
--      λ> length (posicionDeElemento3 (10^4000))
--      13287
--      (0.03 secs, 19,828,744 bytes)
--      λ> length (posicionDeElemento4 (10^4000))
--      13287
--      (0.03 secs, 18,231,536 bytes)
--
--      λ> length (posicionDeElemento3 (10^50000))
--      166096
--      (1.34 secs, 1,832,738,136 bytes)
--      λ> length (posicionDeElemento4 (10^50000))
--      166096
--      (1.70 secs, 1,812,806,080 bytes)
```



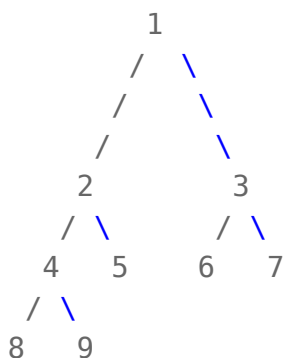
## Ejercicio 20

# Elemento del árbol binario completo según su posición

### Enunciado

Un **árbol binario completo** es un árbol binario que tiene todos los nodos posibles hasta el penúltimo nivel, y donde los elementos del último nivel están colocados de izquierda a derecha sin dejar huecos entre ellos.

La numeración de los árboles binarios completos se realiza a partir de la raíz, recorriendo los niveles de izquierda a derecha. Por ejemplo,



Los árboles binarios se puede representar mediante el siguiente tipo

---

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

---

Cada posición de un elemento de un árbol es una lista de movimientos hacia la izquierda o hacia la derecha. Por ejemplo, la posición de 9 en el árbol anterior es [I,I,D].

Los tipos de los movimientos y de las posiciones se definen por

---

```
data Movimiento = I | D deriving (Show, Eq)
type Posicion   = [Movimiento]
```

---

Definir la función

---

```
elementoEnPosicion :: Posicion -> Integer
```

---

tal que (elementoEnPosicion ms) es el elemento en la posición ms. Por ejemplo,

---

```
elementoEnPosicion [D,I]    == 6
elementoEnPosicion [D,D]    == 7
elementoEnPosicion [I,I,D]  == 9
elementoEnPosicion []       == 1
```

---

## Soluciones

```
import Test.QuickCheck
```

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

```
data Movimiento = I | D deriving (Show, Eq)
```

```
type Posicion = [Movimiento]
```

```
-- 1ª solución
-- =====
```

```
elementoEnPosicion :: Posicion -> Integer
elementoEnPosicion ms =
  aux ms (arbolBinarioCompleto (2^(1 + length ms)))
```



```

where aux []      (N x _ _) = x
      aux (I:ms') (N _ i _) = aux ms' i
      aux (D:ms') (N _ _ d) = aux ms' d
      aux _      _         = error "Imposible"

-- (arbolBinarioCompleto n) es el árbol binario completo con n
-- nodos. Por ejemplo,
--   λ> arbolBinarioCompleto 4
--   N 1 (N 2 (N 4 H H) H) (N 3 H H)
--   λ> pPrint (arbolBinarioCompleto 9)
--   N 1
--     (N 2
--       (N 4
--         (N 8 H H)
--         (N 9 H H))
--       (N 5 H H))
--     (N 3
--       (N 6 H H)
--       (N 7 H H))
arbolBinarioCompleto :: Integer -> Arbol
arbolBinarioCompleto n = aux 1
  where aux i | i <= n    = N i (aux (2*i)) (aux (2*i+1))
            | otherwise = H

-- 2ª solución
-- =====

elementoEnPosicion2 :: Posicion -> Integer
elementoEnPosicion2 = aux . reverse
  where aux []      = 1
        aux (I:ms) = 2 * aux ms
        aux (D:ms) = 2 * aux ms + 1

-- Equivalencia
-- =====

-- La propiedad es
prop_elementoEnPosicion_equiv :: Positive Integer -> Bool
prop_elementoEnPosicion_equiv (Positive n) =
  elementoEnPosicion ps == n &&

```

```

elementoEnPosicion2 ps == n
where ps = posicionDeElemento n

-- (posicionDeElemento n) es la posición del elemento n en el
-- árbol binario completo. Por ejemplo,
--   posicionDeElemento 6 == [D,I]
--   posicionDeElemento 7 == [D,D]
--   posicionDeElemento 9 == [I,I,D]
--   posicionDeElemento 1 == []
posicionDeElemento :: Integer -> Posicion
posicionDeElemento n =
  [f x | x <- tail (reverse (binario n))]
  where f 0 = I
        f 1 = D
        f _ = error "Imposible"

-- (binario n) es la lista de los dígitos de la representación binaria
-- de n. Por ejemplo,
--   binario 11 == [1,1,0,1]
binario :: Integer -> [Integer]
binario n
  | n < 2      = [n]
  | otherwise = n `mod` 2 : binario (n `div` 2)

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

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

--   λ> length (show (elementoEnPosicion (replicate (3*10^5) D)))
--   90310
--   (1.96 secs, 11,518,771,016 bytes)
--   λ> length (show (elementoEnPosicion2 (replicate (3*10^5) D)))
--   90310
--   (14.32 secs, 11,508,181,176 bytes)

```

# Ejercicio 21

## Aproximación entre pi y e

### Enunciado

El día 11 de noviembre, se publicó en la cuenta de Twitter de [Fermat's Library](#) la siguiente curiosa identidad que relaciona los números e y pi:

$$\frac{1}{\pi^2 + 1} + \frac{1}{4\pi^2 + 1} + \frac{1}{9\pi^2 + 1} + \frac{1}{16\pi^2 + 1} + \dots = \frac{1}{e^2 - 1}$$

Definir las siguientes funciones:

---

```
sumaTerminos :: Int -> Double
aproximacion  :: Double -> Int
```

---

tales que

- (sumaTerminos n) es la suma de los primeros n términos de la serie

$$\frac{1}{\pi^2 + 1} + \frac{1}{4\pi^2 + 1} + \frac{1}{9\pi^2 + 1} + \frac{1}{16\pi^2 + 1} + \dots = \frac{1}{e^2 - 1}$$

Por ejemplo,

---

sumaTerminos 10	==	0.14687821811081034
sumaTerminos 100	==	0.15550948345688423
sumaTerminos 1000	==	0.15641637221314514
sumaTerminos 10000	==	0.15650751113789382

---

- (aproximación x) es el menor número de términos que hay que sumar de la serie anterior para que se diferencie (en valor absoluto) de  $\frac{1}{e^2 - 1}$  menos que x. Por ejemplo,

---

aproximacion 0.1	==	1
aproximacion 0.01	==	10
aproximacion 0.001	==	101
aproximacion 0.0001	==	1013

---

## Soluciones

-- 1ª definición de sumaTerminos

sumaTerminos :: Int -> Double

sumaTerminos n =

sum [1 / ((x ^ 2) \* (pi ^ 2)) + 1) | x <- [1 .. fromIntegral n]]

-- 2ª definición de sumaTerminos

sumaTerminos2 :: Int -> Double

sumaTerminos2 0 = 0

sumaTerminos2 n = 1 / (m^2 \* pi^2 + 1) + sumaTerminos2 (n-1)

where m = fromIntegral n

-- Definición de aproximacion

aproximacion :: Double -> Int

aproximacion x =

head [n | n <- [0..]

, abs (sumaTerminos n - 1 / (e^2 - 1)) < x]

where e = exp 1

## Ejercicio 22

# Menor contenedor de primos

### Enunciado

El  $n$ -ésimo menor contenedor de primos es el menor número que contiene como subcadenas los primeros  $n$  primos. Por ejemplo, el 6º menor contenedor de primos es 113257 ya que es el menor que contiene como subcadenas los 6 primeros primos (2, 3, 5, 7, 11 y 13).

Definir la función

---

```
menorContenedor :: Int -> Int
```

---

tal que `(menorContenedor n)` es el  $n$ -ésimo menor contenedor de primos. Por ejemplo,

---

```
menorContenedor 1 == 2
menorContenedor 2 == 23
menorContenedor 3 == 235
menorContenedor 4 == 2357
menorContenedor 5 == 112357
menorContenedor 6 == 113257
```

---

### Soluciones

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

```
-- 1ª solución
```

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

```
menorContenedor :: Int -> Int
```

```
menorContenedor n =
```

```
    head [x | x <- [2..]
```

```
        , and [contenido y x | y <- take n primes]]
```

```
contenido :: Int -> Int -> Bool
```

```
contenido x y =
```

```
    show x `isInfixOf` show y
```

```
-- 2ª solución
```

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

```
menorContenedor2 :: Int -> Int
```

```
menorContenedor2 n =
```

```
    head [x | x <- [2..]
```

```
        , all (`contenido` x) (take n primes)]
```

## Ejercicio 23

# Árbol de computación de Fibonacci

### Enunciado

La sucesión de Fibonacci es

---

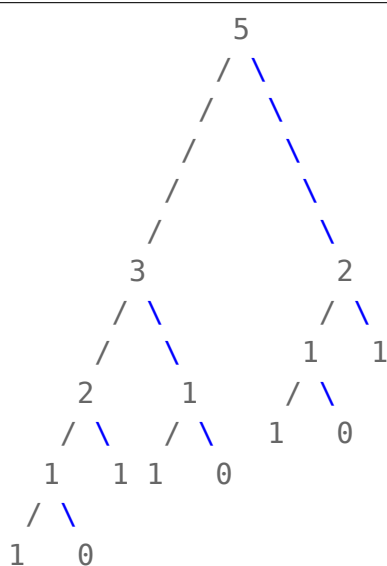
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, \dots$

---

cuyos dos primeros términos son 0 y 1 y los restantes se obtienen sumando los dos anteriores.

El árbol de computación de su 5º término es

---



que, usando los árboles definidos por

---

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

---

se puede representar por

---

```
N 5
  (N 3
    (N 2
      (N 1 (H 1) (H 0))
      (H 1))
    (N 1 (H 1) (H 0)))
  (N 2
    (N 1 (H 1) (H 0))
    (H 1))
```

---

Definir las funciones

---

```
arbolFib      :: Int -> Arbol
nElementosArbolFib :: Int -> Int
```

---

tales que

- (arbolFib n) es el árbol de computación del n-ésimo término de la sucesión de Fibonacci. Por ejemplo,

---

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



```

(N 3
  (N 2
    (N 1 (H 1) (H 0))
    (H 1))
  (N 1 (H 1) (H 0)))
(N 2
  (N 1 (H 1) (H 0))
  (H 1))
(N 3
  (N 2
    (N 1 (H 1) (H 0)) (H 1))
  (N 1 (H 1) (H 0)))

```

- (nElementosArbolFib n) es el número de elementos en el árbol de computación del n-ésimo término de la sucesión de Fibonacci. Por ejemplo,

---

```

nElementosArbolFib 5 == 15
nElementosArbolFib 6 == 25
nElementosArbolFib 30 == 2692537

```

---

## Soluciones

```

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

```

```

-- 1ª definición
-- =====

```

```

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

```

```

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

```

```

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

-- 2ª definición
-- =====

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

-- 3ª definición
-- =====

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

-- 1ª definición de nElementosArbolFib
-- =====

nElementosArbolFib :: Int -> Int
nElementosArbolFib = length . elementos . arbolFib3

-- (elementos a) es la lista de elementos del árbol a. Por ejemplo,
--     λ> elementos (arbolFib 5)
--     [5,3,2,1,1,0,1,1,1,0,2,1,1,0,1]
--     λ> elementos (arbolFib 6)
--     [8,5,3,2,1,1,0,1,1,1,0,2,1,1,0,1,3,2,1,1,0,1,1,1,0]
elementos :: Arbol -> [Int]

```

```
elementos (H x)      = [x]
elementos (N x i d) = x : elementos i ++ elementos d

-- 2ª definición de nElementosArbolFib
-- =====

nElementosArbolFib2 :: Int -> Int
nElementosArbolFib2 0 = 1
nElementosArbolFib2 1 = 1
nElementosArbolFib2 n = 1 + nElementosArbolFib2 (n-1)
                        + nElementosArbolFib2 (n-2)
```



## Ejercicio 24

# Entre dos conjuntos

### Enunciado

Se dice que un  $x$  número se encuentra entre dos conjuntos  $xs$  e  $ys$  si  $x$  es divisible por todos los elementos de  $xs$  y todos los elementos de  $zs$  son divisibles por  $x$ . Por ejemplo, 12 se encuentra entre los conjuntos 2, 6 y 24, 36.

Definir la función

---

```
entreDosConjuntos :: [Int] -> [Int] -> [Int]
```

---

tal que `(entreDosConjuntos xs ys)` es la lista de elementos entre  $xs$  e  $ys$  (se supone que  $xs$  e  $ys$  son listas no vacías de números enteros positivos). Por ejemplo,

---

```
entreDosConjuntos [2,6] [24,36] == [6,12]
entreDosConjuntos [2,4] [32,16,96] == [4,8,16]
```

---

Otros ejemplos

---

```
λ> (xs,a) = ([1..15],product xs)
λ> length (entreDosConjuntos xs [a,2*a..10*a])
270
λ> (xs,a) = ([1..16],product xs)
λ> length (entreDosConjuntos xs [a,2*a..10*a])
360
```

---

## Soluciones

```
import Test.QuickCheck
```

```
-- 1ª solución
```

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

```
entreDosConjuntos :: [Int] -> [Int] -> [Int]
```

```
entreDosConjuntos xs ys =
```

```
  [z | z <- [a..b]
    , and [z `mod` x == 0 | x <- xs]
    , and [y `mod` z == 0 | y <- ys]]
```

```
  where a = maximum xs
```

```
        b = minimum ys
```

```
-- 2ª solución
```

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

```
entreDosConjuntos2 :: [Int] -> [Int] -> [Int]
```

```
entreDosConjuntos2 xs ys =
```

```
  [z | z <- [a..b]
    , all (`divideA` z) xs
    , all (z `divideA`) ys]
```

```
  where a = mcmL xs
```

```
        b = mcdL ys
```

```
--    mcmL [2,3,18] == 18
```

```
--    mcmL [2,3,15] == 30
```

```
mcdL :: [Int] -> Int
```

```
mcdL [x] = x
```

```
mcdL (x:xs) = gcd x (mcdL xs)
```

```
--    mcmL [12,30,18] == 6
```

```
--    mcmL [12,30,14] == 2
```

```
mcmL :: [Int] -> Int
```

```
mcmL [x] = x
```

```
mcmL (x:xs) = lcm x (mcmL xs)
```

```
divideA :: Int -> Int -> Bool
```

```
divideA x y = y `mod` x == 0
```

```
-- 3ª solución
```

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

```
entreDosConjuntos3 :: [Int] -> [Int] -> [Int]
```

```
entreDosConjuntos3 xs ys =  
  [z | z <- [a..b]  
    , all (`divideA` z) xs  
    , all (z `divideA`) ys]  
  where a = mcmL2 xs  
        b = mcdL2 ys
```

```
-- Definición equivalente
```

```
mcdL2 :: [Int] -> Int
```

```
mcdL2 = foldl1 gcd
```

```
-- Definición equivalente
```

```
mcmL2 :: [Int] -> Int
```

```
mcmL2 = foldl1 lcm
```

```
-- 4ª solución
```

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

```
entreDosConjuntos4 :: [Int] -> [Int] -> [Int]
```

```
entreDosConjuntos4 xs ys =  
  [z | z <- [a,a+a..b]  
    , z `divideA` b]  
  where a = mcmL2 xs  
        b = mcdL2 ys
```

```
-- 5ª solución
```

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

```
entreDosConjuntos5 :: [Int] -> [Int] -> [Int]
```

```
entreDosConjuntos5 xs ys =  
  filter (`divideA` b) [a,a+a..b]  
  where a = mcmL2 xs  
        b = mcdL2 ys
```

```
-- Equivalencia
```

```

-- =====

-- Para comprobar la equivalencia se define el tipo de listas no vacías
-- de números enteros positivos:
newtype ListaNoVacíaDePositivos = L [Int]
  deriving Show

-- genListaNoVacíaDePositivos es un generador de listas no vacías de
-- enteros positivos. Por ejemplo,
--   λ> sample genListaNoVacíaDePositivos
--   L [1]
--   L [1,2,2]
--   L [4,3,4]
--   L [1,6,5,2,4]
--   L [2,8]
--   L [11]
--   L [13,2,3]
--   L [7,3,9,15,11,12,13,3,9,6,13,3]
--   L [16,2,11,10,6,5,16,4,1,15,9,11,8,15,2,15,7]
--   L [5,4,9,13,5,6,7]
--   L [7,4,6,12,2,11,6,14,14,13,14,11,6,2,18,8,16,2,13,9]
genListaNoVacíaDePositivos :: Gen ListaNoVacíaDePositivos
genListaNoVacíaDePositivos = do
  x <- arbitrary
  xs <- arbitrary
  return (L (map ((+1) . abs) (x:xs)))

-- Generación arbitraria de listas no vacías de enteros positivos.
instance Arbitrary ListaNoVacíaDePositivos where
  arbitrary = genListaNoVacíaDePositivos

-- La propiedad es
prop_entreDosConjuntos_equiv ::
  ListaNoVacíaDePositivos
  -> ListaNoVacíaDePositivos
  -> Bool
prop_entreDosConjuntos_equiv (L xs) (L ys) =
  entreDosConjuntos xs ys == entreDosConjuntos2 xs ys &&
  entreDosConjuntos xs ys == entreDosConjuntos3 xs ys &&
  entreDosConjuntos xs ys == entreDosConjuntos4 xs ys &&

```



```
entreDosConjuntos xs ys == entreDosConjuntos5 xs ys
```

```
-- La comprobación es
```

```
-- λ> quickCheck prop_entreDosConjuntos_equiv
```

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

```
-- Comparación de eficiencia
```

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

```
-- λ> (xs,a) = ([1..10],product xs)
```

```
-- λ> length (entreDosConjuntos xs [a,2*a..10*a])
```

```
-- 36
```

```
-- (5.08 secs, 4,035,689,200 bytes)
```

```
-- λ> length (entreDosConjuntos2 xs [a,2*a..10*a])
```

```
-- 36
```

```
-- (3.75 secs, 2,471,534,072 bytes)
```

```
-- λ> length (entreDosConjuntos3 xs [a,2*a..10*a])
```

```
-- 36
```

```
-- (3.73 secs, 2,471,528,664 bytes)
```

```
-- λ> length (entreDosConjuntos4 xs [a,2*a..10*a])
```

```
-- 36
```

```
-- (0.01 secs, 442,152 bytes)
```

```
-- λ> length (entreDosConjuntos5 xs [a,2*a..10*a])
```

```
-- 36
```

```
-- (0.00 secs, 374,824 bytes)
```



## Ejercicio 25

# Expresiones aritméticas generales

### Enunciado

Las expresiones aritméticas. generales se contruyen con las sumas generales (sumatorios) y productos generales (productorios). Su tipo es

---

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

---

Por ejemplo, la expresión  $(2 * (1 + 2 + 1) * (2 + 3)) + 1$  se representa por `S [P [N 2, S [N 1, N 2, N 1], S [N 2, N 3]], N 1]`

Definir la función

---

```
valor :: Expresion -> Int
```

---

tal que (valor e) es el valor de la expresión e. Por ejemplo,

---

```
λ> valor (S [P [N 2, S [N 1, N 2, N 1], S [N 2, N 3]], N 1))
41
```

---

## Soluciones

```
data Expression = N Int
                | S [Expression]
                | P [Expression]
    deriving Show

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

## Ejercicio 26

# Superación de límites

*Todo necio confunde valor y precio.*

---

Antonio Machado

## Enunciado

Una sucesión de puntuaciones se puede representar mediante una lista de números. Por ejemplo, [7,5,9,9,4,5,4,2,5,9,12,1]. En la lista anterior, los puntos en donde se alcanzan un nuevo máximo son 7, 9 y 12 (porque son mayores que todos sus anteriores) y en donde se alcanzan un nuevo mínimo son 7, 5, 4, 2 y 1 (porque son menores que todos sus anteriores). Por tanto, el máximo se ha superado 2 veces y el mínimo 4 veces.

Definir las funciones

---

```
nuevosMaximos :: [Int] -> [Int]
nuevosMinimos :: [Int] -> [Int]
nRupturas      :: [Int] -> (Int,Int)
```

---

tales que

- (nuevosMaximos xs) es la lista de los nuevos máximos de xs. Por ejemplo,

---

```
nuevosMaximos [7,5,9,9,4,5,4,2,5,9,12,1] == [7,9,12]
```

---

- (nuevosMinimos xs) es la lista de los nuevos mínimos de xs. Por ejemplo,

---

```
nuevosMinimos [7,5,9,9,4,5,4,2,5,9,12,1] == [7,5,4,2,1]
```

---

- (nRupturas xs) es el par formado por el número de veces que se supera el máximo y el número de veces que se supera el mínimo en xs. Por ejemplo,

---

```
nRupturas [7,5,9,9,4,5,4,2,5,9,12,1] == (2,4)
```

---

## Soluciones

```
import Data.List (group, inits)
```

```
nuevosMaximos :: [Int] -> [Int]
```

```
nuevosMaximos xs = map head (group (map maximum xss))
```

```
  where xss = tail (inits xs)
```

```
nuevosMinimos :: [Int] -> [Int]
```

```
nuevosMinimos xs = map head (group (map minimum xss))
```

```
  where xss = tail (inits xs)
```

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

```
nRupturas [] = (0,0)
```

```
nRupturas xs =
```

```
  ( length (nuevosMaximos xs) - 1
```

```
  , length (nuevosMinimos xs) - 1)
```

## Ejercicio 27

# Intercambio de la primera y última columna de una matriz

*¡Que difícil es,  
cuando todo baja  
no bajar también!*

---

Antonio Machado

## Enunciado

Las matrices se pueden representar mediante listas de listas. Por ejemplo, la matriz

---

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

---

se puede representar por la lista

---

```
[[8,9,7,6],[4,7,6,5],[3,2,1,8]]
```

---

Definir la función

---

```
intercambia :: [[a]] -> [[a]]
```

---

tal que (`intercambia xss`) es la matriz obtenida intercambiando la primera y la última columna de `xss`. Por ejemplo,

---

```
λ> intercambia [[8,9,7,6],[4,7,6,5],[3,2,1,8]]
[[6,9,7,8],[5,7,6,4],[8,2,1,3]]
```

---

## Soluciones

```
intercambia :: [[a]] -> [[a]]
intercambia = map intercambiaL
```

```
-- (intercambiaL xs) es la lista obtenida intercambiando el primero y el
-- último elemento de xs. Por ejemplo,
--   intercambiaL [8,9,7,6] == [6,9,7,8]
intercambiaL :: [a] -> [a]
intercambiaL xs =
  last xs : tail (init xs) ++ [head xs]
```



## Ejercicio 28

# Números primos de Pierpont

*La memoria es infiel: no sólo borra y confunde, sino que, a veces, inventa, para desorientarnos.*

---

Antonio Machado

## Enunciado

Un **número primo de Pierpont** es un número primo de la forma  $2^u 3^v + 1$ , para  $u$  y  $v$  enteros no negativos.

Definir la sucesión

---

```
primosPierpont :: [Integer]
```

---

tal que sus elementos son los números primos de Pierpont. Por ejemplo,

---

```
λ> take 20 primosPierpont
[2,3,5,7,13,17,19,37,73,97,109,163,193,257,433,487,577,769,1153,1297]
λ> primosPierpont !! 49
8503057
```

---

## Soluciones

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

primosPierpont :: [Integer]
primosPierpont =
  [n | n <- primes
    , primoPierpont n]

primoPierpont :: Integer -> Bool
primoPierpont n =
  primeFactors (n-1) `contenidoEn` [2,3]

-- (contenidoEn xs ys) se verifica si xs está contenido en ys. Por
-- ejemplo,
--   contenidoEn [2,3,2,2,3] [2,3] == True
--   contenidoEn [2,3,2,2,1] [2,3] == False
contenidoEn :: [Integer] -> [Integer] -> Bool
contenidoEn xs ys =
  all (`elem` ys) xs
```

## Ejercicio 29

### Grado exponencial

*De cada diez novedades que pretenden descubrirnos, nueve son tonterías. La décima y última, que no es necesidad, resulta a última hora que tampoco es nueva.*

---

Antonio Machado

### Enunciado

El grado exponencial de un número  $n$  es el menor número  $e$  mayor que 1 tal que  $n$  es una subcadena de  $n^e$ . Por ejemplo, el grado exponencial de 2 es 5 ya que 2 es una subcadena de 32 (que es  $2^5$ ) y no es subcadena de las anteriores potencias de 2 (2, 4 y 16). El grado exponencial de 25 es 2 porque 25 es una subcadena de 625 (que es  $25^2$ ).

Definir la función

---

```
gradoExponencial :: Integer -> Integer
```

---

tal que (gradoExponencial  $n$ ) es el grado exponencial de  $n$ . Por ejemplo,

---

```
gradoExponencial 2    == 5
gradoExponencial 25   == 2
gradoExponencial 15   == 26
```

---

```
gradoExponencial 1093 == 100
gradoExponencial 10422 == 200
gradoExponencial 11092 == 300
```

---

## Soluciones

```
import Test.QuickCheck
import Data.List (genericLength, isInfixOf)
```

```
-- 1ª solución
-- =====
```

```
gradoExponencial :: Integer -> Integer
gradoExponencial n =
    head [e | e <- [2..]
          , show n `isInfixOf` show (n^e)]
```

```
-- 2ª solución
-- =====
```

```
gradoExponencial2 :: Integer -> Integer
gradoExponencial2 n =
    2 + genericLength (takeWhile noSubcadena (potencias n))
    where c          = show n
          noSubcadena x = not (c `isInfixOf` show x)
```

```
-- (potencias n) es la lista de las potencias de n a partir de n^2. Por
-- ejemplo,
```

```
-- λ> take 10 (potencias 2)
-- [4,8,16,32,64,128,256,512,1024,2048]
```

```
potencias :: Integer -> [Integer]
potencias n =
    iterate (*n) (n^2)
```

```
-- 3ª solución
-- =====
```

```
gradoExponencial3 :: Integer -> Integer
```

```
gradoExponencial3 n = aux 2
  where aux x
        | cs `isInfixOf` show (n^x) = x
        | otherwise                  = aux (x+1)
        cs = show n

-- Equivalencia
-- =====

-- La propiedad es
prop_gradosExponencial_equiv :: (Positive Integer) -> Bool
prop_gradosExponencial_equiv (Positive n) =
  gradoExponencial n == gradoExponencial2 n &&
  gradoExponencial n == gradoExponencial3 n

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

## Referencia

Basado en la [sucesión A045537](#) de la OEIS.



## Ejercicio 30

# Divisores propios maximales

*Moneda que está en la mano  
quizá se deba guardar;  
la monedita del alma  
se pierde si no se da.*

---

Antonio Machado

## Enunciado

Se dice que  $a$  es un divisor propio maximal de un número  $b$  si  $a$  es un divisor de  $b$  distinto de  $b$  y no existe ningún número  $c$  tal que  $a < c < b$ ,  $a$  es un divisor de  $c$  y  $c$  es un divisor de  $b$ . Por ejemplo, 15 es un divisor propio maximal de 30, pero 5 no lo es.

Definir las funciones

---

```
divisoresPropiosMaximales :: Integer -> [Integer]
nDivisoresPropiosMaximales :: Integer -> Integer
```

---

tales que

- `(divisoresPropiosMaximales x)` es la lista de los divisores propios maximales de  $x$ . Por ejemplo,

---

```
divisoresPropiosMaximales 30 == [6,10,15]
divisoresPropiosMaximales 420 == [60,84,140,210]
```

---

```
divisoresPropiosMaximales 7 == [1]
length (divisoresPropiosMaximales (product [1..3*10^4])) == 3245
```

---

- (nDivisoresPropiosMaximales x) es el número de divisores propios maximales de x. Por ejemplo,

---

```
nDivisoresPropiosMaximales 30 == 3
nDivisoresPropiosMaximales 420 == 4
nDivisoresPropiosMaximales 7 == 1
nDivisoresPropiosMaximales (product [1..3*10^4]) == 3245
```

---

## Soluciones

```
import Data.Numbers.Primes (primeFactors)
import Data.List (genericLength, group, nub)
import Test.QuickCheck

-- 1ª definición de divisoresPropiosMaximales
-- =====

divisoresPropiosMaximales :: Integer -> [Integer]
divisoresPropiosMaximales x =
  [y | y <- divisoresPropios x
    , null [z | z <- divisoresPropios x
      , y < z
      , z `mod` y == 0]]

-- (divisoresPropios x) es la lista de los divisores propios de x; es
-- decir, de los divisores de x distintos de x. Por ejemplo,
--   divisoresPropios 30 == [1,2,3,5,6,10,15]
divisoresPropios :: Integer -> [Integer]
divisoresPropios x =
  [y | y <- [1..x-1]
    , x `mod` y == 0]

-- 2ª definición de divisoresPropiosMaximales
-- =====

divisoresPropiosMaximales2 :: Integer -> [Integer]
```



```

divisoresPropiosMaximales2 x =
  reverse [x `div` y | y <- nub (primeFactors x)]

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

-- La propiedad es
prop_divisoresPropiosMaximales_equiv :: Positive Integer -> Bool
prop_divisoresPropiosMaximales_equiv (Positive x) =
  divisoresPropiosMaximales x == divisoresPropiosMaximales2 x

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

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

--   λ> length (divisoresPropiosMaximales (product [1..10]))
--   4
--   (13.33 secs, 7,037,241,776 bytes)
--   λ> length (divisoresPropiosMaximales2 (product [1..10]))
--   4
--   (0.00 secs, 135,848 bytes)

-- 1ª definición de nDivisoresPropiosMaximales
-- =====

nDivisoresPropiosMaximales :: Integer -> Integer
nDivisoresPropiosMaximales =
  genericLength . divisoresPropiosMaximales

-- 2ª definición de nDivisoresPropiosMaximales
-- =====

nDivisoresPropiosMaximales2 :: Integer -> Integer
nDivisoresPropiosMaximales2 =
  genericLength . divisoresPropiosMaximales2

-- 3ª definición de nDivisoresPropiosMaximales

```

```

-- =====

nDivisoresPropiosMaximales3 :: Integer -> Integer
nDivisoresPropiosMaximales3 =
  genericLength . group . primeFactors

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

-- La propiedad es
prop_nDivisoresPropiosMaximales_equiv :: Positive Integer -> Bool
prop_nDivisoresPropiosMaximales_equiv (Positive x) =
  nDivisoresPropiosMaximales x == nDivisoresPropiosMaximales3 x &&
  nDivisoresPropiosMaximales2 x == nDivisoresPropiosMaximales3 x

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

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

--   λ> nDivisoresPropiosMaximales2 (product [1..10])
--   4
--   (13.33 secs, 7,037,242,536 bytes)
--   λ> nDivisoresPropiosMaximales2 (product [1..10])
--   4
--   (0.00 secs, 135,640 bytes)
--   λ> nDivisoresPropiosMaximales3 (product [1..10])
--   4
--   (0.00 secs, 135,232 bytes)
--
--   λ> nDivisoresPropiosMaximales2 (product [1..3*10^4])
--   3245
--   (3.12 secs, 4,636,274,040 bytes)
--   λ> nDivisoresPropiosMaximales3 (product [1..3*10^4])
--   3245
--   (3.06 secs, 4,649,295,056 bytes)

```

# Ejercicio 31

## Árbol de divisores

*¿Dónde está la utilidad  
de nuestras utilidades?  
Volvamos a la verdad:  
vanidad de vanidades.*

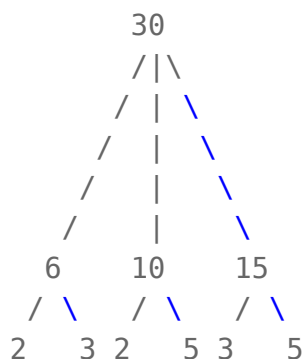
---

Antonio Machado

### Enunciado

Se dice que  $a$  es un divisor propio maximal de un número  $b$  si  $a$  es un divisor de  $b$  distinto de  $b$  y no existe ningún número  $c$  tal que  $a < c < b$ ,  $a$  es un divisor de  $c$  y  $c$  es un divisor de  $b$ . Por ejemplo, 15 es un divisor propio maximal de 30, pero 5 no lo es.

El árbol de los divisores de un número  $x$  es el árbol que tiene como raíz el número  $x$  y cada nodo tiene como hijos sus divisores propios maximales. Por ejemplo, el árbol de divisores de 30 es



Usando el tipo de dato

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

el árbol anterior se representa por

```
N 30
  [N 6
    [N 2 [N 1 []],
     N 3 [N 1 []]],
   N 10
    [N 2 [N 1 []],
     N 5 [N 1 []]],
   N 15
    [N 3 [N 1 []],
     N 5 [N 1 []]]]
```

Definir las funciones

```
arbolDivisores      :: Integer -> Arbol
nOcurrenciasArbolDivisores :: Integer -> Integer -> Integer
```

tales que

- (arbolDivisores x) es el árbol de los divisores del número x. Por ejemplo,

```
λ> arbolDivisores 30
N 30 [N 6  [N 2 [N 1 []],N 3 [N 1 []]],
      N 10 [N 2 [N 1 []],N 5 [N 1 []]],
      N 15 [N 3 [N 1 []],N 5 [N 1 []]]]
```

- (nOcurrenciasArbolDivisores x y) es el número de veces que aparece el número x en el árbol de los divisores del número y. Por ejemplo,

```
nOcurrenciasArbolDivisores 3 30 == 2
nOcurrenciasArbolDivisores 6 30 == 1
nOcurrenciasArbolDivisores 30 30 == 1
```

---

```

n0currenciasArbolDivisores 1 30 == 6
n0currenciasArbolDivisores 9 30 == 0
n0currenciasArbolDivisores 2 (product [1..10]) == 360360
n0currenciasArbolDivisores 3 (product [1..10]) == 180180
n0currenciasArbolDivisores 5 (product [1..10]) == 90090
n0currenciasArbolDivisores 7 (product [1..10]) == 45045
n0currenciasArbolDivisores 6 (product [1..10]) == 102960
n0currenciasArbolDivisores 10 (product [1..10]) == 51480
n0currenciasArbolDivisores 14 (product [1..10]) == 25740

```

---

## Soluciones

```

import Data.Numbers.Primes (primeFactors)
import Data.List (nub)

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

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

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

-- (divisoresPropiosMaximales x) es la lista de los divisores propios
-- maximales de x. Por ejemplo,
--   divisoresPropiosMaximales 30 == [6,10,15]
--   divisoresPropiosMaximales 420 == [60,84,140,210]
--   divisoresPropiosMaximales 7 == [1]
divisoresPropiosMaximales :: Integer -> [Integer]
divisoresPropiosMaximales x =
  reverse [x `div` y | y <- nub (primeFactors x)]

-- Definición de n0currenciasArbolDivisores
-- =====

n0currenciasArbolDivisores :: Integer -> Integer -> Integer

```

```

n0currenciasArbolDivisores x y =
  n0currencias x (arbolDivisores y)

-- (n0currencias x a) es el número de veces que aparece x en el árbol
-- a. Por ejemplo,
--   n0currencias 3 (arbolDivisores 30) == 2
n0currencias :: Integer -> Arbol -> Integer
n0currencias x (N y [])
  | x == y    = 1
  | otherwise = 0
n0currencias x (N y zs)
  | x == y    = 1 + sum [n0currencias x z | z <- zs]
  | otherwise = sum [n0currencias x z | z <- zs]

```

## Ejercicio 32

# Divisores compuestos

*La verdad del hombre empieza  
donde acaba su propia tontería,  
pero la tontería del hombre es  
inagotable.*

---

Antonio Machado

## Enunciado

Definir la función

---

```
divisoresCompuestos :: Integer -> [Integer]
```

---

tal que (divisoresCompuestos x) es la lista de los divisores de x que son números compuestos (es decir, números mayores que 1 que no son primos). Por ejemplo,

---

```
divisoresCompuestos 30 == [6,10,15,30]
length (divisoresCompuestos (product [1..11])) == 534
length (divisoresCompuestos (product [1..14])) == 2585
length (divisoresCompuestos (product [1..16])) == 5369
length (divisoresCompuestos (product [1..25])) == 340022
```

---

## Soluciones

```
import Data.List (group, inits, nub, sort, subsequences)
import Data.Numbers.Primes (isPrime, primeFactors)
import Test.QuickCheck

-- 1ª solución
-- =====

divisoresCompuestos :: Integer -> [Integer]
divisoresCompuestos x =
  [y | y <- divisores x
    , y > 1
    , not (isPrime y)]

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

-- 2ª solución
-- =====

divisoresCompuestos2 :: Integer -> [Integer]
divisoresCompuestos2 x =
  [y | y <- divisores2 x
    , y > 1
    , not (isPrime y)]

-- (divisores2 x) es la lista de los divisores de x. Por ejemplo,
--   divisores2 30 == [1,2,3,5,6,10,15,30]
divisores2 :: Integer -> [Integer]
divisores2 x =
  [y | y <- [1..x `div` 2], x `mod` y == 0] ++ [x]

-- 2ª solución
-- =====
```



```

divisoresCompuestos3 :: Integer -> [Integer]
divisoresCompuestos3 x =
  [y | y <- divisores2 x
    , y > 1
    , not (isPrime y)]

-- (divisores3 x) es la lista de los divisores de x. Por ejemplo,
--   divisores2 30 == [1,2,3,5,6,10,15,30]
divisores3 :: Integer -> [Integer]
divisores3 x =
  nub (sort (ys ++ [x `div` y | y <- ys]))
  where ys = [y | y <- [1..floor (sqrt (fromIntegral x))]
    , x `mod` y == 0]

-- 4ª solución
-- =====

divisoresCompuestos4 :: Integer -> [Integer]
divisoresCompuestos4 x =
  [y | y <- divisores4 x
    , y > 1
    , not (isPrime y)]

-- (divisores4 x) es la lista de los divisores de x. Por ejemplo,
--   divisores4 30 == [1,2,3,5,6,10,15,30]
divisores4 :: Integer -> [Integer]
divisores4 =
  nub . sort . map product . subsequences . primeFactors

-- 5ª solución
-- =====

divisoresCompuestos5 :: Integer -> [Integer]
divisoresCompuestos5 x =
  [y | y <- divisores5 x
    , y > 1
    , not (isPrime y)]

-- (divisores5 x) es la lista de los divisores de x. Por ejemplo,
--   divisores5 30 == [1,2,3,5,6,10,15,30]

```

```

divisores5 :: Integer -> [Integer]
divisores5 =
  sort
    . map (product . concat)
    . productoCartesiano
    . map inits
    . group
    . primeFactors

-- (productoCartesiano xss) es el producto cartesiano de los conjuntos
-- xss. Por ejemplo,
--   λ> productoCartesiano [[1,3],[2,5],[6,4]]
--   [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
productoCartesiano :: [[a]] -> [[a]]
productoCartesiano []      = [[]]
productoCartesiano (xs:xss) =
  [x:ys | x <- xs, ys <- productoCartesiano xss]

-- 6ª solución
-- =====

divisoresCompuestos6 :: Integer -> [Integer]
divisoresCompuestos6 =
  sort
    . map product
    . compuestos
    . map concat
    . productoCartesiano
    . map inits
    . group
    . primeFactors
  where compuestos xss = [xs | xs <- xss, length xs > 1]

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

-- La propiedad es
prop_divisoresCompuestos :: (Positive Integer) -> Bool
prop_divisoresCompuestos (Positive x) =
  all (== divisoresCompuestos x) [f x | f <- [divisoresCompuestos2

```

```

, divisoresCompuestos3
, divisoresCompuestos4
, divisoresCompuestos5
, divisoresCompuestos6 ]]

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

-- Comparación de eficiencia
-- =====

--   λ> length (divisoresCompuestos (product [1..11]))
--   534
--   (14.59 secs, 7,985,108,976 bytes)
--   λ> length (divisoresCompuestos2 (product [1..11]))
--   534
--   (7.36 secs, 3,993,461,168 bytes)
--   λ> length (divisoresCompuestos3 (product [1..11]))
--   534
--   (7.35 secs, 3,993,461,336 bytes)
--   λ> length (divisoresCompuestos4 (product [1..11]))
--   534
--   (0.07 secs, 110,126,392 bytes)
--   λ> length (divisoresCompuestos5 (product [1..11]))
--   534
--   (0.01 secs, 3,332,224 bytes)
--   λ> length (divisoresCompuestos6 (product [1..11]))
--   534
--   (0.01 secs, 1,869,776 bytes)
--
--   λ> length (divisoresCompuestos4 (product [1..14]))
--   2585
--   (9.11 secs, 9,461,570,720 bytes)
--   λ> length (divisoresCompuestos5 (product [1..14]))
--   2585
--   (0.04 secs, 17,139,872 bytes)
--   λ> length (divisoresCompuestos6 (product [1..14]))
--   2585
--   (0.02 secs, 10,140,744 bytes)

```

```
--  
-- λ> length (divisoresCompuestos2 (product [1..16]))  
-- 5369  
-- (1.97 secs, 932,433,176 bytes)  
-- λ> length (divisoresCompuestos5 (product [1..16]))  
-- 5369  
-- (0.03 secs, 37,452,088 bytes)  
-- λ> length (divisoresCompuestos6 (product [1..16]))  
-- 5369  
-- (0.03 secs, 23,017,480 bytes)  
--  
-- λ> length (divisoresCompuestos5 (product [1..25]))  
-- 340022  
-- (2.43 secs, 3,055,140,056 bytes)  
-- λ> length (divisoresCompuestos6 (product [1..25]))  
-- 340022  
-- (1.94 secs, 2,145,440,904 bytes)
```

## Ejercicio 33

# Número de divisores compuestos

*Lo corriente en el hombre es la tendencia a creer verdadero cuanto le reporta alguna utilidad. Por eso hay tantos hombres capaces de comulgar con ruedas de molino.*

---

Antonio Machado

## Enunciado

Definir la función

---

`nDivisoresCompuestos :: Integer -> Integer`

---

tal que (`nDivisoresCompuestos x`) es el número de divisores de `x` que son compuestos (es decir, números mayores que 1 que no son primos). Por ejemplo,

---

```
nDivisoresCompuestos 30 == 4
nDivisoresCompuestos (product [1..11]) == 534
nDivisoresCompuestos (product [1..25]) == 340022
length (show (nDivisoresCompuestos (product [1..3*10^4]))) == 1948
```

---

## Soluciones

```

import Data.List (genericLength, group, inits, sort)
import Data.Numbers.Primes (isPrime, primeFactors)
import Test.QuickCheck

-- 1ª solución
-- =====

nDivisoresCompuestos :: Integer -> Integer
nDivisoresCompuestos =
  genericLength . divisoresCompuestos

-- (divisoresCompuestos x) es la lista de los divisores de x que
-- son números compuestos (es decir, números mayores que 1 que no son
-- primos). Por ejemplo,
--   divisoresCompuestos 30 == [6,10,15,30]
divisoresCompuestos :: Integer -> [Integer]
divisoresCompuestos x =
  [y | y <- divisores x
    , y > 1
    , not (isPrime y)]

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--   divisores 30 == [1,2,3,5,6,10,15,30]
divisores :: Integer -> [Integer]
divisores =
  sort
  . map (product . concat)
  . productoCartesiano
  . map inits
  . group
  . primeFactors

-- (productoCartesiano xss) es el producto cartesiano de los conjuntos xss. Por
-- ejemplo,
--   λ> productoCartesiano [[1,3],[2,5],[6,4]]
--   [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
productoCartesiano :: [[a]] -> [[a]]
productoCartesiano [] = [[]]

```

```

productoCartesiano (xs:xss) =
  [x:ys | x <- xs, ys <- productoCartesiano xss]

-- 2ª solución
-- =====

nDivisoresCompuestos2 :: Integer -> Integer
nDivisoresCompuestos2 x =
  nDivisores x - nDivisoresPrimos x - 1

-- (nDivisores x) es el número de divisores de x. Por ejemplo,
--   nDivisores 30 == 8
nDivisores :: Integer -> Integer
nDivisores x =
  product [1 + genericLength xs | xs <- group (primeFactors x)]

-- (nDivisoresPrimos x) es el número de divisores primos de x. Por
-- ejemplo,
--   nDivisoresPrimos 30 == 3
nDivisoresPrimos :: Integer -> Integer
nDivisoresPrimos =
  genericLength . group . primeFactors

-- 3ª solución
-- =====

nDivisoresCompuestos3 :: Integer -> Integer
nDivisoresCompuestos3 x =
  nDivisores' - nDivisoresPrimos' - 1
  where xss          = group (primeFactors x)
        nDivisores'  = product [1 + genericLength xs | xs <- xss]
        nDivisoresPrimos' = genericLength xss

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

-- La propiedad es
prop_nDivisoresCompuestos :: (Positive Integer) -> Bool
prop_nDivisoresCompuestos (Positive x) =
  all (== nDivisoresCompuestos x) [f x | f <- [ nDivisoresCompuestos2

```

```
, nDivisoresCompuestos3 ]]
```

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

-- Comparación de eficiencia
-- =====

-- λ> nDivisoresCompuestos (product [1..25])
-- 340022
-- (2.53 secs, 3,145,029,032 bytes)
-- λ> nDivisoresCompuestos2 (product [1..25])
-- 340022
-- (0.00 secs, 220,192 bytes)
--
-- λ> length (show (nDivisoresCompuestos2 (product [1..3*10^4])))
-- 1948
-- (5.22 secs, 8,431,630,288 bytes)
-- λ> length (show (nDivisoresCompuestos3 (product [1..3*10^4])))
-- 1948
-- (3.06 secs, 4,662,277,664 bytes)
```



## Ejercicio 34

# Tablas de operaciones binarias

*¿Tu verdad? No, la Verdad,  
y ven conmigo a buscarla.  
La tuya guárdatela.*

---

Antonio Machado

## Enunciado

Para representar las operaciones binarias en un conjunto finito  $A$  con  $n$  elementos se pueden numerar sus elementos desde el 0 al  $n-1$ . Entonces cada operación binaria en  $A$  se puede ver como una lista de listas  $xss$  tal que el valor de aplicar la operación a los elementos  $i$  y  $j$  es el  $j$ -ésimo elemento del  $i$ -ésimo elemento de  $xss$ . Por ejemplo, si  $A = 0,1,2$  entonces las tabla de la suma y de la resta módulo 3 en  $A$  son

---

0 1 2	0 2 1
1 2 0	1 0 2
2 0 1	2 1 0
<b>Suma</b>	<b>Resta</b>

---

Definir las funciones

---

```
tablaOperacion :: (Int -> Int -> Int) -> Int -> [[Int]]
tablaSuma      :: Int -> [[Int]]
```

```

tablaResta      :: Int -> [[Int]]
tablaProducto   :: Int -> [[Int]]

```

---

tales que

- (tablaOperacion f n) es la tabla de la operación f módulo n en [0..n-1]. Por ejemplo,

```

tablaOperacion (+) 3 == [[0,1,2],[1,2,0],[2,0,1]]
tablaOperacion (-) 3 == [[0,2,1],[1,0,2],[2,1,0]]
tablaOperacion (-) 4 == [[0,3,2,1],[1,0,3,2],[2,1,0,3],[3,2,1,0]]
tablaOperacion (\x y -> abs (x-y)) 3 == [[0,1,2],[1,0,1],[2,1,0]]

```

---

- (tablaSuma n) es la tabla de la suma módulo n en [0..n-1]. Por ejemplo,

```

tablaSuma 3 == [[0,1,2],[1,2,0],[2,0,1]]
tablaSuma 4 == [[0,1,2,3],[1,2,3,0],[2,3,0,1],[3,0,1,2]]

```

---

- (tablaResta n) es la tabla de la resta módulo n en [0..n-1]. Por ejemplo,

```

tablaResta 3 == [[0,2,1],[1,0,2],[2,1,0]]
tablaResta 4 == [[0,3,2,1],[1,0,3,2],[2,1,0,3],[3,2,1,0]]

```

---

- (tablaProducto n) es la tabla del producto módulo n en [0..n-1]. Por ejemplo,

```

tablaProducto 3 == [[0,0,0],[0,1,2],[0,2,1]]
tablaProducto 4 == [[0,0,0,0],[0,1,2,3],[0,2,0,2],[0,3,2,1]]

```

---

Comprobar con QuickCheck, si parato entero positivo n de verificar las siguientes propiedades:

- La suma, módulo n, de todos los números de (tablaSuma n) es 0.
- La suma, módulo n, de todos los números de (tablaResta n) es 0.
- La suma, módulo n, de todos los números de (tablaProducto n) es n/2 si n es el doble de un número impar y es 0, en caso contrario.

## Soluciones

```
import Test.QuickCheck
```

```
tablaOperacion :: (Int -> Int -> Int) -> Int -> [[Int]]
tablaOperacion f n =
  [[f i j `mod` n | j <- [0..n-1]] | i <- [0..n-1]]
```

```
tablaSuma :: Int -> [[Int]]
tablaSuma = tablaOperacion (+)
```

```
tablaResta :: Int -> [[Int]]
tablaResta = tablaOperacion (-)
```

```
tablaProducto :: Int -> [[Int]]
tablaProducto = tablaOperacion (*)
```

```
-- (sumaTabla xss) es la suma, módulo n, de los elementos de la tabla de
-- operación xss (donde n es el número de elementos de xss). Por
-- ejemplo,
--     sumaTabla [[0,2,1],[1,1,2],[2,1,0]] == 1
sumaTabla :: [[Int]] -> Int
sumaTabla = sum . concat
```

```
-- La propiedad de la tabla de la suma es
prop_tablaSuma :: Positive Int -> Bool
prop_tablaSuma (Positive n) =
  sumaTabla (tablaSuma n) == 0
```

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

```
-- La propiedad de la tabla de la resta es
prop_tablaResta :: Positive Int -> Bool
prop_tablaResta (Positive n) =
  sumaTabla (tablaResta n) == 0
```

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

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

-- La propiedad de la tabla del producto es
prop_tablaProducto :: Positive Int -> Bool
prop_tablaProducto (Positive n)
  | even n && odd (n `div` 2) = suma == n `div` 2
  | otherwise                = suma == 0
  where suma = sumaTabla (tablaProducto n)
```

## Ejercicio 35

# Reconocimiento de conmutatividad

*Nuestras horas son minutos  
cuando esperamos saber, y  
siglos cuando sabemos lo que se  
puede aprender.*

---

Antonio Machado

## Enunciado

Para representar las operaciones binarias en un conjunto finito  $A$  con  $n$  elementos se pueden numerar sus elementos desde el 0 al  $n-1$ . Entonces cada operación binaria en  $A$  se puede ver como una lista de listas  $xss$  tal que el valor de aplicar la operación a los elementos  $i$  y  $j$  es el  $j$ -ésimo elemento del  $i$ -ésimo elemento de  $xss$ . Por ejemplo, si  $A = 0,1,2$  entonces las tablas de la suma y de la resta módulo 3 en  $A$  son

---

0 1 2	0 2 1
1 2 0	1 0 2
2 0 1	2 1 0
<b>Suma</b>	<b>Resta</b>

---

Definir la función

---

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

---

tal que (conmutativa xss) se verifica si la operación cuya tabla es xss es conmutativa. Por ejemplo,

---

```
conmutativa [[0,1,2],[1,0,1],[2,1,0]] == True
conmutativa [[0,1,2],[1,0,0],[2,1,0]] == False
conmutativa [[i+j `mod` 2000 | j <- [0..1999]] | i <- [0..1999]] == True
conmutativa [[i-j `mod` 2000 | j <- [0..1999]] | i <- [0..1999]] == False
```

---

## Soluciones

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

```
-- 1ª solución
-- =====
```

```
conmutativa :: [[Int]] -> Bool
conmutativa xss =
  and [producto i j == producto j i | i <- [0..n-1], j <- [0..n-1]]
  where producto i j = (xss !! i) !! j
        n            = length xss
```

```
-- 2ª solución
-- =====
```

```
conmutativa2 :: [[Int]] -> Bool
conmutativa2 [] = True
conmutativa2 t@(xs:xss) = xs == map head t
                        && conmutativa2 (map tail xss)
```

```
-- 3ª solución
-- =====
```

```
conmutativa3 :: [[Int]] -> Bool
conmutativa3 xss = xss == transpose xss
```

```
-- 4ª solución
-- =====
```

```

conmutativa4 :: [[Int]] -> Bool
conmutativa4 = (==) <*> transpose

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

-- Para comprobar la equivalencia se define el tipo de tabla de
-- operaciones binarias:
newtype Tabla = T [[Int]]
    deriving Show

-- genTabla es un generador de tablas de operaciones binaria. Por ejemplo,
--     λ> sample genTabla
--     T [[2,0,0],[1,2,1],[1,0,2]]
--     T [[0,3,0,1],[0,1,2,1],[0,2,1,2],[3,0,0,2]]
--     T [[2,0,1],[1,0,0],[2,1,2]]
--     T [[1,0],[0,1]]
--     T [[1,1],[0,1]]
--     T [[1,1,2],[1,0,1],[2,1,0]]
--     T [[4,4,3,0,2],[2,2,0,1,2],[4,0,1,0,0],[0,4,4,3,3],[3,0,4,2,1]]
--     T [[3,4,1,4,1],[2,4,4,0,4],[1,2,1,4,3],[3,1,4,4,2],[4,1,3,2,3]]
--     T [[2,0,1],[2,1,0],[0,2,2]]
--     T [[3,2,0,3],[2,1,1,1],[0,2,1,0],[3,3,2,3]]
--     T [[2,0,2,0],[0,0,3,1],[1,2,3,2],[3,3,0,2]]
genTabla :: Gen Tabla
genTabla = do
    n <- choose (2,20)
    xs <- vectorOf (n^2) (elements [0..n-1])
    return (T (separa n xs))

-- (separa n xs) es la lista obtenidaseparando los elementos de xs en
-- grupos de n elementos. Por ejemplo,
--     separa 3 [1..9] == [[1,2,3],[4,5,6],[7,8,9]]
separa :: Int -> [a] -> [[a]]
separa _ [] = []
separa n xs = take n xs : separa n (drop n xs)

-- Generación arbitraria de tablas
instance Arbitrary Tabla where
    arbitrary = genTabla

```

```

-- La propiedad es
prop_conmutativa :: Tabla -> Bool
prop_conmutativa (T xss) =
  conmutativa xss == conmutativa2 xss &&
  conmutativa2 xss == conmutativa3 xss &&
  conmutativa2 xss == conmutativa4 xss

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

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

-- Para las comparaciones se usará la función tablaSuma tal que
-- (tablaSuma n) es la tabla de la suma módulo n en [0..n-1]. Por
-- ejemplo,
--   tablaSuma 3 == [[0,1,2],[1,2,3],[2,3,4]]
tablaSuma :: Int -> [[Int]]
tablaSuma n =
  [[i + j `mod` n | j <- [0..n-1]] | i <- [0..n-1]]

-- La comparación es
--   λ> conmutativa (tablaSuma 400)
--   True
--   (1.92 secs, 147,608,696 bytes)
--   λ> conmutativa2 (tablaSuma 400)
--   True
--   (0.14 secs, 63,101,112 bytes)
--   λ> conmutativa3 (tablaSuma 400)
--   True
--   (0.10 secs, 64,302,608 bytes)
--   λ> conmutativa4 (tablaSuma 400)
--   True
--   (0.10 secs, 61,738,928 bytes)
--
--   λ> conmutativa2 (tablaSuma 2000)
--   True
--   (1.81 secs, 1,569,390,480 bytes)

```



---

```
-- λ> conmutativa3 (tablaSuma 2000)
-- True
-- (3.07 secs, 1,601,006,840 bytes)
-- λ> conmutativa4 (tablaSuma 2000)
-- True
-- (3.14 secs, 1,536,971,288 bytes)
```



## Ejercicio 36

# Árbol de subconjuntos

*Nunca traces tu frontera,  
ni cuides de tu perfil;  
todo eso es cosa de fuera.*

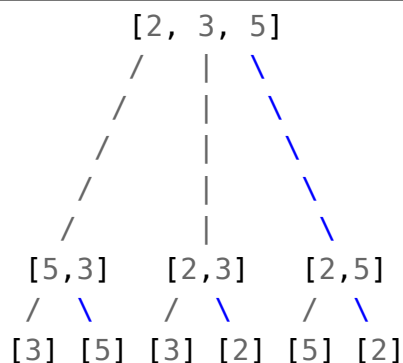
---

Antonio Machado

## Enunciado

Se dice que  $A$  es un subconjunto maximal de  $B$  si  $A \subset B$  y no existe ningún  $C$  tal que  $A \subset C$  y  $C \subset B$ . Por ejemplo,  $2,5$  es un subconjunto maximal de  $2,3,5$ , pero  $3$  no lo es.

El árbol de los subconjuntos de un conjunto  $A$  es el árbol que tiene como raíz el conjunto  $A$  y cada nodo tiene como hijos sus subconjuntos maximales. Por ejemplo, el árbol de subconjuntos de  $2,3,5$  es



```

      |   |   |   |   |
    [ ] [ ] [ ] [ ] [ ] [ ]

```

Usando el tipo de dato

```

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

```

el árbol anterior se representa por

```

N [2,5,3]
  [N [5,3]
    [N [3]
      [N [] []],
      N [5]
        [N [] []]],
    N [2,3]
      [N [3]
        [N [] []],
        N [2]
          [N [] []]],
    N [2,5]
      [N [5]
        [N [] []],
        N [2]
          [N [] []]]]

```

Definir las funciones

```

arbolSubconjuntos :: [Int] -> Arbol
n0currenciasArbolSubconjuntos :: [Int] -> [Int] -> Int

```

tales que

- (arbolSubconjuntos x) es el árbol de los subconjuntos de xs. Por ejemplo,

```

λ> arbolSubconjuntos [2,5,3]
N [2,5,3] [N [5,3] [N [3] [N [] []],N [5] [N [] []]],
          N [2,3] [N [3] [N [] []],N [2] [N [] []]],
          N [2,5] [N [5] [N [] []],N [2] [N [] []]]]

```

- `(nOcurrenciasArbolSubconjuntos xs ys)` es el número de veces que aparece el conjunto `xs` en el árbol de los subconjuntos de `ys`. Por ejemplo,

---

<code>nOcurrenciasArbolSubconjuntos []</code>	<code>[2,5,3]</code>	<code>==</code>	<code>6</code>
<code>nOcurrenciasArbolSubconjuntos [3]</code>	<code>[2,5,3]</code>	<code>==</code>	<code>2</code>
<code>nOcurrenciasArbolSubconjuntos [3,5]</code>	<code>[2,5,3]</code>	<code>==</code>	<code>1</code>
<code>nOcurrenciasArbolSubconjuntos [3,5,2]</code>	<code>[2,5,3]</code>	<code>==</code>	<code>1</code>

---

Comprobar con QuickChek que, para todo entero positivo  $n$ , el número de ocurrencia de un subconjunto `xs` de `[1..n]` en el árbol de los subconjuntos de `[1..n]` es el factorial de  $n-k$  (donde  $k$  es el número de elementos de `xs`).

## Soluciones

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

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

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

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

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

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

```

-- (n0currencias x a) es el número de veces que aparece x en el árbol
-- a. Por ejemplo,
--     n0currencias 3 (arbolSubconjuntos 30) == 2
n0currencias :: [Int] -> Arbol -> Int
n0currencias xs (N ys [])
  | conjunto xs == conjunto ys = 1
  | otherwise                  = 0
n0currencias xs (N ys zs)
  | conjunto xs == conjunto ys = 1 + sum [n0currencias xs z | z <- zs]
  | otherwise                  = sum [n0currencias xs z | z <- zs]

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

-- La propiedad es
prop_n0currencias :: (Positive Int) -> [Int] -> Bool
prop_n0currencias (Positive n) xs =
  n0currenciasArbolSubconjuntos ys [1..n] == factorial (n-k)
  where ys = nub [1 + x `mod` n | x <- xs]
        k  = length ys
        factorial m = product [1..m]

-- La comprobación es
--     λ> quickCheckWith (stdArgs {maxSize=9}) prop_n0currencias
--     +++ OK, passed 100 tests.

```

## Ejercicio 37

# El teorema de Navidad de Fermat

- ¡Cuándo llegará otro día!  
- Hoy es siempre todavía.

---

Antonio Machado

## Enunciado

El 25 de diciembre de 1640, en una carta a Mersenne, Fermat demostró la conjetura de Girard: todo primo de la forma  $4n+1$  puede expresarse de manera única como suma de dos cuadrados. Por eso es conocido como el Teorema de Navidad de Fermat

Definir las funciones

---

representaciones	:: Integer -> [(Integer,Integer)]
primosImparesConRepresentacionUnica	:: [Integer]
primos4nM1	:: [Integer]

---

tales que

- (representaciones n) es la lista de pares de números naturales (x,y) tales que  $n = x^2 + y^2$  con  $x \leq y$ . Por ejemplo.

---

representaciones	20	==	[(2,4)]
representaciones	25	==	[(0,5),(3,4)]

```
representaciones 325 == [(1,18),(6,17),(10,15)]
representaciones 100000147984 == [(0,316228)]
length (representaciones (10^10)) == 6
length (representaciones (4*10^12)) == 7
```

---

- `primosImparesConRepresentacionUnica` es la lista de los números primos impares que se pueden escribir exactamente de una manera como suma de cuadrados de pares de números naturales  $(x,y)$  con  $x \leq y$ . Por ejemplo,

```
λ> take 20 primosImparesConRepresentacionUnica
[5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
```

---

- `primos4nM1` es la lista de los números primos que se pueden escribir como uno más un múltiplo de 4 (es decir, que son congruentes con 1 módulo 4). Por ejemplo,

```
λ> take 20 primos4nM1
[5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
```

---

El [teorema de Navidad de Fermat](#) que afirma que un número primo impar  $p$  se puede escribir exactamente de una manera como suma de dos cuadrados de números naturales  $p = x^2 + y^2$  (con  $x \leq y$ ) si, y sólo si,  $p$  se puede escribir como uno más un múltiplo de 4 (es decir, que son congruentes con 1 módulo 4).

Comprobar con QuickCheck el teorema de Navidad de Fermat; es decir, que para todo número  $n$ , los  $n$ -ésimos elementos de `primosImparesConRepresentacionUnica` y de `primos4nM1` son iguales.

## Soluciones

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

-- 1ª definición de representaciones
-- =====

representaciones :: Integer -> [(Integer,Integer)]
representaciones n =
```



```

[(x,y) | x <- [0..n], y <- [x..n], n == x*x + y*y]

-- 2ª definición de representaciones
-- =====

representaciones2 :: Integer -> [(Integer,Integer)]
representaciones2 n =
  [(x,raiz z) | x <- [0..raiz (n `div` 2)]
               , let z = n - x*x
               , esCuadrado z]

-- (esCuadrado x) se verifica si x es un número al cuadrado. Por
-- ejemplo,
--     esCuadrado 25 == True
--     esCuadrado 26 == False
esCuadrado :: Integer -> Bool
esCuadrado x = x == y * y
  where y = raiz x

-- (raiz x) es la raíz cuadrada entera de x. Por ejemplo,
--     raiz 25 == 5
--     raiz 24 == 4
--     raiz 26 == 5
raiz :: Integer -> Integer
raiz 0 = 0
raiz 1 = 1
raiz x = aux (0,x)
  where aux (a,b) | d == x      = c
                  | c == a      = a
                  | d < x       = aux (c,b)
                  | otherwise   = aux (a,c)
                  where c = (a+b) `div` 2
                        d = c^2

-- 3ª definición de representaciones
-- =====

representaciones3 :: Integer -> [(Integer,Integer)]
representaciones3 n =
  [(x,raiz3 z) | x <- [0..raiz3 (n `div` 2)]

```

```

    , let z = n - x*x
    , esCuadrado3 z]

-- (esCuadrado x) se verifica si x es un número al cuadrado. Por
-- ejemplo,
--     esCuadrado3 25 == True
--     esCuadrado3 26 == False
esCuadrado3 :: Integer -> Bool
esCuadrado3 x = x == y * y
    where y = raiz3 x

-- (raiz3 x) es la raíz cuadrada entera de x. Por ejemplo,
--     raiz3 25 == 5
--     raiz3 24 == 4
--     raiz3 26 == 5
raiz3 :: Integer -> Integer
raiz3 x = floor (sqrt (fromIntegral x))

-- 4ª definición de representaciones
-- =====

representaciones4 :: Integer -> [(Integer, Integer)]
representaciones4 n = aux 0 (floor (sqrt (fromIntegral n)))
    where aux x y
          | x > y      = []
          | otherwise = case compare (x*x + y*y) n of
              LT -> aux (x + 1) y
              EQ -> (x, y) : aux (x + 1) (y - 1)
              GT -> aux x (y - 1)

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

-- La propiedad es
prop_representaciones_equiv :: (Positive Integer) -> Bool
prop_representaciones_equiv (Positive n) =
    representaciones n == representaciones2 n &&
    representaciones2 n == representaciones3 n &&
    representaciones3 n == representaciones4 n

```

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

-- Comparación de eficiencia de las definiciones de representaciones
-- =====

--   λ> representaciones 3025
--   [(0,55),(33,44)]
--   (2.86 secs, 1,393,133,528 bytes)
--   λ> representaciones2 3025
--   [(0,55),(33,44)]
--   (0.00 secs, 867,944 bytes)
--   λ> representaciones3 3025
--   [(0,55),(33,44)]
--   (0.00 secs, 173,512 bytes)
--   λ> representaciones4 3025
--   [(0,55),(33,44)]
--   (0.00 secs, 423,424 bytes)
--
--   λ> length (representaciones2 (10^10))
--   6
--   (3.38 secs, 2,188,903,544 bytes)
--   λ> length (representaciones3 (10^10))
--   6
--   (0.10 secs, 62,349,048 bytes)
--   λ> length (representaciones4 (10^10))
--   6
--   (0.11 secs, 48,052,360 bytes)
--
--   λ> length (representaciones3 (4*10^12))
--   7
--   (1.85 secs, 1,222,007,176 bytes)
--   λ> length (representaciones4 (4*10^12))
--   7
--   (1.79 secs, 953,497,480 bytes)

-- Definición de primosImparesConRepresentacionUnica
-- =====
```

```

primosImparesConRepresentacionUnica :: [Integer]
primosImparesConRepresentacionUnica =
  [x | x <- tail primes
    , length (representaciones4 x) == 1]

-- Definición de primos4nM1
-- =====

primos4nM1 :: [Integer]
primos4nM1 = [x | x <- primes
  , x `mod` 4 == 1]

-- Teorema de Navidad de Fermat
-- =====

-- La propiedad es
prop_teoremaDeNavidadDeFermat :: Positive Int -> Bool
prop_teoremaDeNavidadDeFermat (Positive n) =
  primosImparesConRepresentacionUnica !! n == primos4nM1 !! n

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

```

## Ejercicio 38

# El 2019 es apocalíptico

*A vosotros no os importe pensar lo que habéis leído ochenta veces y oído quinientas, porque no es lo mismo pensar que haber leído.*

Antonio Machado

## Enunciado

Un número natural  $n$  es [apocalíptico](http://bit.ly/2RqeeNk) si  $2^n$  contiene la secuencia 666. Por ejemplo, 157 es apocalíptico porque  $2^{157}$  es

182687704666362864775460604089535377456991567872

que contiene la secuencia 666.

Definir las funciones

---

```
esApocaliptico      :: Integer -> Bool
apocalipticos       :: [Integer]
posicionApocaliptica :: Integer -> Maybe Int
```

---

tales que

- (esApocaliptico  $n$ ) se verifica si  $n$  es un número apocalíptico. Por ejemplo,

---

```
esApocaliptico 157 == True
esApocaliptico 2019 == True
esApocaliptico 2018 == False
```

---

- apocalipticos es la lista de los números apocalípticos. Por ejemplo,

---

```
take 9 apocalipticos == [157,192,218,220,222,224,226,243,245]
apocalipticos !! 450 == 2019
```

---

- (posicionApocalitica n) es justo la posición de n en la sucesión de números apocalípticos, si n es apocalíptico o Nothing, en caso contrario. Por ejemplo,

---

```
posicionApocalitica 157 == Just 0
posicionApocalitica 2019 == Just 450
posicionApocalitica 2018 == Nothing
```

---

## Soluciones

```
import Data.List (isInfixOf, elemIndex)

-- 1ª definición de esApocaliptico
esApocaliptico :: Integer -> Bool
esApocaliptico n = "666" `isInfixOf` show (2^n)

-- 2ª definición de esApocaliptico
esApocaliptico2 :: Integer -> Bool
esApocaliptico2 = isInfixOf "666" . show . (2^)

-- 1ª definición de apocalipticos
apocalipticos :: [Integer]
apocalipticos = [n | n <- [1..], esApocaliptico n]

-- 2ª definición de apocalipticos
apocalipticos2 :: [Integer]
apocalipticos2 = filter esApocaliptico [1..]

-- 1ª definición de posicionApocalitica
```

```
posicionApocaliptica :: Integer -> Maybe Int
posicionApocaliptica n
  | y == n    = Just (length xs)
  | otherwise = Nothing
  where (xs,y:_) = span (<n) apocalipticos

-- 2ª definición de posicionApocaliptica
posicionApocaliptica2 :: Integer -> Maybe Int
posicionApocaliptica2 n
  | esApocaliptico n = elemIndex n apocalipticos
  | otherwise        = Nothing
```