Exercitium (curso 2018–19)

Ejercicios de programación funcional con Haskell

(hasta el 8 de marzo de 2019)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 18 de marzo de 2019

“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

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 [1] durante el curso 2018–19.

El principal objetivo de Exercitium es servir de complemento a la asignatura de Informática [2] de 1º del Grado en Matemáticas de la Universidad de Sevilla.

Con los problemas de Exercitium, a diferencias de los de las relaciones [3], 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 [4].

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

[1] https://www.glc.us.es/~jalonso/exercitium

[2] https://www.cs.us.es/~jalonso/cursos/i1m-18

[3] https://www.cs.us.es/~jalonso/cursos/i1m-18/ejercicios/ejercicios-I1M-2018.pdf

[4] https://www.glc.us.es/~jalonso/vestigium/category/curso/i1m/i1m2018

El código del libro se encuentra en GitHub [5]

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

[5] 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

Versión del 12 de enero de 2019

Se han añadido los ejercicios resueltos del 31 de diciembre al 4 de enero:

- El teorema de Navidad de Fermat
- El 2019 es apocalíptico
- El 2019 es malvado
- El 2019 es semiprimo
- El 2019 es un número de la suerte

Versión del 19 de enero de 2019

Se han añadido los ejercicios resueltos hasta el 11 de enero:

- Cadena descendiente de subnúmeros
- Mínimo producto escalar
- Numeración de ternas de naturales
- Subárboles monovalorados
- Mayor prefijo con suma acotada

Versión del 26 de enero de 2019

Se han añadido los ejercicios resueltos hasta del 14 al 18 de enero:

- Ofertas 3 por 2
- Representación de conjuntos mediante intervalos
- Números altamente compuestos
- Posiciones del 2019 en el número pi
- Mínimo número de operaciones para transformar un número en otro

Versión del 2 de febrero de 2019

Se han añadido los ejercicios resueltos hasta del 21 al 25 de enero:

- Intersección de listas infinitas crecientes
- Soluciones de $x^2 = y^3 = k$
- Sucesión triangular
- Números primos en pi
- Recorrido de árboles en espiral

Versión del 9 de febrero de 2019

Se han añadido los ejercicios resueltos hasta del 28 de enero al 1 de febrero:

- Números con dígitos 1 y 2
- Árboles con n elementos
- Impares en filas del triángulo de Pascal
- Triángulo de Pascal binario
- Dígitos en las posiciones pares de cuadrados

Versión del 16 de febrero de 2019

Se han añadido los ejercicios resueltos hasta del 4 al 8 de febrero:

- Límites de sucesiones
- Medias de dígitos de pi
- Exterior de árboles
- Aritmética lunar
- Término ausente en una progresión aritmética

Versión del 23 de febrero de 2019

Se han añadido los ejercicios resueltos hasta del 11 al 15 de febrero:

- Particiones de enteros positivos
- Sucesión de sumas de dos números abundantes
- Cálculo de pi mediante la serie de Nilakantha
- Simplificación de expresiones booleanas
- Sucesión de Cantor de números innombrables

Versión del 2 de marzo de 2019

Se han añadido los ejercicios resueltos hasta del 18 al 22 de febrero:

- Ternas euclídeas
- Mezcla de listas
- Mayor exponente
- Número de sumandos en suma de cuadrados
- Divisiones del círculo

Versión del 9 de marzo de 2019

Se han añadido los ejercicios resueltos hasta del 25 de febrero al 1 de marzo:

- Inserciones en una lista de listas
- Particiones de un conjunto
- Número de particiones de un conjunto
- Descomposiciones en sumas de cuadrados
- Número de descomposiciones en sumas de cuadrados

Versión del 16 de marzo de 2019

Se han añadido los ejercicios resueltos del 4 al 8 de marzo:

- Hojas con caminos no decrecientes
- Las torres de Hanói
- Número como suma de sus dígitos
- Suma de segmentos iniciales
- Camino de máxima suma en una matriz

Ejercicio 1

Listas equidigitales

Se miente más de la cuenta
por falta de fantasía:
también la verdad se inventa.

Antonio Machado

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

```haskell
-- 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

En mi soledad
he visto cosas muy claras,
que no son verdad.

Antonio Machado

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 ”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: 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 distancia(xs,ys) = 0.

Soluciones

```haskell
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_distancia1 :: [Int] -> [Int] -> Bool
prop_distancia1 xs ys =
  (distancia xs ys == 0) == (xs == ys)

-- La comprobación es
--    ghci> quickCheck prop_distancia1
--    *** 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:
```

```haskell
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

> *Incierto es, lo porvenir. ¿Quién sabe lo que va a pasar? Pero incierto es también lo pretérito. ¿Quién sabe lo que ha pasado? De suerte que ni el porvenir está escrito en ninguna parte, ni el pasado tampoco.*
>
> Antonio Machado

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

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

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

```haskell
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

```haskell
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,
--    ultimoNoNulo 5040  ==  4
ultimoNoNulo2 :: Integer -> Integer
```

```haskell
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

Ya es broma pesada:
todo para mí,
y yo para nada.

Antonio Machado

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

```haskell
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

Algunos sentimientos perduran a través de los siglos, pero no por eso han de ser eternos. ¿Cuántos siglos durará todavía el sentimiento de la patria? ¿Y el sentimiento de la paternidad.

Antonio Machado

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 (((10^10)^10)^10)  ==  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 =
```

```haskell
  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

Ayudadme a comprender lo que os digo, y os lo explicaré más despacio.

Antonio Machado

Enunciado

El número 9009 es capicúa y es producto de dos números de dos dígitos, pues 9009 = 91x99.

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

Hay que tener los ojos muy abiertos para ver las cosas como son; aún más abiertos para verlas otras de lo que son; más abiertos todavía para verlas mejores de lo que son.

Antonio Machado

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 ”2z 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

```haskell
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

Toda la imaginería
que no ha brotado del río,
barata bisutería.

Antonio Machado

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

```haskell
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]
```

```haskell
--    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)

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

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

Ejercicio 9

Reconocimiento de particiones

Sentía los cuatro vientos,
en la encrucijada
de su pensamiento.

Antonio Machado

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

```haskell
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]]
```

```haskell
-- 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

No hay lío político que no sea un trueque, una confusión de máscaras, un mal ensayo de comedia, en que nadie sabe su papel.

Antonio Machado

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

El escepticismo que, lejos de ser, como muchos creen, un afán de negarlo todo es, por el contrario, el único medio de defender algunas cosas.

Antonio Machado

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)
```

```haskell
-- 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

Sed incompresivos; yo os aconsejo la incomprensión, aunque sólo sea para destripar los chistes de los tontos.

Antonio Machado

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

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

```haskell
-- 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)
```

Ejercicio 13

Suma de inversos de potencias de cuatro

Confiamos
en que no será verdad
nada de lo que pensamos.

Antonio Machado

Enunciado

Esta semana se ha publicado en Twitter una demostración visual de que

$$1/4 + 1/4^2 + 1/4^3 + \cdots = 1/3$$

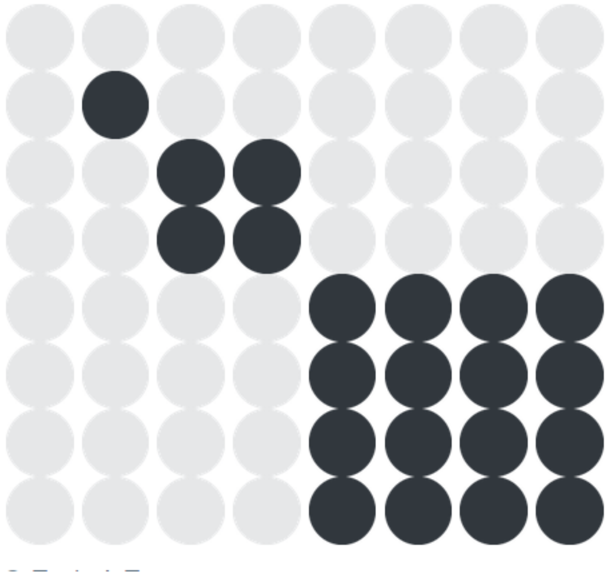
como se muestra en la siguiente imagen

Definir las funciones

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

tales que

- sumaInversosPotenciasDeCuatro es la lista de las 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]
```

```haskell
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

Sube y sube, pero ten
cuidado Nefelibata,
que entre las nubes también,
se puede meter la pata.

Antonio Machado

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

```haskell
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

Si me tengo que morir
poco me importa aprender.
Y si no puedo saber,
poco me importa vivir.

Antonio Machado

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

```haskell
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
```

```haskell
--    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

Tras el vivir y el soñar,
está lo que más importa:
despertar.

Antonio Machado

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

– Ya se oyen palabras viejas.
– Pues aguzad las orejas.

Antonio Machado

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,

```
          1
         / \
        /   \
       /     \
      2       3
     / \     / \
    4   5   6   7
   / \
  8   9
```

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

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

Antonio Machado

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

```
         3
        / \
       /   \
      0     5
     / \   / \
    5   0 0   3
   / \
  2   4
```

se representa por

```haskell
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 al árbol anterior es [I,I,D].

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

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

Definir la función

```haskell
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

```haskell
import Data.List (nub)
import Test.QuickCheck
```

```haskell
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
```

```haskell
          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

El ojo que ves no es
ojo porque tú lo veas;
es ojo porque te ve.

Antonio Machado

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,

```
          1
         / \
        /   \
       /     \
      2       3
     / \     / \
    4   5   6   7
```

```
        / \
       8   9
```

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 al 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
```

```haskell
  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
```

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

-- 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

Las más hondas palabras
del sabio nos enseñan
lo que el silbar del viento cuando sopla
o el sonar de las aguas cuando ruedan.

Antonio Machado

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,

```
        1
       / \
      /   \
     /     \
    2       3
   / \     / \
  4   5   6   7
```

```
         / \
        8   9
```

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 al 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)
```

```haskell
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

> *"Sólo sé que no se nada" contenía la jactancia de un excesivo saber, puesto que olvidó añadir: y aun de esto mismo no estoy completamente seguro.*
>
> Antonio Machado

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:

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

Definir las siguientes funciones:

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

tales que

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

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

  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

```haskell
-- 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

¡Ya hay hombres activos!
Soñaba la charca
con sus mosquitos.

Antonio Machado

Enunciado

El n–ésimo menor contenenedor 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 contenenedor de primos. Por ejemplo,

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

Soluciones

```haskell
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

Toda visión requiere distancia.

Antonio Machado

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,...
```

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

El árbol de computación de su 5º término es

```
               5
              / \
             /   \
            /     \
           /       \
          /         \
         3           2
        / \         / \
```

```
       /   \       1   1
      2     1     / \
     / \   / \   1   0
    1   1 1   0
   / \
  1   0
```

que, usando los árboles definidos por

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

se puede representar por

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

Definir 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

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

-- 1ª definición
-- =============

arbolFib :: Int -> Arbol
arbolFib 0 = H 0
```

```haskell
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
```

```haskell
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

Las razones no se transmiten, se engendran, por cooperación, en el diálogo.

Antonio Machado

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
```

```haskell
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
```

```haskell
-- 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 ListaNoVaciaDePositivos = L [Int]
  deriving Show

-- genListaNoVaciaDePositivos es un generador de listas no vacióas de
-- enteros positivos. Por ejemplo,
--    λ> sample genListaNoVaciaDePositivos
--    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]
genListaNoVaciaDePositivos :: Gen ListaNoVaciaDePositivos
genListaNoVaciaDePositivos = do
  x  <- arbitrary
  xs <- arbitrary
  return (L (map ((+1) . abs) (x:xs)))

-- Generación arbitraria de listas no vacías de enteros positivos.
instance Arbitrary ListaNoVaciaDePositivos where
  arbitrary = genListaNoVaciaDePositivos
```

```haskell
-- La propiedad es
prop_entreDosConjuntos_equiv ::
     ListaNoVaciaDePositivos
  -> ListaNoVaciaDePositivos
  -> 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

Vivir es devorar tiempo, esperar; y por muy trascendente que quiera ser nuestra espera, siempre será espera de seguir esperando.

Antonio Machado

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

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

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

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

```haskell
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

```haskell
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 necedad, 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 nos 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

```haskell
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
```

```haskell
  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]
```

```haskell
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
```

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

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

```
          30
          /|\
         / | \
        /  |  \
       /   |   \
      /    |    \
     6    10    15
```

```
  / \    / \    / \
 2   3  2   5  3   5
```

Usando el tipo de dato

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

el árbol anterior se representa por

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

Definir las funciones

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

tales que

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

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

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

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

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

Soluciones

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

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

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

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

```haskell
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]
```

```haskell
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
```

```haskell
                                                    , 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

```haskell
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,

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

Soluciones

```haskell
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
Suma     Resta
```

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

```haskell
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 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
Suma     Resta
```

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

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

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

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

-- genTabla es un generador de tablas de operciones 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
```

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

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

```
          [2, 3, 5]
          /   |  \
         /    |   \
        /     |    \
       /      |     \
      /       |      \
    [5,3]   [2,3]   [2,5]
    /  \    /  \    /  \
  [3] [5] [3] [2] [5] [2]
```

```
  |    |    |    |    |    |
 [ ]  [ ]  [ ]  [ ]  [ ]  [ ]
```

Usando el tipo de dato

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

el árbol anterior se representa por

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

Definir las funciones

```
arbolSubconjuntos :: [Int] -> Arbol
nOcurrenciasArbolSubconjuntos :: [Int] -> [Int] -> Int
```

tales que

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

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

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

```
nOcurrenciasArbolSubconjuntos []      [2,5,3]  ==  6
nOcurrenciasArbolSubconjuntos [3]     [2,5,3]  ==  2
nOcurrenciasArbolSubconjuntos [3,5]   [2,5,3]  ==  1
nOcurrenciasArbolSubconjuntos [3,5,2] [2,5,3]  ==  1
```

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)
```

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

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

-- La propiedad es
prop_nOcurrencias :: (Positive Int) -> [Int] -> Bool
prop_nOcurrencias (Positive n) xs =
  nOcurrenciasArbolSubconjuntos 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_nOcurrencias
--    +++ 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 torema 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
-- =================================
```

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

```haskell
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 &&
```

```haskell
  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,

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

Soluciones

```haskell
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 posicionApocaliptica
```

```haskell
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
```

Ejercicio 39

El 2019 es malvado

...Yo os enseño, o pretendo enseñaros a que dudéis de todo: de lo humano y de lo divino, sin excluir vuestra propia existencia.

Antonio Machado

Enunciado

Un número malvado es un número natural cuya expresión en base 2 contiene un número par de unos. Por ejemplo, 6 es malvado porque su expresión en base 2 es 110 que tiene dos unos.

Definir las funciones

```
esMalvado        :: Integer -> Bool
malvados         :: [Integer]
posicionMalvada  :: Integer -> Maybe Int
```

tales que

- (esMalvado n) se verifica si n es un número malvado. Por ejemplo,

```
esMalvado 6              ==  True
esMalvado 7              ==  False
esMalvado 2019           ==  True
```

```
esMalvado (10^70000)     ==  True
esMalvado (10^(3*10^7))  ==  True
```

- malvados es la sucesión de los números malvados. Por ejemplo,

```
λ> take 20 malvados
[0,3,5,6,9,10,12,15,17,18,20,23,24,27,29,30,33,34,36,39]
malvados !! 1009    ==  2019
malvados !! 10      ==  20
malvados !! (10^2)  ==  201
malvados !! (10^3)  ==  2000
malvados !! (10^4)  ==  20001
malvados !! (10^5)  ==  200000
malvados !! (10^6)  ==  2000001
```

- (posicionMalvada n) es justo la posición de n en la sucesión de números malvados, si n es malvado o Nothing, en caso contrario. Por ejemplo,

```
posicionMalvada 6        ==  Just 3
posicionMalvada 2019     ==  Just 1009
posicionMalvada 2018     ==  Nothing
posicionMalvada 2000001  ==  Just 1000000
posicionMalvada (10^7)   ==  Just 5000000
```

Soluciones

```haskell
import Data.List (genericLength, elemIndex)
import Data.Bits (popCount)

-- 1ª definición de esMalvado
-- ==========================

esMalvado :: Integer -> Bool
esMalvado n = even (numeroUnosBin n)

-- Sin argumentos
esMalvado' :: Integer -> Bool
esMalvado' = even . numeroUnosBin
```

```haskell
-- (numeroUnosBin n) es el número de unos de la representación binaria
-- del número decimal n. Por ejemplo,
--    numeroUnosBin 11  ==  3
--    numeroUnosBin 12  ==  2
numeroUnosBin :: Integer -> Integer
numeroUnosBin n  = genericLength (filter (== 1) (binario n))

-- Sin argumentos
numeroUnosBin' :: Integer -> Integer
numeroUnosBin' = genericLength . filter (== 1) . binario

-- (binario n) es el número binario correspondiente al número decimal n.
-- Por ejemplo,
--    binario 11  ==  [1,1,0,1]
--    binario 12  ==  [0,0,1,1]
binario :: Integer -> [Integer]
binario n | n < 2     = [n]
          | otherwise = n `mod` 2 : binario (n `div` 2)

-- 2ª definición de esMalvado
-- ==========================

esMalvado2 :: Integer -> Bool
esMalvado2 n = even (numeroUnosBin n)

-- (numeroIntBin n) es el número de unos que contiene la representación
-- binaria del número decimal n. Por ejemplo,
--    numeroIntBin 11  ==  3
--    numeroIntBin 12  ==  2
numeroIntBin :: Integer -> Integer
numeroIntBin n | n < 2     = n
               | otherwise = n `mod` 2 + numeroIntBin (n `div` 2)

-- 3ª definición de esMalvado
-- ==========================

esMalvado3 :: Integer -> Bool
esMalvado3 n = even (popCount n)

-- Sin argumentos
```

```haskell
esMalvado3' :: Integer -> Bool
esMalvado3' = even . popCount

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

--    λ> esMalvado (10^30000)
--    True
--    (1.79 secs, 664,627,936 bytes)
--    λ> esMalvado2 (10^30000)
--    True
--    (1.79 secs, 664,626,992 bytes)
--    λ> esMalvado3 (10^30000)
--    True
--    (0.03 secs, 141,432 bytes)
--
--    λ> esMalvado (10^40000)
--    False
--    (2.95 secs, 1,162,091,464 bytes)
--    λ> esMalvado2 (10^40000)
--    False
--    (2.96 secs, 1,162,091,096 bytes)
--    λ> esMalvado3 (10^40000)
--    False
--    (0.04 secs, 155,248 bytes)

-- 1ª definición de malvados
-- =========================

malvados :: [Integer]
malvados = [n | n <- [0..], esMalvado3 n]

-- 2ª definición de malvados
-- =========================

malvados2 :: [Integer]
malvados2 = filter esMalvado3 [0..]

-- 1ª definición de posicionMalvada
-- ================================
```

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

-- 2ª definición de posicionMalvada
posicionMalvada2 :: Integer -> Maybe Int
posicionMalvada2 n
  | esMalvado n = elemIndex n malvados
  | otherwise        = Nothing
```

Ejercicio 40

El 2019 es semiprimo

Porque toda visión requiere distancia, no hay manera de ver las cosas sin salirse de ellas.

Antonio Machado

Enunciado

Un número semiprimo es un número natural que es producto de dos números primos no necesariamente distintos. Por ejemplo, 26 es semiprimo (porque 26 = 2*13) y 49 también lo es (porque 49 = 7*7).

Definir las funciones

```
esSemiprimo :: Integer -> Bool
semiprimos  :: [Integer]
```

tales que

- (esSemiprimo n) se verifica si n es semiprimo. Por ejemplo,

```
esSemiprimo 26          ==  True
esSemiprimo 49          ==  True
esSemiprimo 8           ==  False
esSemiprimo 2019        ==  True
esSemiprimo (21+10^14)  ==  True
```

- semiprimos es la sucesión de números semiprimos. Por ejemplo,

```
take 10 semiprimos   ==  [4,6,9,10,14,15,21,22,25,26]
semiprimos !! 579    ==  2019
semiprimos !! 10000  ==  40886
```

Soluciones

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

-- 1ª definición de esSemiprimo
-- ============================

esSemiprimo :: Integer -> Bool
esSemiprimo n =
  not (null [x | x <- [n,n-1..2],
                 primo x,
                 n `mod` x == 0,
                 primo (n `div` x)])

primo :: Integer -> Bool
primo n = [x | x <- [1..n], n `mod` x == 0] == [1,n]

-- 2ª definición de esSemiprimo
-- ============================

esSemiprimo2 :: Integer -> Bool
esSemiprimo2 n =
  not (null [x | x <- [n-1,n-2..2],
                 isPrime x,
                 n `mod` x == 0,
                 isPrime (n `div` x)])

-- 3ª definición de esSemiprimo
-- ============================

esSemiprimo3 :: Integer -> Bool
esSemiprimo3 n =
```

```haskell
  not (null [x | x <- reverse (takeWhile (<n) primes),
                 n `mod` x == 0,
                 isPrime (n `div` x)])

-- 4ª definición de esSemiprimo
-- ============================

esSemiprimo4 :: Integer -> Bool
esSemiprimo4 n =
  length (primeFactors n) == 2

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

-- La propiedad es
prop_esSemiprimo :: Positive Integer -> Bool
prop_esSemiprimo (Positive n) =
  all (== esSemiprimo n) [f n | f <- [ esSemiprimo2
                                     , esSemiprimo3
                                     , esSemiprimo4
                                     ]]

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

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

--    λ> esSemiprimo 5001
--    True
--    (1.90 secs, 274,450,648 bytes)
--    λ> esSemiprimo2 5001
--    True
--    (0.07 secs, 29,377,016 bytes)
--    λ> esSemiprimo3 5001
--    True
--    (0.01 secs, 1,706,840 bytes)
--    λ> esSemiprimo4 5001
--    True
```

```haskell
--    (0.01 secs, 142,840 bytes)
--
--    λ> esSemiprimo2 100001
--    True
--    (2.74 secs, 1,473,519,064 bytes)
--    λ> esSemiprimo3 100001
--    True
--    (0.09 secs, 30,650,352 bytes)
--    λ> esSemiprimo4 100001
--    True
--    (0.01 secs, 155,200 bytes)
--
--    λ> esSemiprimo3 10000001
--    True
--    (8.73 secs, 4,357,875,016 bytes)
--    λ> esSemiprimo4 10000001
--    True
--    (0.01 secs, 456,328 bytes)

-- Definición de semiprimos
-- ========================

semiprimos :: [Integer]
semiprimos = filter esSemiprimo4 [4..]
```

Ejercicio 41

El 2019 es un número de la suerte

Ya es sólo brocal el pozo;
púlpito será mañana;
pasado mañana, trono.

Antonio Machado

Enunciado

Un número de la suerte es un número natural que se genera por una criba, similar a la criba de Eratóstenes, como se indica a continuación:

Se comienza con la lista de los números enteros a partir de 1:

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25...
```

Se eliminan los números de dos en dos

```
1,  3,  5,  7,  9,   11,   13,   15,   17,   19,   21,   23,   25...
```

Como el segundo número que ha quedado es 3, se eliminan los números restantes de tres en tres:

```
1,  3,      7,  9,         13,   15,         19,   21,         25...
```

Como el tercer número que ha quedado es 7, se eliminan los números restantes de siete en siete:

```
1,  3,      7,  9,        13,  15,              21,       25...
```

Este procedimiento se repite indefinidamente y los supervivientes son los números de la suerte:

```
1,3,7,9,13,15,21,25,31,33,37,43,49,51,63,67,69,73,75,79
```

Definir las funciones

```
numerosDeLaSuerte  :: [Int]
esNumeroDeLaSuerte :: Int -> Bool
```

tales que

- numerosDeLaSuerte es la sucesión de los números de la suerte. Por ejemplo,

```
λ> take 20 numerosDeLaSuerte
[1,3,7,9,13,15,21,25,31,33,37,43,49,51,63,67,69,73,75,79]
λ> numerosDeLaSuerte !! 277
2019
λ> numerosDeLaSuerte !! 2000
19309
```

- (esNumeroDeLaSuerte n) que se verifica si n es un número de la suerte. Por ejemplo,

```
esNumeroDeLaSuerte 15    ==  True
esNumeroDeLaSuerte 16    ==  False
esNumeroDeLaSuerte 2019  ==  True
```

Soluciones

```
-- 1ª definición de numerosDeLaSuerte
numerosDeLaSuerte :: [Int]
numerosDeLaSuerte = criba 3 [1,3..]
```

```
  where
    criba i (n:s:xs) =
      n : criba (i + 1) (s : [x | (k, x) <- zip [i..] xs
                                , rem k s /= 0])


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

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

--    λ> numerosDeLaSuerte2 !! 200
--    1387
--    (9.25 secs, 2,863,983,232 bytes)
--    λ> numerosDeLaSuerte !! 200
--    1387
--    (0.06 secs, 10,263,880 bytes)

-- Definición de esNumeroDeLaSuerte
esNumeroDeLaSuerte :: Int -> Bool
esNumeroDeLaSuerte n =
  n == head (dropWhile (<n) numerosDeLaSuerte)
```

Ejercicio 42

Cadena descendiente de subnúmeros

La inseguridad, la incertidumbre, la desconfianza, son acaso nuestras únicas verdades. Hay que aferrarse a ellas.

Antonio Machado

Enunciado

Una particularidad del 2019 es que se puede escribir como una cadena de dos subnúmeros consecutivos (el 20 y el 19).

Definir la función

```
cadena :: Integer -> [Integer]
```

tal que (cadena n) es la cadena de subnúmeros consecutivos de n cuya unión es n; es decir, es la lista de números [x,x-1,...x-k] tal que su concatenación es n. Por ejemplo,

```
cadena 2019          == [20,19]
cadena 2018          == [2018]
cadena 1009          == [1009]
cadena 110109        == [110,109]
```

```
cadena 201200199198 == [201,200,199,198]
cadena 3246         == [3246]
cadena 87654        == [8,7,6,5,4]
cadena 123456       == [123456]
cadena 1009998      == [100,99,98]
cadena 100908       == [100908]
cadena 1110987      == [11,10,9,8,7]
cadena 210          == [2,1,0]
cadena 1            == [1]
cadena 0            == [0]
cadena 312          == [312]
cadena 191          == [191]
length (cadena (read (concatMap show [2019,2018..0])))  ==  2020
```

Nota: Los subnúmeros no pueden empezar por cero. Por ejemplo, [10,09] no es una cadena de 1009 como se observa en el tercer ejemplo.

Soluciones

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

-- 1ª solución
-- ===========

cadena :: Integer -> [Integer]
cadena = head . cadenasL . digitos

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

-- (cadenasL xs) son las cadenas descendientes del número cuyos dígitos
-- son xs. Por ejemplo,
--    cadenasL [2,0,1,9]      == [[20,19],[2019]]
--    cadenasL [1,0,0,9]      == [[1009]]
--    cadenasL [1,1,0,1,0,9]  == [[110,109],[110109]]
```

```haskell
cadenasL :: [Integer] -> [[Integer]]
cadenasL []       = []
cadenasL [x]      = [[x]]
cadenasL [1,0]    = [[1,0],[10]]
cadenasL (x:0:zs) = cadenasL (10*x:zs)
cadenasL (x:y:zs) =
     [x:a:as | (a:as) <- cadenasL (y:zs), a == x-1]
  ++ cadenasL (10*x+y:zs)

-- 2ª solución
-- ===========

cadena2 :: Integer -> [Integer]
cadena2 n = (head . concatMap aux . iniciales) n
  where aux x = [[x,x-1..x-k] | k <- [0..x]
                              , concatMap show [x,x-1..x-k] == ds]
        ds    = show n

-- (iniciales n) es la lista de los subnúmeros iniciales de n. Por
-- ejemplo,
--    iniciales 2019  ==  [2,20,201,2019]
iniciales :: Integer -> [Integer]
iniciales = map read . tail . inits . show

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

-- La propiedad es
prop_cadena :: Positive Integer -> Bool
prop_cadena (Positive n) =
  cadena n == cadena2 n

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

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

--    λ> length (cadena (read (concatMap show [15,14..0])))
```

```
--    16
--    (3.28 secs, 452,846,008 bytes)
--    λ> length (cadena2 (read (concatMap show [15,14..0])))
--    16
--    (0.03 secs, 176,360 bytes)
```

Ejercicio 43

Mínimo producto escalar

El escepticismo es una posición vital, no lógica, que ni afirma ni niega, se limita a preguntar, y no se asusta de las contradicciones.

Antonio Machado

Enunciado

El producto escalar de los vectores $(a_1, a_2, \ldots, a_n)$ y $(b_1, b_2, \ldots, b_n)$ es

$$a_1 b_1 + a_2 b_2 + \cdots + a_n * b_n$$

Definir la función

```
menorProductoEscalar :: (Ord a, Num a) => [a] -> [a] -> a
```

tal que (menorProductoEscalar xs ys) es el mínimo de los productos escalares de las permutaciones de xs y de las permutaciones de ys. Por ejemplo,

```
menorProductoEscalar [3,2,5]  [1,4,6]        == 29
menorProductoEscalar [3,2,5]  [1,4,-6]       == -19
menorProductoEscalar [0..9]   [0..9]         == 120
menorProductoEscalar [0..99]  [0..99]        == 161700
menorProductoEscalar [0..999] [0..999]       == 166167000
```

```
  menorProductoEscalar [0..9999] [0..9999]       == 166616670000
  menorProductoEscalar [0..99999] [0..99999]     == 166661666700000
  menorProductoEscalar [0..999999] [0..999999]   == 166666166667000000
```

Soluciones

```haskell
import Data.List (sort, permutations)
import Test.QuickCheck

-- 1ª solución
menorProductoEscalar :: (Ord a, Num a) => [a] -> [a] -> a
menorProductoEscalar xs ys =
  minimum [sum (zipWith (*) pxs pys) | pxs <- permutations xs,
                                       pys <- permutations ys]

-- 2ª solución
menorProductoEscalar2 :: (Ord a, Num a) => [a] -> [a] -> a
menorProductoEscalar2 xs ys =
  minimum [sum (zipWith (*) pxs ys) | pxs <- permutations xs]

-- 3ª solución
menorProductoEscalar3 :: (Ord a, Num a) => [a] -> [a] -> a
menorProductoEscalar3 xs ys =
  sum (zipWith (*) (sort xs) (reverse (sort ys)))

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

-- La propiedad es
prop_menorProductoEscalar :: [Integer] -> [Integer] -> Bool
prop_menorProductoEscalar xs ys =
  menorProductoEscalar3 xs' ys' == menorProductoEscalar  xs' ys' &&
  menorProductoEscalar3 xs' ys' == menorProductoEscalar2 xs' ys'
  where n   = min (length xs) (length ys)
        xs' = take n xs
        ys' = take n ys

-- La comprobación es
```

```
--    λ> quickCheckWith (stdArgs {maxSize=7}) prop_menorProductoEscalar
--    +++ OK, passed 100 tests.

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

-- La comparación es
--    λ> menorProductoEscalar1 [0..5] [0..5]
--    20
--    (3.24 secs, 977385528 bytes)
--    λ> menorProductoEscalar2 [0..5] [0..5]
--    20
--    (0.01 secs, 4185776 bytes)
--
--    λ> menorProductoEscalar2 [0..9] [0..9]
--    120
--    (23.86 secs, 9342872784 bytes)
--    λ> menorProductoEscalar3 [0..9] [0..9]
--    120
--    (0.01 secs, 2580824 bytes)
--
--    λ> menorProductoEscalar3 [0..10^6] [0..10^6]
--    166666666666500000
--    (2.46 secs, 473,338,912 bytes)
```

Ejercicio 44

Numeración de ternas de naturales

¿Cabe una comunión cordial entre hombres, que nos permita cantar en coro, animados de un mismo sentir?

Antonio Machado

Enunciado

Las ternas de números naturales se pueden ordenar como sigue

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

Definir la función

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

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

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

Comprobar con QuickCheck que

- la posición de (x,0,0) es $x(x^2+6x+11)/6$
- la posición de (0,y,0) es $y(y^2+3y+ 8)/6$
- la posición de (0,0,z) es $z(z^2+3z+ 2)/6$
- la posición de (x,x,x) es $x(9x^2+14x+7)/2$

Soluciones

```haskell
import Test.QuickCheck
import Data.List (elemIndex)
import Data.Maybe (fromJust)

-- 1ª solución
-- ===========

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

-- ternas es la lista ordenada de las ternas de números naturales. Por ejemplo,
--    ghci> take 9 ternas
--    [(0,0,0),(0,0,1),(0,1,0),(1,0,0),(0,0,2),(0,1,1),(0,2,0),(1,0,1),(1,1,0)]
ternas :: [(Int,Int,Int)]
ternas = [(x,y,n-x-y) | n <- [0..], x <- [0..n], y <- [0..n-x]]

-- 2ª solución
-- ===========

posicion2 :: (Int,Int,Int) -> Int
posicion2 t = aux 0 ternas
  where aux n (t':ts) | t' == t   = n
```

```haskell
                          | otherwise = aux (n+1) ts

-- 3ª solución
-- ===========

posicion3 :: (Int,Int,Int) -> Int
posicion3 t =
  head [n | (n,t') <- zip [0..] ternas, t' == t]

-- 4ª solución
-- ===========

posicion4 :: (Int,Int,Int) -> Int
posicion4 t = fromJust (elemIndex t ternas)

-- 5ª solución
-- ===========

posicion5 :: (Int,Int,Int) -> Int
posicion5 = fromJust . (`elemIndex` ternas)

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

-- La propiedad es
prop_posicion_equiv :: NonNegative Int
                    -> NonNegative Int
                    -> NonNegative Int
                    -> Bool
prop_posicion_equiv (NonNegative x) (NonNegative y) (NonNegative z) =
  all (== posicion (x,y,z)) [f (x,y,z) | f <- [ posicion2
                                              , posicion3
                                              , posicion4
                                              , posicion5
                                              ]]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=20}) prop_posicion_equiv
--    +++ OK, passed 100 tests.
```

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

--   λ> posicion (200,0,0)
--   1373700
--   (2.48 secs, 368,657,528 bytes)
--   λ> posicion2 (200,0,0)
--   1373700
--   (3.96 secs, 397,973,912 bytes)
--   λ> posicion3 (200,0,0)
--   1373700
--   (1.47 secs, 285,831,056 bytes)
--   λ> posicion4 (200,0,0)
--   1373700
--   (0.13 secs, 102,320 bytes)
--   λ> posicion5 (200,0,0)
--   1373700
--   (0.14 secs, 106,376 bytes)

-- Propiedades
-- ===========

-- La 1ª propiedad es
prop_posicion1 :: NonNegative Int -> Bool
prop_posicion1 (NonNegative x) =
  posicion (x,0,0) == x * (x^2 + 6*x + 11) `div` 6

-- Su comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=20}) prop_posicion1
--    +++ OK, passed 100 tests.

-- La 2ª propiedad es
prop_posicion2 :: NonNegative Int -> Bool
prop_posicion2 (NonNegative y) =
  posicion (0,y,0) == y * (y^2 + 3*y + 8) `div` 6

-- Su comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=20}) prop_posicion2
--    +++ OK, passed 100 tests.
```

```haskell
-- La 3ª propiedad es
prop_posicion3 :: NonNegative Int -> Bool
prop_posicion3 (NonNegative z) =
  posicion (0,0,z) == z * (z^2 + 3*z + 2) `div` 6

-- Su comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=20}) prop_posicion3
--    +++ OK, passed 100 tests.

-- La 4ª propiedad es
prop_posicion4 :: NonNegative Int -> Bool
prop_posicion4 (NonNegative x) =
  posicion (x,x,x) == x * (9 * x^2 + 14 * x + 7) `div` 2

-- Su comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=20}) prop_posicion4
--    +++ OK, passed 100 tests.
```

Ejercicio 45

Subárboles monovalorados

Y nadie pregunta
ni nadie contesta,
todos hablan solos.

Antonio Machado

Enunciado

Los árboles binarios con valores enteros se pueden representar mediante el tipo Arbol definido por

```
data Arbol = H Int
           | N Int Arbol Arbol
           deriving Show
```

Por ejemplo, el árbol

```
        7
       / \
      /   \
     /     \
    4       9
   / \     / \
  1   3   5   6
```

se puede representar por

```
N 7 (N 4 (H 1) (H 3)) (N 9 (H 5) (H 6))
```

Un árbol es monovalorado si todos sus elementos son iguales. Por ejemplo, de los siguientes árboles sólo son monovalorados los dos primeros

```
   5           9             5           9
  / \         / \           / \         / \
 5   5       9   9         5   6       9   7
                / \                       / \
               9   9                     9   9
```

Definir la función

```
monovalorados :: Arbol -> [Arbol]
```

tal que (monovalorados a) es la lista de los subárboles monovalorados de a. Por ejemplo,

```
λ> monovalorados (N 5 (H 5) (H 5))
[N 5 (H 5) (H 5),H 5,H 5]
λ> monovalorados (N 5 (H 5) (H 6))
[H 5,H 6]
λ> monovalorados (N 9 (H 9) (N 9 (H 9) (H 9)))
[N 9 (H 9) (N 9 (H 9) (H 9)),H 9,N 9 (H 9) (H 9),H 9,H 9]
λ> monovalorados (N 9 (H 9) (N 7 (H 9) (H 9)))
[H 9,H 9,H 9]
λ> monovalorados (N 9 (H 9) (N 9 (H 7) (H 9)))
[H 9,H 7,H 9]
```

Soluciones

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

monovalorados :: Arbol -> [Arbol]
```

```haskell
monovalorados (H x) = [H x]
monovalorados (N x i d)
    | todosIguales i x && todosIguales d x =
        N x i d : subarboles i ++ subarboles d
    | otherwise = monovalorados i ++ monovalorados d

-- (todosIguales a x) se verifica si todos los valores de los nodos y
-- las hojas del árbol a son iguales a x.
todosIguales :: Arbol -> Int -> Bool
todosIguales (H y) x     = y == x
todosIguales (N y i d) x = y == x && todosIguales i x && todosIguales d x

-- (subarboles a) es la lista de los subárboles de a.
subarboles :: Arbol -> [Arbol]
subarboles (H x)     = [H x]
subarboles (N x i d) = N x i d : subarboles i ++ subarboles d
```

Ejercicio 46

Mayor prefijo con suma acotada

Sed hombres de mal gusto. Yo os aconsejo el mal gusto para combatir los excesos de la moda.

Antonio Machado

Enunciado

Definir la función

```
mayorPrefijoAcotado :: [Int] -> Int -> [Int]
```

tal que (mayorPrefijoAcotado xs y) es el mayor prefijo de la lista de números enteros positivos xs cuya suma es menor o igual que y. Por ejemplo,

```
mayorPrefijoAcotado [45,30,55,20,80,20] 75   ==  [45,30]
mayorPrefijoAcotado [45,30,55,20,80,20] 140  ==  [45,30,55]
mayorPrefijoAcotado [45,30,55,20,80,20] 180  ==  [45,30,55,20]
length (mayorPrefijoAcotado (repeat 1) (8*10^6)) == 8000000
```

Soluciones

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

```haskell
-- 1ª solución
-- ===========

mayorPrefijoAcotado :: [Int] -> Int -> [Int]
mayorPrefijoAcotado [] _     = []
mayorPrefijoAcotado (x:xs) y
  | x > y     = []
  | otherwise = x : mayorPrefijoAcotado xs (y-x)

-- 2ª solución
-- ===========

mayorPrefijoAcotado2 :: [Int] -> Int -> [Int]
mayorPrefijoAcotado2 xs y =
  take (longitudMayorPrefijoAcotado2 xs y) xs

longitudMayorPrefijoAcotado2 :: [Int] -> Int -> Int
longitudMayorPrefijoAcotado2 xs y =
  length (takeWhile (<=y) (map sum (inits xs))) - 1

-- 3ª solución
-- ===========

mayorPrefijoAcotado3 :: [Int] -> Int -> [Int]
mayorPrefijoAcotado3 xs y =
  take (longitudMayorPrefijoAcotado3 xs y) xs

longitudMayorPrefijoAcotado3 :: [Int] -> Int -> Int
longitudMayorPrefijoAcotado3 xs y =
  length (takeWhile (<= y) (scanl1 (+) xs))

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

-- La propiedad es
prop_equiv :: [Int] -> Int -> Bool
prop_equiv xs y =
  mayorPrefijoAcotado xs' y' == mayorPrefijoAcotado2 xs' y' &&
  mayorPrefijoAcotado xs' y' == mayorPrefijoAcotado3 xs' y'
  where xs' = map abs xs
```

```
        y'  = abs y

-- La comprobación es
--    λ> quickCheck prop_equiv
--    +++ OK, passed 100 tests.
--    (0.01 secs, 2,463,688 bytes)

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

--    λ> length (mayorPrefijoAcotado (repeat 1) (2*10^4))
--    20000
--    (0.04 secs, 5,086,544 bytes)
--    λ> length (mayorPrefijoAcotado2 (repeat 1) (2*10^4))
--    20000
--    (11.22 secs, 27,083,980,168 bytes)
--    λ> length (mayorPrefijoAcotado3 (repeat 1) (2*10^4))
--    20000
--    (0.02 secs, 4,768,992 bytes)
--
--    λ> length (mayorPrefijoAcotado (repeat 1) (8*10^6))
--    8000000
--    (3.19 secs, 1,984,129,832 bytes)
--    λ> length (mayorPrefijoAcotado3 (repeat 1) (8*10^6))
--    8000000
--    (1.02 secs, 1,856,130,936 bytes)
```

Ejercicio 47

Ofertas 3 por 2

Despacito y buena letra:
el hacer las cosas bien
importa más que el hacerlas.

Antonio Machado

Enunciado

En una tienda tiene la “oferta 3 por 2” de forma que cada cliente que elige 3 artículos obtiene el más barato de forma gratuita. Por ejemplo, si los precios de los artículos elegidos por un cliente son 10, 2, 4, 5 euros pagará 19 euros si agrupa los artículos en (10,2,4) y (5) o pagará 17 si lo agrupa en (5,10,4) y (2).

Definir la función

```
minimoConOferta :: [Int] -> Int
```

tal que (minimoConOferta xs) es lo mínimo que pagará el cliente si los precios de la compra son xs; es decir, lo que pagará agrupando los artículos de forma óptima para aplicar la oferta 3 por 2. Por ejemplo,

```
minimoConOferta [10,2,4,5]     ==  17
minimoConOferta [3,2,3,2]      ==  8
minimoConOferta [6,4,5,5,5,5]  ==  21
```

Soluciones

```haskell
import Data.List (sort, sortOn)
import Data.Ord  (Down(..))

-- 1ª solución
-- ===========

minimoConOferta :: [Int] -> Int
minimoConOferta xs = sum (sinTerceros (reverse (sort xs)))

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

-- 2ª solución
-- ===========

minimoConOferta2 :: [Int] -> Int
minimoConOferta2 = sum . sinTerceros . reverse . sort

-- 3ª solución
-- ===========

minimoConOferta3 :: [Int] -> Int
minimoConOferta3 = sum . sinTerceros . sortOn Down

-- 4ª solución
-- ===========

minimoConOferta4 :: [Int] -> Int
minimoConOferta4 xs = aux (reverse (sort xs)) 0
  where aux (a:b:_:ds) n = aux ds (a+b+n)
        aux as         n = n + sum as

-- 5ª solución
-- ===========
```

```haskell
minimoConOferta5 :: [Int] -> Int
minimoConOferta5 xs = aux (sortOn Down xs) 0
  where aux (a:b:_:ds) n = aux ds (a+b+n)
        aux as         n = n + sum as
```

Ejercicio 48

Representación de conjuntos mediante intervalos

Cuando el saber se especializa, crece el volumen total de la cultura. Esta es la ilusión y el consuelo de los especialistas. ¡Lo que sabemos entre todos! ¡Oh, eso es lo que no sabe nadie!

Antonio Machado

Enunciado

Un conjunto de números enteros se pueden representar mediante una lista ordenada de intervalos tales que la diferencia entre el menor elemento de un intervalo y el mayor elemento de su intervalo anterior es mayor que uno.

Por ejemplo, el conjunto {2, 7, 4, 3, 9, 6} se puede representar mediante la lista de intervalos [(2,4),(6,7),(9,9)] de forma que en el primer intervalo se agrupan los números 2, 3 y 4; en el segundo, los números 6 y 7 y el tercero, el número 9.

Definir la función

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

tal que (intervalos xs) es lista ordenada de intervalos que representa al conjunto xs. Por ejemplo,

```
λ> intervalos [2,7,4,3,9,6]
[(2,4),(6,7),(9,9)]
λ> intervalos [180,141,174,143,142,175]
[(141,143),(174,175),(180,180)]
```

Soluciones

```
import Data.List (sort)

intervalos :: [Int] -> [(Int,Int)]
intervalos = map intervalo . segmentos

-- (segmentos xs) es la lista de segmentos formados por elementos
-- consecutivos de xs. Por ejemplo,
--    segmentos [2,7,4,3,9,6]  ==  [[2,3,4],[6,7],[9]]
segmentos :: [Int] -> [[Int]]
segmentos xs = aux bs [[b]]
  where aux [] zs = zs
        aux (y:ys) ((a:as):zs) | y == a-1  = aux ys ((y:a:as):zs)
                               | otherwise = aux ys ([y]:(a:as):zs)
        (b:bs) = reverse (sort xs)

-- (intervalo xs) es el intervalo correspondiente al segmento xs. Por
-- ejemplo,
--    intervalo [2,3,4]  ==  (2,4)
--    intervalo [6,7]    ==  (6,7)
--    intervalo [9]      ==  (9,9)
intervalo :: [Int] -> (Int,Int)
intervalo xs = (head xs, last xs)
```

Ejercicio 49

Números altamente compuestos

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

Antonio Machado

Enunciado

Un número [altamente compuesto](http://bit.ly/2H7Vj61) es un entero positivo con más divisores que cualquier entero positivo más pequeño. Por ejemplo,

- 4 es un número altamente compuesto porque es el menor con 3 divisores,
- 5 no es altamente compuesto porque tiene menos divisores que 4 y
- 6 es un número altamente compuesto porque es el menor con 4 divisores,

Los primeros números altamente compuestos son

```
1, 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, ...
```

Definir las funciones

```
esAltamenteCompuesto        :: Int -> Bool
altamenteCompuestos         :: [Int]
graficaAltamenteCompuestos :: Int -> IO ()
```

tales que

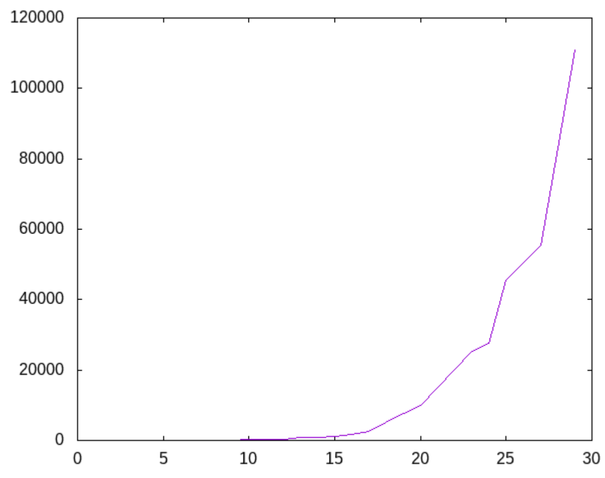
- (esAltamanteCompuesto x) se verifica si x es altamente compuesto. Por ejemplo,

```
esAltamenteCompuesto 4      ==  True
esAltamenteCompuesto 5      ==  False
esAltamenteCompuesto 6      ==  True
esAltamenteCompuesto 1260   ==  True
esAltamenteCompuesto 2520   ==  True
esAltamenteCompuesto 27720  ==  True
```

- altamente compuestos es la sucesión de los números altamente compuestos. Por ejemplo,

```
λ> take 20 altamenteCompuestos
[1,2,4,6,12,24,36,48,60,120,180,240,360,720,840,1260,1680,2520,5040,7560]
```

- (graficaAltamenteCompuestos n) dibuja la gráfica de los n primeros números altamente compuestos. Por ejemplo,
(graficaAltamenteCompuestos 25) dibuja



Soluciones

```haskell
import Data.List (group)
import Data.Numbers.Primes (primeFactors)
import Graphics.Gnuplot.Simple

-- 1ª definición de esAltamenteCompuesto
-- ======================================

esAltamenteCompuesto :: Int -> Bool
esAltamenteCompuesto x =
  and [nDivisores x > nDivisores y | y <- [1..x-1]]

-- (nDivisores x) es el número de divisores de x. Por ejemplo,
--    nDivisores 30  ==  8
nDivisores :: Int -> Int
nDivisores x = length (divisores x)

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

-- 2ª definición de esAltamenteCompuesto
-- ======================================

esAltamenteCompuesto2 :: Int -> Bool
esAltamenteCompuesto2 x =
  all (nDivisores2 x >) [nDivisores2 y | y <- [1..x-1]]

-- (nDivisores2 x) es el número de divisores de x. Por ejemplo,
--    nDivisores2 30  ==  8
nDivisores2 :: Int -> Int
nDivisores2 = succ . length . divisoresPropios

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

```haskell
divisoresPropios :: Int -> [Int]
divisoresPropios x =
  [y | y <- [1..x `div` 2]
     , x `mod` y == 0]

-- 3ª definición de esAltamenteCompuesto
-- ====================================

esAltamenteCompuesto3 :: Int -> Bool
esAltamenteCompuesto3 x =
  all (nDivisores3 x >) [nDivisores3 y | y <- [1..x-1]]

-- (nDivisores3 x) es el número de divisores de x. Por ejemplo,
--    nDivisores3 30  ==  8
nDivisores3 :: Int -> Int
nDivisores3 x =
  product [1 + length xs | xs <- group (primeFactors x)]

-- 4ª definición de esAltamenteCompuesto
-- ====================================

esAltamenteCompuesto4 :: Int -> Bool
esAltamenteCompuesto4 x =
  x `pertenece` altamenteCompuestos2

-- 1ª definición de altamenteCompuestos
-- ===================================

altamenteCompuestos :: [Int]
altamenteCompuestos =
  filter esAltamenteCompuesto4 [1..]

-- 2ª definición de altamenteCompuestos
-- ===================================

altamenteCompuestos2 :: [Int]
altamenteCompuestos2 =
  1 : [y | ((_,n),(y,m)) <- zip sucMaxDivisores (tail sucMaxDivisores)
         , m > n]
```

```haskell
-- sucMaxDivisores es la sucesión formada por los números enteros
-- positivos y el máximo número de divisores hasta cada número. Por
-- ejemplo,
--    λ> take 12 sucMaxDivisores
--    [(1,1),(2,2),(3,2),(4,3),(5,3),(6,4),(7,4),(8,4),(9,4),(10,4),(11,4),(12,6)
sucMaxDivisores :: [(Int,Int)]
sucMaxDivisores =
  zip [1..] (scanl1 max (map nDivisores3 [1..]))

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

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

--    λ> esAltamenteCompuesto 1260
--    True
--    (2.99 secs, 499,820,296 bytes)
--    λ> esAltamenteCompuesto2 1260
--    True
--    (0.51 secs, 83,902,744 bytes)
--    λ> esAltamenteCompuesto3 1260
--    True
--    (0.04 secs, 15,294,192 bytes)
--    λ> esAltamenteCompuesto4 1260
--    True
--    (0.04 secs, 15,594,392 bytes)
--
--    λ> esAltamenteCompuesto2 2520
--    True
--    (2.10 secs, 332,940,168 bytes)
--    λ> esAltamenteCompuesto3 2520
--    True
--    (0.09 secs, 37,896,168 bytes)
--    λ> esAltamenteCompuesto4 2520
--    True
--    (0.06 secs, 23,087,456 bytes)
--
--    λ> esAltamenteCompuesto3 27720
```

```haskell
--    True
--    (1.32 secs, 841,010,624 bytes)
--    λ> esAltamenteCompuesto4 27720
--    True
--    (1.33 secs, 810,870,384 bytes)

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

--    λ> altamenteCompuestos !! 25
--    45360
--    (2.84 secs, 1,612,045,976 bytes)
--    λ> altamenteCompuestos2 !! 25
--    45360
--    (0.01 secs, 102,176 bytes)

-- Definición de graficaAltamenteCompuestos
-- ========================================

graficaAltamenteCompuestos :: Int -> IO ()
graficaAltamenteCompuestos n =
  plotList [ Key Nothing
           , PNG ("Numeros_altamente_compuestos.png")
           ]
           (take n altamenteCompuestos2)
```

Ejercicio 50

Posiciones del 2019 en el número pi

> *Aprendió tantas cosas, que no tuvo tiempo para pensar en ninguna de ellas.*
>
> Antonio Machado

Enunciado

El fichero Digitos_de_pi.txt contiene el número pi con un millón de decimales; es decir,

$$3{,}14159265358979323846264338 32\ldots 83996346460422090106105779458151$$

Definir la función

```haskell
posiciones :: String -> Int -> IO [Int]
```

tal que (posicion cs k) es es la lista de las posiciones iniciales de cs en la sucesión formada por los k primeros dígitos decimales del número pi. Por ejemplo,

```haskell
λ> posiciones "141" 1000
[0,294]
```

```
λ> posiciones "4159" 10000
[1,5797,6955,9599]
```

Calcular la primera posición de 2019 en los decimales de pi y el número de veces que aparece 2019 en en el primer millón de decimales de pi.

Soluciones

```
import Data.List ( isPrefixOf
                 , findIndices
                 , tails
                 )

-- 1ª definición
-- =============

posiciones :: String -> Int -> IO [Int]
posiciones cs k = do
  ds <- readFile "Digitos_de_pi.txt"
  return (posicionesEnLista cs (take (k-1) (drop 2 ds)))

--    posicionesEnLista "23" "234235523"  ==  [0,3,7]
posicionesEnLista :: Eq a => [a] -> [a] -> [Int]
posicionesEnLista xs ys = reverse (aux ys 0 [])
  where aux []      _ ns = ns
        aux (y:ys') n ns | xs `isPrefixOf` (y:ys') = aux ys' (n+1) (n:ns)
                         | otherwise               = aux ys' (n+1) ns

-- 2ª definición
-- =============

posiciones2 :: String -> Int -> IO [Int]
posiciones2 cs k = do
  ds <- readFile "Digitos_de_pi.txt"
  return (findIndices (cs `isPrefixOf`) (tails (take (k-1) (drop 2 ds))))

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

```
--    λ> length <$> posiciones "2019" (10^6)
--    112
--    (1.73 secs, 352,481,272 bytes)
--    λ> length <$> posiciones2 "2019" (10^6)
--    112
--    (0.16 secs, 144,476,384 bytes)

-- El cálculo es
--    λ> ps <- posiciones "2019" (10^6)
--    λ> head ps
--    243
--    λ> length ps
--    112
-- Por tanto, la posición de la primera ocurrencia es 243 y hay 112
-- ocurrencias. Otra forma de hacer los cálculos anteriores es
--    λ> head <$> posiciones "2019" (10^6)
--    243
--    λ> length <$> posiciones "2019" (10^6)
--    112
```

Ejercicio 51

Mínimo número de operaciones para transformar un número en otro

¿Dijiste media verdad?
Dirán que mientes dos veces
si dices la otra mitad.

Antonio Machado

Enunciado

Se considera el siguiente par de operaciones sobre los números:

- multiplicar por dos
- restar uno.

Dados dos números x e y se desea calcular el menor número de operaciones para transformar x en y. Por ejemplo, el menor número de operaciones para transformar el 4 en 7 es 2:

```
4 ------> 8 ------> 7
   (*2)      (-1)
```

y el menor número de operaciones para transformar 2 en 5 es 4

```
 2 ------> 4 ------> 3 ------> 6 ------> 5
    (*2)      (-1)      (*2)      (-1)
```

Definir las siguientes funciones

```haskell
arbolOp :: Int -> Int -> Arbol
minNOp  :: Int -> Int -> Int
```

tales que

- (arbolOp x n) es el árbol de profundidad n obtenido aplicándole a x las dos operaciones. Por ejemplo,

```
λ> arbolOp 4 1
N 4 (H 8) (H 3)
λ> arbolOp 4 2
N 4 (N 8 (H 16) (H 7))
    (N 3 (H 6) (H 2))
λ> arbolOp 2 3
N 2 (N 4
       (N 8 (H 16) (H 7))
       (N 3 (H 6) (H 2)))
    (N 1
       (N 2 (H 4) (H 1))
       (H 0))
λ> arbolOp 2 4
N 2 (N 4 (N 8
             (N 16 (H 32) (H 15))
             (N 7 (H 14) (H 6)))
          (N 3
             (N 6 (H 12) (H 5))
             (N 2 (H 4) (H 1))))
    (N 1 (N 2
             (N 4 (H 8) (H 3))
             (N 1 (H 2) (H 0)))
          (H 0))
```

- (minNOp x y) es el menor número de operaciones necesarias para transformar x en y. Por ejemplo,

```
minNOp 4 7  ==  2
minNOp 2 5  ==  4
minNOp 2 2  ==  0
```

Soluciones

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

arbolOp :: Int -> Int -> Arbol
arbolOp 0 _ = H 0
arbolOp x 0 = H x
arbolOp x n = N x (arbolOp (2 * x) (n - 1)) (arbolOp (x - 1) (n - 1))

ocurre :: Int -> Arbol -> Bool
ocurre x (H y)     = x == y
ocurre x (N y i d) = x == y || ocurre x i || ocurre x d

minNOp :: Int -> Int -> Int
minNOp x y =
  head [n | n <- [0..]
          , ocurre y (arbolOp x n)]
```

Ejercicio 52

Intersección de listas infinitas crecientes

Alguna vez he pensado
si el alma será la ausencia,
mientras más cerca más lejos;
mientras más lejos más cerca.

Antonio Machado

Enunciado

Definir la función

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

tal que (interseccion xss) es la intersección de la lista no vacía de listas infinitas crecientes xss; es decir, la lista de los elementos que pertenecen a todas las listas de xss. Por ejemplo,

```
λ> take 10 (interseccion [[2,4..],[3,6..],[5,10..]])
[30,60,90,120,150,180,210,240,270,300]
λ> take 10 (interseccion [[2,5..],[3,5..],[5,7..]])
[5,11,17,23,29,35,41,47,53,59]
```

Soluciones

```
-- 1ª solución
-- ===========

interseccion :: Ord a => [[a]] -> [a]
interseccion [xs]         = xs
interseccion (xs:ys:zss) = interseccionDos xs (interseccion (ys:zss))

interseccionDos :: Ord a => [a] -> [a] -> [a]
interseccionDos (x:xs) (y:ys)
  | x == y    = x : interseccionDos xs ys
  | x < y     = interseccionDos (dropWhile (<y) xs) (y:ys)
  | otherwise = interseccionDos (x:xs) (dropWhile (<x) ys)

-- 2ª solución
-- ===========

interseccion2 :: Ord a => [[a]] -> [a]
interseccion2 = foldl1 interseccionDos

-- 3ª solución
-- ===========

interseccion3 :: Ord a => [[a]] -> [a]
interseccion3 (xs:xss) =
  [x | x <- xs, all (x `pertenece`) xss]

pertenece :: Ord a => a -> [a] -> Bool
pertenece x xs = x == head (dropWhile (<x) xs)
```

Ejercicio 53

Soluciones de $x^2 = y^3 = k$

Leyendo a Cervantes me parece comprenderlo todo.

Antonio Machado

Enunciado

Definir la función

```
soluciones :: [(Integer,Integer,Integer)]
```

tal que sus elementos son las ternas (x,y,k) de soluciones del sistema $x^2 = y^3 = k$. Por ejemplo,

```
λ> take 6 soluciones
[(0,0,0),(-1,1,1),(1,1,1),(-8,4,64),(8,4,64),(-27,9,729)]
λ> soluciones !! (6*10^5+6)
(27000810008100027,90001800009,729043741093514580109350437400729)
```

Soluciones

```
-- 1ª solución
-- ===========
```

```haskell
soluciones :: [(Integer,Integer,Integer)]
soluciones = [(n^3, n^2, n^6) | n <- enteros]

-- enteros es la lista ordenada de los números enteros. Por ejemplo,
--    λ> take 20 enteros
--    [0,-1,1,-2,2,-3,3,-4,4,-5,5,-6,6,-7,7,-8,8,-9,9,-10]
enteros :: [Integer]
enteros = 0 : concat [[-x,x] | x <- [1..]]

-- 2ª solución
-- ===========

soluciones2 :: [(Integer,Integer,Integer)]
soluciones2 = [(x^3,x^2,x^6) | x <- 0 : aux 1]
  where aux n  = -n : n : aux (n+1)

-- 3ª solución
-- ===========

soluciones3 :: [(Integer,Integer,Integer)]
soluciones3 =
  (0,0,0) : [(x,y,k) | k <- [n^6 | n <- [1..]]
                     , let Just x' = raiz 2 k
                     , let Just y  = raiz 3 k
                     , x <- [-x',x']]

-- (raiz n x) es es justo la raíz n-ésima del número natural x, si x es
-- una potencia n-ésima y Nothing en caso contrario. Por ejemplo,
--    raiz 2 16   ==  Just 4
--    raiz 3 216  ==  Just 6
--    raiz 5 216  ==  Nothing
raiz :: Int -> Integer -> Maybe Integer
raiz _ 1 = Just 1
raiz n 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
```

```haskell
                    d = c^n

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

--    λ> soluciones !! (6*10^5+6)
--    (27000810008100027,90001800009,729043741093514580109350437400729)
--    (1.87 secs, 247,352,728 bytes)
--    λ> soluciones2 !! (6*10^5+6)
--    (27000810008100027,90001800009,729043741093514580109350437400729)
--    (1.44 secs, 243,012,936 bytes)
--    λ> soluciones3 !! (6*10^5+6)
--    (27000810008100027,90001800009,729043741093514580109350437400729)
--    (0.84 secs, 199,599,664 bytes)
```

Ejercicio 54

Sucesión triangular

Nadie debe asustarse de lo que piensa, aunque su pensar aparezca en pugna con las leyes más elementales de la lógica. Porque todo ha de ser pensado por alguien, y el mayor desatino puede ser un punto de vista de lo real.

Antonio Machado

Enunciado

La sucesión triangular es la obtenida concatenando las listas [1], [1,2], [1,2,3], [1,2,3,4], ...

Sus primeros términos son 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6

Definir las funciones

```haskell
sucTriangular        :: [Integer]
terminoSucTriangular :: Int -> Integer
graficaSucTriangular :: Int -> IO ()
```

tales que

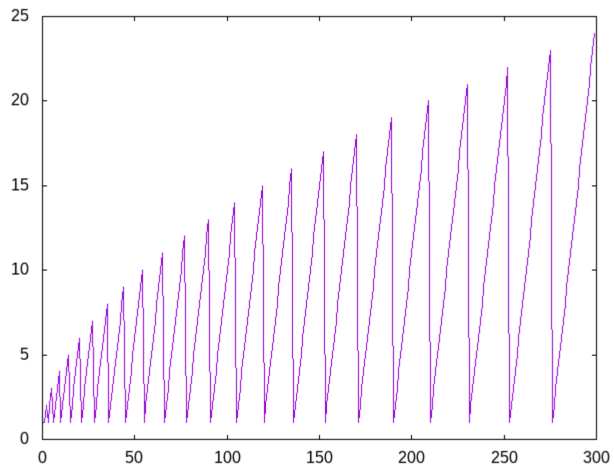
- sucTriangular es la lista de los términos de la sucesión triangular. Por ejemplo,

```
λ> take 30 sucTriangular
[1,1,2,1,2,3,1,2,3,4,1,2,3,4,5,1,2,3,4,5,6,1,2,3,4,5,6,7,1,2]
```

- (terminoSucTriangular n) es el término n-ésimo de la sucesión triangular. Por ejemplo,

```
terminoSucTriangular 5       ==  3
terminoSucTriangular 10      ==  1
terminoSucTriangular 20      ==  6
terminoSucTriangular 100     ==  10
terminoSucTriangular 1001    ==  12
terminoSucTriangular (10^5)  ==  320
```

- (graficaSucTriangular n) dibuja la gráfica de los n primeros términos de la sucesión triangular. Por ejemplo, (graficaSucTriangular 300) dibuja



Soluciones

```haskell
import Data.List (inits)
import Test.QuickCheck
import Graphics.Gnuplot.Simple

-- 1ª definición de sucTriangular
-- ===============================
```

```haskell
sucTriangular :: [Integer]
sucTriangular =
  concat [[1..n] | n <- [1..]]

-- 2ª definición de sucTriangular
-- ===============================

sucTriangular2 :: [Integer]
sucTriangular2 =
  [x | n <- [1..], x <- [1..n]]

-- 3ª definición de sucTriangular
-- ===============================

sucTriangular3 :: [Integer]
sucTriangular3 =
  concat (tail (inits [1..]))

-- 1ª definición de terminoSucTriangular
-- ======================================

terminoSucTriangular :: Int -> Integer
terminoSucTriangular k =
  sucTriangular !! k

-- 2ª definición de terminoSucTriangular
-- ======================================

terminoSucTriangular2 :: Int -> Integer
terminoSucTriangular2 k =
  sucTriangular2 !! k

-- 3ª definición de terminoSucTriangular
-- ======================================

terminoSucTriangular3 :: Int -> Integer
terminoSucTriangular3 k =
  sucTriangular3 !! k

-- Equivalencia de definiciones
```

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

-- La propiedad es
prop_terminoTriangular :: Positive Int -> Bool
prop_terminoTriangular (Positive n) =
  terminoSucTriangular n == terminoSucTriangular2 n &&
  terminoSucTriangular n == terminoSucTriangular3 n

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

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

--    λ> terminoSucTriangular (3*10^6)
--    2425
--    (2.07 secs, 384,707,936 bytes)
--    λ> terminoSucTriangular2 (3*10^6)
--    2425
--    (2.22 secs, 432,571,208 bytes)
--    λ> terminoSucTriangular3 (3*10^6)
--    2425
--    (0.69 secs, 311,259,504 bytes)

-- Definición de graficaSucTriangular
-- ==================================

graficaSucTriangular :: Int -> IO ()
graficaSucTriangular n =
  plotList [ Key Nothing
           , PNG "Sucesion_triangular.png"
           ]
           (take n sucTriangular)
```

Ejercicio 55

Números primos en pi

Al borde del sendero un día nos sentamos.
Ya nuestra vida es tiempo, y nuestra sola cuita
son las desesperantes posturas que tomamos
para aguardar ... Mas ella no faltará a la cita.

Antonio Machado

Enunciado

El fichero Digitos_de_pi.txt contiene el número pi con un millón de decimales; es decir,

```
3.14159265358979323846264338 32 ... 83996346460422090106105779458151
```

Definir las funciones

```
nOcurrenciasPrimosEnPi :: Int -> Int -> IO [Int]
graficaPrimosEnPi      :: Int -> Int -> IO ()
```

tales que

- + (nOcurrenciasPrimosEnPi n k) es la lista de longitud n cuyo i-ésimo elemento es el número de ocurrencias del i-ésimo número primo en los k primeros decimales del número pi. Por ejemplo,

```
nOcurrenciasPrimosEnPi 4 20 == [2,3,3,1]
```

ya que los 20 primeros decimales de pi son 14159265358979323846 y en ellos ocurre el 2 dos veces, el 3 ocurre 3 veces, el 5 ocurre 3 veces y el 7 ocurre 1 vez. Otros ejemplos son

```
λ> nOcurrenciasPrimosEnPi 10 100
[12,11,8,8,1,0,1,1,2,0]
λ> nOcurrenciasPrimosEnPi 10 (10^4)
[1021,974,1046,970,99,102,90,113,99,95]
λ> nOcurrenciasPrimosEnPi 10 (10^6)
[100026,100229,100359,99800,10064,10012,9944,10148,9951,9912]
```

- (graficaPrimosEnPi n k) dibuja la gráfica del número de ocurrencias de los n primeros números primos en los k primeros dígitos de pi. Por ejemplo,
  - `(graficaPrimosEnPi 10 (10^4))` dibuja la Figura 55.1

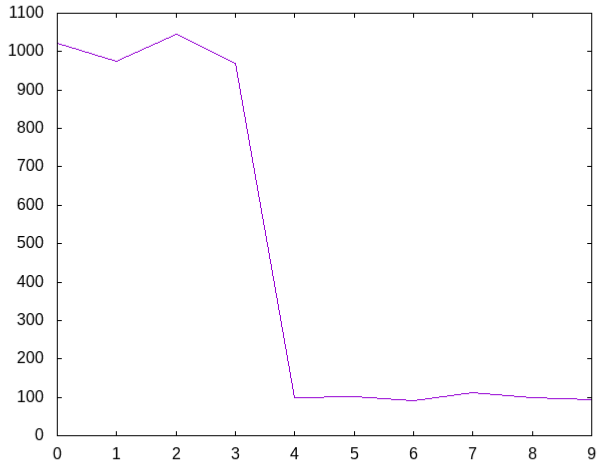


Figura 55.1: (graficaPrimosEnPi 10 10000)

  - `(graficaPrimosEnPi 10 (10^6))` dibuja la Figura 55.2
  - `(graficaPrimosEnPi 50 (10^5))` dibuja la Figura 55.3

Soluciones

```
import Data.List            ( isPrefixOf
                            , findIndices
```

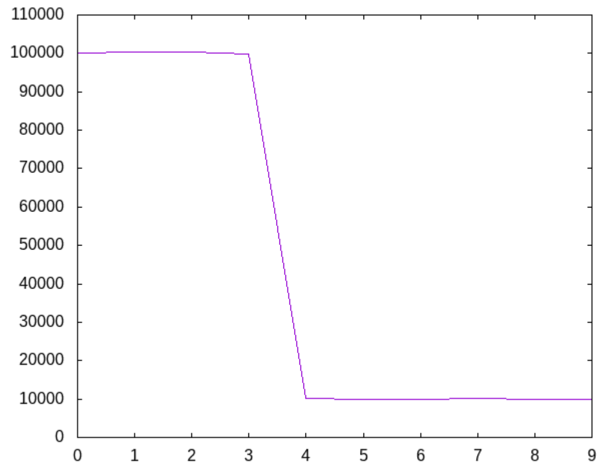


Figura 55.2: (graficaPrimosEnPi 10 1000000)

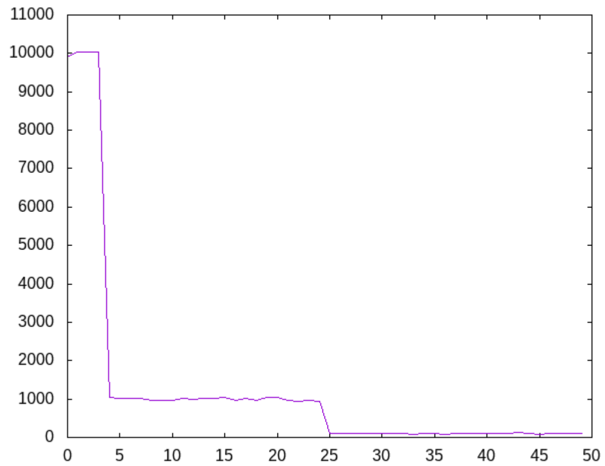


Figura 55.3: (graficaPrimosEnPi 50 100000)

```haskell
                                   , tails )
import Data.Numbers.Primes     ( primes)
import Graphics.Gnuplot.Simple ( Attribute (Key, PNG)
                                   , plotList )

-- Definición de nOcurrenciasPrimosEnPi
-- ====================================

nOcurrenciasPrimosEnPi :: Int -> Int -> IO [Int]
nOcurrenciasPrimosEnPi n k = do
  (_:_:ds) <- readFile "Digitos_de_pi.txt"
  let ps = take n primes
  let es = take k ds
  return [nOcurrencias (show x) es | x <- ps]

-- (nOcurrencias xs yss) es el número de ocurrencias de xs en yss. Por
-- ejemplo,
--    nOcurrencias "ac" "acbadcacaac"  ==  3
nOcurrencias :: Eq a => [a] -> [a] -> Int
nOcurrencias xs yss = length (ocurrencias xs yss)

-- (ocurrencias xs yss) es el índice de las posiciones del primer
-- elemento de xs en las ocurrencias de xs en yss. Por ejemplo,
--    ocurrencias "ac" "acbadcacaac"  ==  [0,6,9]
ocurrencias :: Eq a => [a] -> [a] -> [Int]
ocurrencias xs yss =
  findIndices (xs `isPrefixOf`) (tails yss)

-- Definición de graficaPrimosEnPi
-- ===============================

graficaPrimosEnPi :: Int -> Int -> IO ()
graficaPrimosEnPi n k = do
  xs <- nOcurrenciasPrimosEnPi n k
  plotList [ Key Nothing
           , PNG ("Numeros_primos_en_pi_" ++ show (n,k) ++ ".png")
           ]
           xs
```

Ejercicio 56

Recorrido de árboles en espiral

Dice la monotonía
del agua clara al caer:
un día es como otro día;
hoy es lo mismo que ayer.

Antonio Machado

Enunciado

Los árboles se pueden representar mediante el siguiente tipo de datos

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

Por ejemplo, los árboles

```
        1             1             1
       / \           / \           / \
      /   \         8   3         8   3
     2     3           /|\       /|\   |
    / \   / \          4 5 6     4 5 6 7
   4   5 6   7
```

se representan por

```
  ej1, ej2, ej3 :: Arbol Int
  ej1 = N 1 [N 2 [N 4 [], N 5 []], N 3 [N 6 [], N 7 []]]
  ej2 = N 1 [N 8 [], N 3 [N 4 [], N 5 [], N 6 []]]
  ej3 = N 1 [N 8 [N 4 [], N 5 [], N 6 []], N 3 [N 7 []]]

Definir la función
\begin{descripcion}
  espiral :: Arbol a -> [a]
```

tal que (espiral x) es la lista de los nodos del árbol x recorridos en espiral; es decir, la raíz de x, los nodos del primer nivel de izquierda a derecha, los nodos del segundo nivel de derecha a izquierda y así sucesivamente. Por ejemplo,

```
  espiral ej1  ==  [1,2,3,7,6,5,4]
  espiral ej2  ==  [1,8,3,6,5,4]
  espiral ej3  ==  [1,8,3,7,6,5,4]
```

Soluciones

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

ej1, ej2, ej3 :: Arbol Int
ej1 = N 1 [N 2 [N 4 [], N 5 []], N 3 [N 6 [], N 7 []]]
ej2 = N 1 [N 8 [], N 3 [N 4 [], N 5 [], N 6 []]]
ej3 = N 1 [N 8 [N 4 [], N 5 [], N 6 []], N 3 [N 7 []]]

-- 1ª solución
-- ===========

espiral :: Arbol a -> [a]
espiral x =
  concat [f xs | (f,xs) <- zip (cycle [reverse,id]) (niveles x)]

-- (niveles x) es la lista de los niveles del árbol x. Por ejemplo,
--    niveles ej1 == [[1],[8,3],[4]]
--    niveles ej2 == [[1],[8,3],[4,5,6]]
```

```
--    niveles ej3 == [[1],[8,3],[4,5,6,7]]
niveles :: Arbol a -> [[a]]
niveles x = takeWhile (not . null) [nivel n x | n <- [0..]]

-- (nivel n x) es el nivel de nivel n del árbol x. Por ejemplo,
--    nivel 0 ej1  ==  [1]
--    nivel 1 ej1  ==  [8,3]
--    nivel 2 ej1  ==  [4]
--    nivel 4 ej1  ==  []
nivel :: Int -> Arbol a ->  [a]
nivel 0 (N x _)  = [x]
nivel n (N _ xs) = concatMap (nivel (n-1)) xs

-- 2ª solución
-- ===========

espiral2 :: Arbol a -> [a]
espiral2 =
  concat . zipWith ($) (cycle [reverse,id]) . niveles

-- 3ª solución
-- ===========

espiral3 :: Arbol a -> [a]
espiral3 = concat . zipWith ($) (cycle [reverse,id]) . niveles3

niveles3 :: Arbol a -> [[a]]
niveles3 t = map (map raiz)
           . takeWhile (not . null)
           . iterate (concatMap subBosque) $ [t]

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

subBosque :: Arbol a -> [Arbol a]
subBosque (N _ ts) = ts

-- 4ª solución
-- ===========
```

```haskell
espiral4 :: Arbol a -> [a]
espiral4 = concat . zipWith ($) (cycle [reverse,id]) . niveles4

niveles4 :: Arbol a -> [[a]]
niveles4 = map (map raiz)
         . takeWhile (not . null)
         . iterate (concatMap subBosque)
         . return

-- 5ª definición
-- =============

espiral5 :: Arbol a -> [a]
espiral5 x = concat $ zipWith ($) (cycle [reverse,id]) $ niveles5 [x]

niveles5 :: [Arbol a] -> [[a]]
niveles5 [] = []
niveles5 xs = a : niveles5 (concat b)
  where (a,b) = unzip $ map (\(N x y) -> (x,y)) xs

-- 6ª definición
-- =============

espiral6 :: Arbol a -> [a]
espiral6 = concat . zipWith ($) (cycle [reverse,id]) . niveles5 . return
```

Ejercicio 57

Números con dígitos 1 y 2

¿Para qué llamar caminos
a los surcos del azar? …
Todo el que camina anda,
como Jesús, sobre el mar.

Antonio Machado

Enunciado

Definir las funciones

```
numerosCon1y2                :: Int -> [Int]
restosNumerosCon1y2          :: Int -> [Int]
grafica_restosNumerosCon1y2 :: Int -> IO ()
```

tales que

- (numerosCon1y2 n) es la lista ordenada de números de n dígitos que se pueden formar con los dígitos 1 y 2. Por ejemplo,

```
numerosCon1y2 2  ==  [11,12,21,22]
numerosCon1y2 3  ==  [111,112,121,122,211,212,221,222]
```

- (restosNumerosCon1y2 n) es la lista de los restos de dividir los elementos de (restosNumerosCon1y2 n) entre 2^n. Por ejemplo,

```
restosNumerosCon1y2 2 == [3,0,1,2]
restosNumerosCon1y2 3 == [7,0,1,2,3,4,5,6]
restosNumerosCon1y2 4 == [7,8,1,2,11,12,5,6,15,0,9,10,3,4,13,14]
```

- `(grafica_restosNumerosCon1y2 n)` dibuja la gráfica de los restos de dividir los elementos de (restosNumerosCon1y2 n) entre 2^n. Por ejemplo,
  - `(grafica_restosNumerosCon1y2 3)` dibuja la Figura 57.1

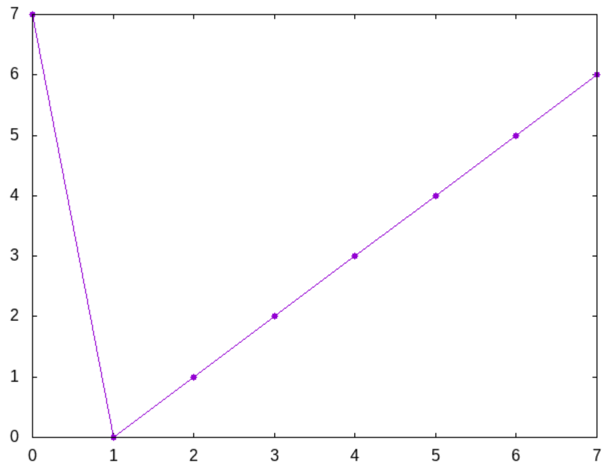


Figura 57.1: (graficaPrimosEnPi 10 10000)

  - `(grafica_restosNumerosCon1y2 4)` dibuja la Figura 57.2

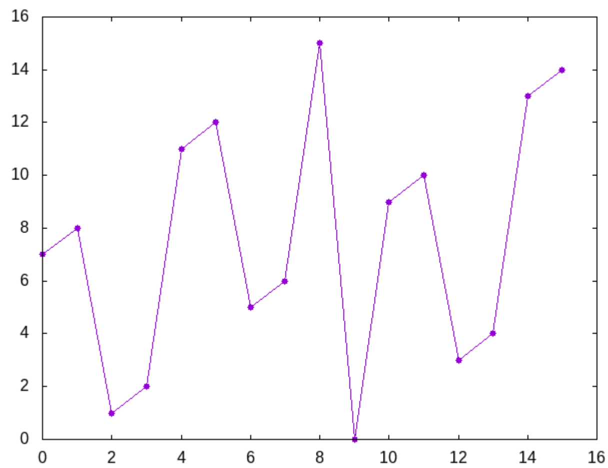


Figura 57.2: (graficaPrimosEnPi 10 10000)

- `(grafica_restosNumerosCon1y2 5)` dibuja la Figura 57.3

Nota: En la definición usar la función plotListStyle y como segundo argumento usar

```
(defaultStyle {plotType = LinesPoints,
               lineSpec = CustomStyle [PointType 7]})
```

Comprobar con QuickCheck que todos los elementos de (restosNumerosCon1y2 n) son distintos.

Soluciones

```haskell
import Data.List (nub)
import Test.QuickCheck
import Graphics.Gnuplot.Simple
import Control.Monad (replicateM)

-- 1ª definición de numerosCon1y2
-- ==============================

numerosCon1y2 :: Int -> [Int]
numerosCon1y2 = map digitosAnumero . digitos

-- (dígitos n) es la lista ordenada de de listas de n elementos que
-- se pueden formar con los dígitos 1 y 2. Por ejemplo,
--    λ> digitos 2
--    [[1,1],[1,2],[2,1],[2,2]]
--    λ> digitos 3
--    [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
digitos :: Int -> [[Int]]
digitos 0 = [[]]
digitos n = map (1:) xss ++ map (2:) xss
  where xss = digitos (n-1)

-- (digitosAnumero ds) es el número cuyos dígitos son ds. Por ejemplo,
--    digitosAnumero [2,0,1,9]  ==  2019
digitosAnumero :: [Int] -> Int
digitosAnumero = read . concatMap show
```

```haskell
-- 2ª definición de numerosCon1y2
-- ===============================

numerosCon1y22 :: Int -> [Int]
numerosCon1y22 = map digitosAnumero . digitos2

-- (dígitos2 n) es la lista ordenada de las listas de n elementos que
-- se pueden formar con los dígitos 1 y 2. Por ejemplo,
--    λ> digitos2 2
--    [[1,1],[1,2],[2,1],[2,2]]
--    λ> digitos2 3
--    [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
digitos2 :: Int -> [[Int]]
digitos2 n = sucDigitos !! n

-- sucDigitos es la lista ordenada de las listas que se pueden formar
-- con los dígitos 1 y 2. Por ejemplo,
--    λ> take 4 sucDigitos
--    [[[]],
--     [[1],[2]],
--     [[1,1],[1,2],[2,1],[2,2]],
--     [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]]
sucDigitos :: [[[Int]]]
sucDigitos = iterate siguiente [[]]
  where siguiente xss = map (1:) xss ++ map (2:) xss

-- 3ª definición de numerosCon1y2
-- ===============================

numerosCon1y23 :: Int -> [Int]
numerosCon1y23 = map digitosAnumero . digitos2

-- (digitos3 n) es la lista ordenada de las listas de n elementos que
-- se pueden formar con los dígitos 1 y 2. Por ejemplo,
--    λ> digitos3 2
--    [[1,1],[1,2],[2,1],[2,2]]
--    λ> digitos3 3
--    [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
digitos3 :: Int -> [[Int]]
```

```haskell
digitos3 n = replicateM n [1,2]

-- Definición de restosNumerosCon1y2
-- =================================

restosNumerosCon1y2 :: Int -> [Int]
restosNumerosCon1y2 n =
  [x `mod` m | x <- numerosCon1y2 n]
  where m = 2^n

-- Definición de grafica_restosNumerosCon1y2
-- =========================================

grafica_restosNumerosCon1y2 :: Int -> IO ()
grafica_restosNumerosCon1y2 n =
  plotListStyle
    [ Key Nothing
    -- , PNG ("Numeros_con_digitos_1_y_2_" ++ show n ++ ".png")
    ]
    (defaultStyle {plotType = LinesPoints,
                   lineSpec = CustomStyle [PointType 7]})
    (restosNumerosCon1y2 n)

-- Propiedad de restosNumerosCon1y2
-- ================================

-- La propiedad
prop_restosNumerosCon1y2 :: Positive Int -> Bool
prop_restosNumerosCon1y2 (Positive n) =
  todosDistintos (restosNumerosCon1y2 n)
  where todosDistintos xs = xs == nub xs

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=12}) prop_restosNumerosCon1y2
--    +++ OK, passed 100 tests.
```

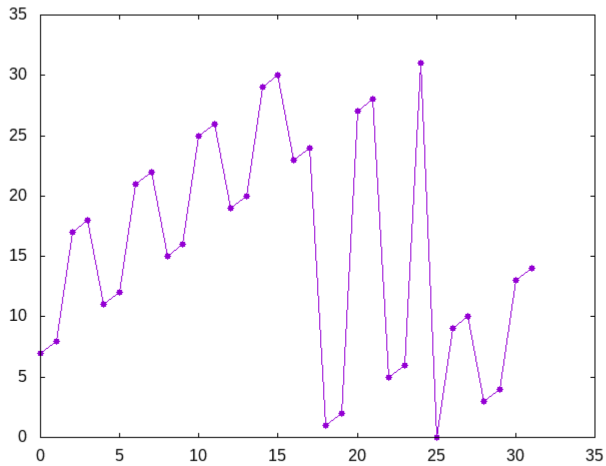


Figura 57.3: (graficaPrimosEnPi 10 10000)

Ejercicio 58

Árboles con n elementos

Ni vale nada el fruto
cogido sin sazón …
Ni aunque te elogie un bruto
ha de tener razón.

Antonio Machado

Enunciado

La árboles binarios se pueden representar con

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

Definir las funciones

```
arboles  :: Integer -> a -> [Arbol a]
nArboles :: [Integer]
```

tales que

- (arboles n x) es la lista de todos los árboles binarios con n elementos iguales a x. Por ejemplo,

```
λ> arboles 0 7
[]
λ> arboles 1 7
[H 7]
λ> arboles 2 7
[]
λ> arboles 3 7
[N 7 (H 7) (H 7)]
λ> arboles 4 7
[]
λ> arboles 5 7
[N 7 (H 7) (N 7 (H 7) (H 7)),N 7 (N 7 (H 7) (H 7)) (H 7)]
λ> arboles 6 7
[]
λ> arboles 7 7
[N 7 (H 7) (N 7 (H 7) (N 7 (H 7) (H 7))),
 N 7 (H 7) (N 7 (N 7 (H 7) (H 7)) (H 7)),
 N 7 (N 7 (H 7) (H 7)) (N 7 (H 7) (H 7)),
 N 7 (N 7 (H 7) (N 7 (H 7) (H 7))) (H 7),
 N 7 (N 7 (N 7 (H 7) (H 7)) (H 7)) (H 7)]
```

- nArboles es la sucesión de los números de árboles con k elementos iguales a 7, con $k \in \{1, 3, 5, \dots\}$. Por ejemplo,

```
λ> take 14 nArboles
[1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012,742900]
λ> nArboles !! 100
896519947090131496687170070074100632420837521538745909320
λ> length (show (nArboles !! 1000))
598
```

Soluciones

```
import Data.List (genericLength)

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

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

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

-- 2ª definición de arboles
-- ========================

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

-- 1ª definición de nArboles
-- =========================

nArboles :: [Integer]
nArboles = [genericLength (arboles2 n 7) | n <- [1,3..]]

-- 2ª definición de nArboles
-- =========================

-- Con la definición anterior se observa que nArboles es
--    1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012,742900
-- son los números de Catalan https://en.wikipedia.org/wiki/Catalan_number
-- Una forma de calcularlos (ver https://oeis.org/A000108) es
--     (2n)!/(n!(n+1)!)

nArboles2 :: [Integer]
nArboles2 =
  [factorial (2*n) `div` (factorial n * factorial (n+1)) | n <- [0..]]
```

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

-- 3ª definición de nArboles
-- =========================

-- 1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012,742900

--  1
--  1 = 1*1
--  2 = 1*1  + 1*1
--  5 = 1*2  + 1*1 + 2*1
-- 14 = 1*5  + 1*2 + 2*1 + 5*1
-- 42 = 1*14 + 1*5 + 2*2 + 5*1 + 14*1

nArboles3 :: [Integer]
nArboles3 = 1 : aux [1]
  where aux cs = c : aux (c:cs)
          where c = sum (zipWith (*) cs (reverse cs))

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

--    λ> length (show (nArboles !! 12))
--    6
--    (6.50 secs, 1,060,563,128 bytes)
--    λ> length (show (nArboles2 !! 12))
--    6
--    (0.01 secs, 108,520 bytes)
--    λ> length (show (nArboles3 !! 12))
--    6
--    (0.01 secs, 119,096 bytes)
--
--    λ> length (show (nArboles2 !! 1000))
--    598
--    (0.01 secs, 4,796,440 bytes)
--    λ> length (show (nArboles3 !! 1000))
--    598
--    (1.66 secs, 321,771,704 bytes)
```

Ejercicio 59

Impares en filas del triángulo de Pascal

De lo que llaman los hombres
virtud, justicia y bondad,
una mitad es envidia,
y la otra no es caridad.

Antonio Machado

Enunciado

El triángulo de Pascal es un triángulo de números

```
        1
       1 1
      1 2 1
    1  3 3  1
   1 4  6  4 1
  1 5 10 10 5 1
 ...............
```

construido de la siguiente forma

- la primera fila está formada por el número 1;

- las filas siguientes se construyen sumando los números adyacentes de la fila superior y añadiendo un 1 al principio y al final de la fila.

Definir las funciones

```
imparesPascal           :: [[Integer]]
nImparesPascal          :: [Int]
grafica_nImparesPascal :: Int -> IO ()
```

tales que

- imparesPascal es la lista de los elementos impares en cada una de las filas del triángulo de Pascal. Por ejemplo,

```
λ> take 8 imparesPascal
[[1],
 [1,1],
 [1,1],
 [1,3,3,1],
 [1,1],
 [1,5,5,1],
 [1,15,15,1],
 [1,7,21,35,35,21,7,1]]
```

- nImparesPascal es la lista del número de elementos impares en cada una de las filas del triángulo de Pascal. Por ejemplo,

```
λ> take 30 nImparesPascal
[1,2,2,4,2,4,4,8,2,4,4,8,4,8,8,16,2,4,4,8,4,8,8,16,4,8,8,16,8,16]
λ> maximum (take (10^6) nImparesPascal)
524288
```

- `(grafica_nImparesPascal n)` dibuja la gráfica de los n primeros términos de nImparesPascal. Por ejemplo,
  - `(grafica_nImparesPascal 50)` dibuja la Figura 59.1
  - `(grafica_nImparesPascal 100)` dibuja la Figura 59.2

Comprobar con QuickCheck que todos los elementos de nImparesPascal son potencias de dos.

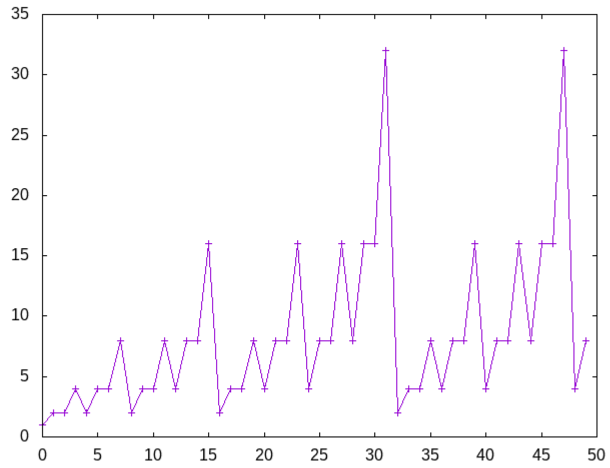


Figura 59.1: (grafica_nImparesPascal 50)

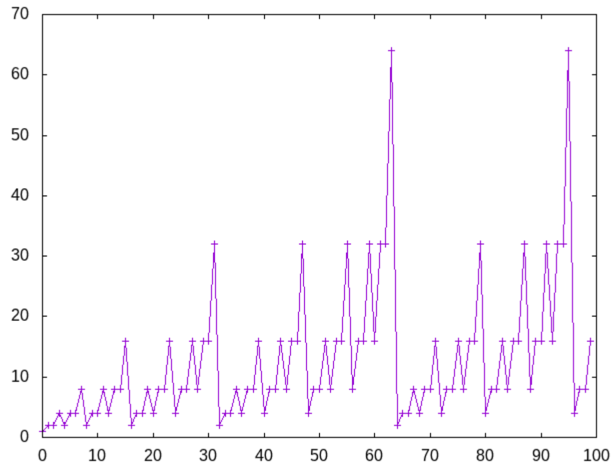


Figura 59.2: (grafica_nImparesPascal 100)

Soluciones

```haskell
import Data.List (transpose)
import Test.QuickCheck
import Graphics.Gnuplot.Simple

-- 1ª definición de imparesPascal
-- ===============================

imparesPascal :: [[Integer]]
imparesPascal =
  map (filter odd) pascal

-- pascal es la lista de las filas del triángulo de Pascal. Por ejemplo,
--    λ> take 7 pascal
--    [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1],[1,6,15,20,15,6,1]
pascal :: [[Integer]]
pascal = [1] : map f pascal
  where f xs = zipWith (+) (0:xs) (xs++[0])

-- 2ª definición de imparesPascal
-- ===============================

imparesPascal2 :: [[Integer]]
imparesPascal2 =
  map (filter odd) pascal

pascal2 :: [[Integer]]
pascal2 = iterate f [1]
  where f xs = zipWith (+) (0:xs) (xs++[0])

-- 1ª definición de nImparesPascal
-- ================================

nImparesPascal :: [Int]
nImparesPascal =
  map length imparesPascal

-- 2ª definición de nImparesPascal
-- ================================
```

```haskell
nImparesPascal2 :: [Int]
nImparesPascal2 =
  map (length . filter odd) imparesPascal

-- 3ª definición de nImparesPascal
-- ===============================

--    λ> take 32 nImparesPascal2
--    [1,2,
--     2,4,
--     2,4,4,8,
--     2,4,4,8,4,8,8,16,
--     2,4,4,8,4,8,8,16,4,8,8,16,8,16,16,32]
nImparesPascal3 :: [Int]
nImparesPascal3 = 1 : zs
  where zs = 2 : concat (transpose [zs, map (*2) zs])

-- Definición de grafica_nImparesPascal
-- =========================================

grafica_nImparesPascal :: Int -> IO ()
grafica_nImparesPascal n =
  plotListStyle
    [ Key Nothing
    , PNG ("Impares_en_filas_del_triangulo_de_Pascal_" ++ show n ++ ".png")
    ]
    (defaultStyle {plotType = LinesPoints})
    (take n nImparesPascal3)

-- Propiedad de nImparesPascal
-- ===========================

-- La propiedad es
prop_nImparesPascal :: Positive Int -> Bool
prop_nImparesPascal (Positive n) =
  esPotenciaDeDos (nImparesPascal3 !! n)

-- (esPotenciaDeDos n) se verifica si n es una potencia de dos. Por
-- ejemplo,
```

```
--    esPotenciaDeDos 16  ==  True
--    esPotenciaDeDos 18  ==  False
esPotenciaDeDos :: Int -> Bool
esPotenciaDeDos 1 = True
esPotenciaDeDos n = even n && esPotenciaDeDos (n `div` 2)

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

Ejercicio 60

Triángulo de Pascal binario

La envidia de la virtud
hizo a Caín criminal.
¡Gloria a Caín! Hoy el vicio
es lo que se envidia más.

Antonio Machado

Enunciado

Los triángulos binarios de Pascal se forman a partir de una lista de ceros y unos usando las reglas del triángulo de Pascal, donde cada uno de los números es suma módulo dos de los dos situados en diagonal por encima suyo. Por ejemplo, los triángulos binarios de Pascal correspondientes a [1,0,1,1,1] y [1,0,1,1,0] son

```
1 0 1 1 1   1 0 1 1 0
 1 1 0 0     1 1 0 1
  0 1 0       0 1 1
   1 1         1 0
    0           1
```

Sus finales, desde el extremo inferior al extremos superior derecho, son [0,1,0,0,1] y [1,0,1,1,0], respectivamente.

Una lista es Pascal capicúa si es igual a los finales de su triángulo binario de Pascal. Por ejemplo, [1,0,1,1,0] es Pascal capicúa.

Definir las funciones

```
trianguloPascalBinario :: [Int] -> [[Int]]
pascalCapicuas         :: Int -> [[Int]]
nPascalCapicuas        :: Int -> Integer
```

tales que

- (trianguloPascalBinario xs) es el triágulo binario de Pascal correspondiente a la lista xs. Por ejemplo,

```
λ> trianguloPascalBinario [1,0,1,1,1]
[[1,0,1,1,1],[1,1,0,0],[0,1,0],[1,1],[0]]
λ> trianguloPascalBinario [1,0,1,1,0]
[[1,0,1,1,0],[1,1,0,1],[0,1,1],[1,0],[1]]
```

- (pascalCapicuas n) es la lista de listas de Pascal capicúas de n elementos. Por ejemplo,

```
λ> pascalCapicuas 2
[[0,0],[1,0]]
λ> pascalCapicuas 3
[[0,0,0],[0,1,0],[1,0,0],[1,1,0]]
λ> pascalCapicuas 4
[[0,0,0,0],[0,1,1,0],[1,0,0,0],[1,1,1,0]]
```

- (nPascalCapicuas n) es el número de listas de Pascal capicúas de n elementos. Por ejemplo,

```
λ> nPascalCapicuas 2
2
λ> nPascalCapicuas 3
4
λ> nPascalCapicuas 4
4
λ> nPascalCapicuas 400
1606938044258990275541962092341162602522202993782792835301376
λ> length (show (nPascalCapicuas (10^5)))
15052
λ> length (show (nPascalCapicuas (10^6)))
```

```
150515
λ> length (show (nPascalCapicuas (10^7)))
1505150
```

Soluciones

```haskell
import Data.List (genericLength, unfoldr)
import Control.Monad (replicateM)

-- Definición de trianguloPascalBinario
-- ====================================

trianguloPascalBinario :: [Int] -> [[Int]]
trianguloPascalBinario xs =
  takeWhile (not . null) (iterate siguiente xs)

-- (siguiente xs) es la línea siguiente a la xs en el triángulo binario
-- de Pascal. Por ejemplo,
--    λ> siguiente [1,0,1,1,1]
--    [1,1,0,0]
--    λ> siguiente it
--    [0,1,0]
--    λ> siguiente it
--    [1,1]
--    λ> siguiente it
--    [0]
--    λ> siguiente it
--    []
--    λ> siguiente it
--    []
siguiente :: [Int] -> [Int]
siguiente xs = [(x + y) `mod` 2 | (x,y) <- zip xs (tail xs)]

-- 2ª definición de trianguloPascalBinario
-- =======================================

trianguloPascalBinario2 :: [Int] -> [[Int]]
trianguloPascalBinario2 = unfoldr f
```

```haskell
  where f [] = Nothing
        f xs = Just (xs, siguiente xs)

-- Definición de pascalCapicuas
-- ============================

pascalCapicuas :: Int -> [[Int]]
pascalCapicuas n =
  [xs | xs <- inicios n
      , esPascalCapicua xs]

-- (inicios n) es la lista de longitud n formadas por ceros y unos. Por
-- ejemplo,
--    λ> inicios 0
--    [[]]
--    λ> inicios 1
--    [[0],[1]]
--    λ> inicios 2
--    [[0,0],[0,1],[1,0],[1,1]]
--    λ> inicios 3
--    [[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]
inicios :: Int -> [[Int]]
inicios 0 = [[]]
inicios n = map (0:) xss ++ map (1:) xss
  where xss = inicios (n-1)

-- Otra forma de definir inicios es
inicios2 :: Int -> [[Int]]
inicios2 n = sucInicios !! n
  where sucInicios     = iterate siguiente' [[]]
        siguiente' xss = map (0:) xss ++ map (1:) xss

-- Y otra es
inicios3 :: Int -> [[Int]]
inicios3 n = replicateM n [0,1]

-- (esPascalCapicua xs) se verifica si xs es una lista de Pascal
-- capicúa. Por ejemplo,
--    esPascalCapicua [1,0,1,1,0]  ==  True
--    esPascalCapicua [1,0,1,1,1]  ==  False
```

```haskell
esPascalCapicua :: [Int] -> Bool
esPascalCapicua xs =
  xs == finalesTrianguloPascalBinario xs

-- (finalesTrianguloPascalBinario xs) es la inversa de la lista de los
-- finales del triángulo binarios de xs. Por ejemplo,
--    λ> finalesTrianguloPascalBinario [1,0,1,1,1]
--    [0,1,0,0,1]
finalesTrianguloPascalBinario :: [Int] -> [Int]
finalesTrianguloPascalBinario =
  reverse . map last . trianguloPascalBinario

-- 1ª definición de nPascalCapicuas
-- ================================

nPascalCapicuas :: Int -> Integer
nPascalCapicuas =
  genericLength . pascalCapicuas

-- 2ª definición de nPascalCapicuas
-- ================================

nPascalCapicuas2 :: Int -> Integer
nPascalCapicuas2 n =
  2 ^ ((n + 1) `div` 2)
```

Ejercicio 61

Dígitos en las posiciones pares de cuadrados

¡Ojos que a la luz se abrieron
un día para, después,
ciegos tornar a la tierra,
hartos de mirar sin ver.

Antonio Machado

Enunciado

Definir las funciones

```
digitosPosParesCuadrado    :: Integer -> ([Integer],Int)
invDigitosPosParesCuadrado :: ([Integer],Int) -> [Integer]
```

tales que

- (digitosPosParesCuadrado n) es el par formados por los dígitos de n² en la posiciones pares y por el número de dígitos de n². Por ejemplo,

```
digitosPosParesCuadrado 8     ==  ([6],2)
digitosPosParesCuadrado 14    ==  ([1,6],3)
digitosPosParesCuadrado 36    ==  ([1,9],4)
```

```
digitosPosParesCuadrado 116   ==  ([1,4,6],5)
digitosPosParesCuadrado 2019  ==  ([4,7,3,1],7)
```

- (invDigitosPosParesCuadrado (xs,k)) es la lista de los números n tales que xs es la lista de los dígitos de n² en la posiciones pares y k es el número de dígitos de n². Por ejemplo,

```
invDigitosPosParesCuadrado ([6],2)               ==  [8]
invDigitosPosParesCuadrado ([1,6],3)             ==  [14]
invDigitosPosParesCuadrado ([1,9],4)             ==  [36]
invDigitosPosParesCuadrado ([1,4,6],5)           ==  [116,136]
invDigitosPosParesCuadrado ([4,7,3,1],7)         ==  [2019,2139,2231]
invDigitosPosParesCuadrado ([1,2],3)             ==  []
invDigitosPosParesCuadrado ([1,2],4)             ==  [32,35,39]
invDigitosPosParesCuadrado ([1,2,3,4,5,6],11)  ==  [115256,127334,135254]
```

Comprobar con QuickCheck que para todo entero positivo n se verifica que para todo entero positivo m, m pertenece a (invDigitosPosParesCuadrado (digitosPosParesCuadrado n)) si, y sólo si, (digitosPosParesCuadrado m) es igual a (digitosPosParesCuadrado n)

Soluciones

```
import Test.QuickCheck

-- Definición de digitosPosParesCuadrado
-- =====================================

digitosPosParesCuadrado :: Integer -> ([Integer],Int)
digitosPosParesCuadrado n =
  (digitosPosPares (n^2),length (show (n^2)))

-- (digitosPosPares n) es la lista de los dígitos de n en posiciones
-- pares. Por ejemplo,
--    digitosPosPares 24012019  ==  [2,0,2,1]
digitosPosPares :: Integer -> [Integer]
digitosPosPares n = elementosPosPares (digitos n)

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
```

```
--    digitos 325  ==  [3,2,5]
digitos :: Integer -> [Integer]
digitos n = [read [c] | c <- show n]

-- (elementosPosPares xs) es la lista de los elementos de xs en
-- posiciones pares. Por ejemplo,
--    elementosPosPares [3,2,5,7,6,4]  ==  [3,5,6]
elementosPosPares :: [a] -> [a]
elementosPosPares []       = []
elementosPosPares [x]      = [x]
elementosPosPares (x:_:zs) = x : elementosPosPares zs

-- 1ª definición de invDigitosPosParesCuadrado
-- ===========================================

invDigitosPosParesCuadrado :: ([Integer],Int) -> [Integer]
invDigitosPosParesCuadrado (xs, a) =
  [x | x <- [ceiling (sqrt 10^(a-1))..ceiling (sqrt 10^a)]
     , digitosPosParesCuadrado x == (xs,a)]

-- 2ª definición de invDigitosPosParesCuadrado
-- ===========================================

invDigitosPosParesCuadrado2 :: ([Integer],Int) -> [Integer]
invDigitosPosParesCuadrado2 x =
  [n | n <- [a..b], digitosPosParesCuadrado n == x]
  where a = floor (sqrt (fromIntegral (completaNum x 0)))
        b = ceiling (sqrt (fromIntegral (completaNum x 9)))

-- (completaNum (xs,k) n) es el número cuyos dígitos en las posiciones
-- pares son los de xs y los de las posiciones impares son iguales a n
-- (se supone que k es igual al doble de la longitud de xs o un
-- menos). Por ejemplo,
--    completaNum ([1,3,8],5) 4  ==  14348
--    completaNum ([1,3,8],6) 4  ==  143484
completaNum :: ([Integer],Int) -> Integer -> Integer
completaNum x n = digitosAnumero (completa x n)

-- (completa (xs,k) n) es la lista cuyos elementos en las posiciones
-- pares son los de xs y los de las posiciones impares son iguales a n
```

```haskell
-- (se supone que k es igual al doble de la longitud de xs o un
-- menos). Por ejemplo,
--    completa ([1,3,8],5) 4  ==  [1,4,3,4,8]
--    completa ([1,3,8],6) 4  ==  [1,4,3,4,8,4]
completa :: ([Integer],Int) -> Integer -> [Integer]
completa (xs,k) n
  | even k    = ys
  | otherwise = init ys
  where ys = concat [[x,n] | x <- xs]

-- (digitosAnumero ds) es el número cuyos dígitos son ds. Por ejemplo,
--    digitosAnumero [2,0,1,9]  ==  2019
digitosAnumero :: [Integer] -> Integer
digitosAnumero = read . concatMap show

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

--    λ> invDigitosPosParesCuadrado ([1,2,1,5,7,4,9],13)
--    [1106393,1234567,1314597]
--    (7.55 secs, 13,764,850,536 bytes)
--    λ> invDigitosPosParesCuadrado2 ([1,2,1,5,7,4,9],13)
--    [1106393,1234567,1314597]
--    (1.96 secs, 3,780,368,816 bytes)

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_digitosPosParesCuadrado :: Positive Integer -> Positive Integer -> Bool
prop_digitosPosParesCuadrado (Positive n) (Positive m) =
  (digitosPosParesCuadrado m == x)
  == (m `elem` invDigitosPosParesCuadrado x)
  where x = digitosPosParesCuadrado n

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

Ejercicio 62

Límites de sucesiones

De diez cabezas, nueve
embisten y una piensa.
Nunca extrañéis que un bruto
se descuerne luchando por la idea.

Antonio Machado

Enunciado

El límite de una sucesión, con una aproximación a y una amplitud n, es el primer término x de la sucesión tal que el valor absoluto de x y cualquiera de sus n siguentes elementos es menor que a.

Definir la función

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

tal que (limite xs a n) es el límite de xs xon aproximación a y amplitud n. Por ejemplo,

```
λ> limite [(2*n+1)/(n+5) | n <- [1..]] 0.001 300
1.9939919893190920
λ> limite [(2*n+1)/(n+5) | n <- [1..]] 1e-6 300
1.9998260062637745
λ> limite [(1+1/n)**n | n <- [1..]] 0.001 300
2.7155953364173175
```

Soluciones

```haskell
import Data.List (tails)

-- 1ª solución
-- ===========

limite :: [Double] -> Double -> Int -> Double
limite (n:ns) x a
  | abs (n - maximum (take (a-1) ns)) < x = n
  | otherwise                             = limite ns x a

-- 2ª solución
-- ===========

limite2 :: [Double] -> Double -> Int -> Double
limite2 xs a n =
  head [ x | (x:ys) <- segmentos xs n
       , all (\y ->  abs (y - x) < a) ys]

-- (segmentos xs n) es la lista de los segmentos de la lista infinita xs
-- con n elementos. Por ejemplo,
--    λ> take 5 (segmentos [1..] 3)
--    [[1,2,3],[2,3,4],[3,4,5],[4,5,6],[5,6,7]]
segmentos :: [a] -> Int -> [[a]]
segmentos xs n = map (take n) (tails xs)

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

--    λ> limite [(1+1/n)**n | n <- [1..]] 1e-8 100
--    2.7182700737511185
--    (2.01 secs, 1,185,073,864 bytes)
--    λ> limite2 [(1+1/n)**n | n <- [1..]] 1e-8 100
--    2.7182700737511185
--    (5.03 secs, 1,044,694,264 bytes)
```

Ejercicio 63

Medias de dígitos de pi

Es el mejor de los buenos
quien sabe que en esta vida
todo es cuestión de medida:
un poco más, algo menos.

Antonio Machado

Enunciado

El fichero Digitos_de_pi.txt contiene el número pi con un millón de decimales; es decir,

$$3{,}14159265358979323846264338 32..{,}83996346460422090106105779458151$$

Definir las funciones

```
mediasDigitosDePi         :: IO [Double]
graficaMediasDigitosDePi :: Int -> IO ()
```

tales que

- mediasDigitosDePi es la sucesión cuyo n–ésimo elemento es la media de los n primeros dígitos de pi. Por ejemplo,

```
λ> xs <- mediasDigitosDePi
λ> take 5 xs
```

```
[1.0,2.5,2.0,2.75,4.0]
λ> take 10 xs
[1.0,2.5,2.0,2.75,4.0,3.6666666666666665,4.0,4.125,4.0,4.1]
λ> take 10 <$> mediasDigitosDePi
[1.0,2.5,2.0,2.75,4.0,3.6666666666666665,4.0,4.125,4.0,4.1]
```

- (graficaMediasDigitosDePi n) dibuja la gráfica de los n primeros términos de mediasDigitosDePi. Por ejemplo,
    - (graficaMediasDigitosDePi 20) dibuja la Figura 63.1

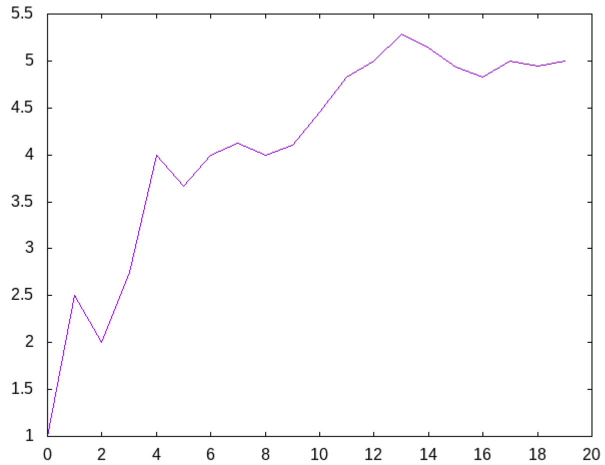


Figura 63.1: (graficaMediasDigitosDePi 20)

- (graficaMediasDigitosDePi 200) dibuja la Figura 63.2
- (graficaMediasDigitosDePi 2000) dibuja la Figura 63.3

Soluciones

```
import Data.Char (digitToInt)
import Data.List (genericLength, inits)
import Graphics.Gnuplot.Simple ( Attribute (Key, PNG)
                               , plotList )

-- Definición de mediasDigitosDePi
-- ===============================
```

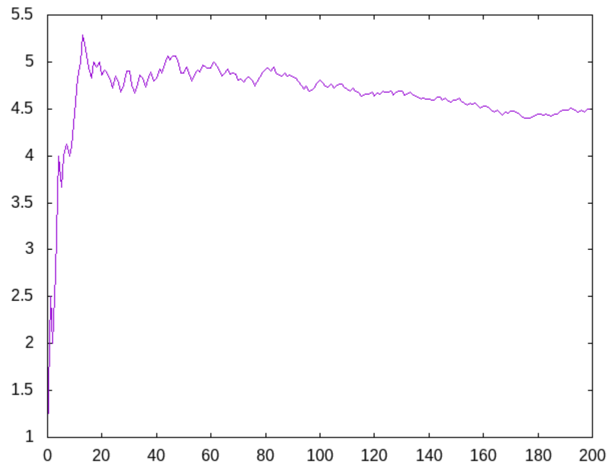


Figura 63.2: (graficaMediasDigitosDePi 200)

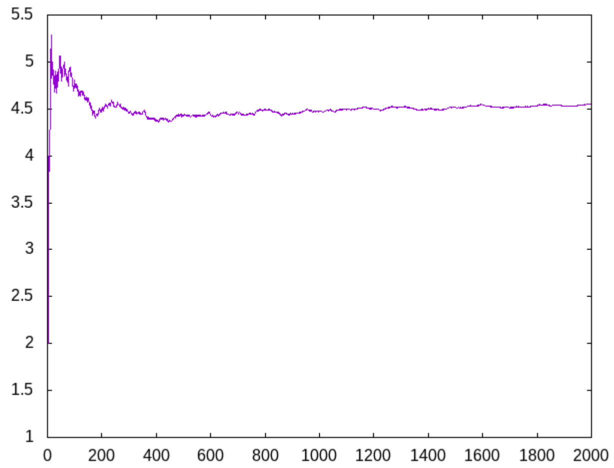


Figura 63.3: (graficaMediasDigitosDePi 2000)

```
mediasDigitosDePi :: IO [Double]
mediasDigitosDePi = do
  (_:_:ds) <- readFile "Digitos_de_pi.txt"
  return (medias (digitos ds))

-- (digitos cs) es la lista de los digitos de cs. Por ejempplo,
--    digitos "1415926535"  ==  [1,4,1,5,9,2,6,5,3,5]
digitos :: String -> [Int]
digitos = map digitToInt

-- (medias xs) es la lista de las medias de los segmentos iniciales de
-- xs. Por ejemplo,
--    λ> medias [1,4,1,5,9,2,6,5,3,5]
--    [1.0,2.5,2.0,2.75,4.0,3.6666666666666665,4.0,4.125,4.0,4.1]
medias :: [Int] -> [Double]
medias xs = map media (tail (inits xs))

-- (media xs) es la media aritmética de xs. Por ejemplo,
--    media [1,4,1,5,9]  ==  4.0
media :: [Int] -> Double
media xs = fromIntegral (sum xs) / genericLength xs

-- Definición de graficaMediasDigitosDePi
-- ======================================

graficaMediasDigitosDePi :: Int -> IO ()
graficaMediasDigitosDePi n = do
  xs <- mediasDigitosDePi
  plotList [ Key Nothing
           , PNG ("Medias_de_digitos_de_pi_" ++ show n ++ ".png")
           ]
           (take n xs)
```

Ejercicio 64

Exterior de árboles

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

Antonio Machado

Enunciado

Los árboles binarios con datos en las hojas y los nodos se definen por

```haskell
data Arbol = H Int
           | N Int Arbol Arbol
  deriving Show
```

Por ejemplo, los árboles

```
         3                3                3
        / \              / \              / \
       /   \            /   \            /   \
      2     8          2     8          2     8
     / \   / \        / \   / \        / \   / \
    5   7 6   9      5   7 6   9      5   7 6   9
   / \                    / \                  / \
  1   4                  1   4                1   4
```

se representan por

```
ejArbol1, ejArbol2, ejArbol3 :: Arbol
ejArbol1 = N 3
             (N 2
                (N 5 (H 1) (H 4))
                (H 7))
             (N 8 (H 6) (H 9))
ejArbol2 = N 3
             (N 2 (H 5) (H 7))
             (N 8 (N 6 (H 1) (H 4))
                  (H 9))
ejArbol3 = N 3
             (N 2 (H 5) (H 7))
             (N 8 (H 6)
                  (N 9 (H 1) (H 4)))
```

Definir la función

```
exterior :: Arbol -> [Int]
```

tal que (exterior a) es la lista de los elementos exteriores del árbol a. Por ejemplo,

```
exterior ejArbol1  ==  [3,2,5,1,4,7,6,9,8]
exterior ejArbol2  ==  [3,2,5,7,1,4,9,8]
exterior ejArbol3  ==  [3,2,5,7,6,1,4,9,8]
```

El orden de los elementos es desde la raíz hasta el extremo inferior izquierdo desde él hasta el inferior derecho y desde él hasta la raíz.

Soluciones

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

ejArbol1, ejArbol2, ejArbol3 :: Arbol
```

```haskell
ejArbol1 = N 3
             (N 2
                (N 5 (H 1) (H 4))
                (H 7))
             (N 8 (H 6) (H 9))
ejArbol2 = N 3
             (N 2 (H 5) (H 7))
             (N 8 (N 6 (H 1) (H 4))
                  (H 9))
ejArbol3 = N 3
             (N 2 (H 5) (H 7))
             (N 8 (H 6)
                  (N 9 (H 1) (H 4)))

exterior :: Arbol -> [Int]
exterior (H x) = [x]
exterior a =
  ramaIzquierda a ++ hojas a ++ reverse (tail (ramaDerecha a))

-- (ramaIzquierda a) es la rama izquierda del árbol a. Por ejemplo,
--    ramaIzquierda ejArbol1  ==  [3,2,5]
--    ramaIzquierda ejArbol3  ==  [3,2]
ramaIzquierda :: Arbol -> [Int]
ramaIzquierda (H _)     = []
ramaIzquierda (N x i _) = x : ramaIzquierda i

-- (ramaDerecha a) es la rama derecha del árbol a. Por ejemplo,
--    ramaDerecha ejArbol1  ==  [3,8]
--    ramaDerecha ejArbol3  ==  [3,8,9]
ramaDerecha :: Arbol -> [Int]
ramaDerecha (H _)     = []
ramaDerecha (N x _ d) = x : ramaDerecha d

-- (hojas a) es la lista de las hojas del árbol a. Por ejemplo,
--    hojas ejArbol1  ==  [1,4,7,6,9]
--    hojas ejArbol3  ==  [5,7,6,1,4]
hojas :: Arbol -> [Int]
hojas (H x)     = [x]
hojas (N _ i d) = hojas i ++ hojas d
```

Ejercicio 65

Aritmética lunar

Cantad conmigo en coro: saber, nada sabemos,
de arcano mar vinimos, a ignota mar iremos ...
La luz nada ilumina y el sabio nada enseña.
¿Qué dice la palabra? ¿Qué el agua de la peña?

Antonio Machado

Enunciado

En la [aritmética lunar](http://bit.ly/2SaE9ZH) la suma y el producto se hace como en la terrícola salvo que sus tablas de sumar y de multiplicar son distintas. La suma lunar de dos dígitos es su máximo (por ejemplo, 1 + 3 = 3 y 7 + 4 = 7) y el producto lunar de dos dígitos es su mínimo (por ejemplo, 1 x 3 = 1 y 7 x 4 = 4). Por tanto,

```
  3 5 7        3 5 7
+   6 4      *   6 4
-------      -------
  3 6 7        3 4 4
             3 5 6
             -------
             3 5 6 4
```

Definir las funciones

```
suma     :: Integer -> Integer -> Integer
producto :: Integer -> Integer -> Integer
```

tales que

+ (suma x y) es la suma lunar de x e y. Por ejemplo,

```
suma 357 64  ==  367
suma 64 357  ==  367
suma 1 3     ==  3
suma 7 4     ==  7
```

+ (producto x y) es el producto lunar de x e y. Por ejemplo,

```
producto 357 64  ==  3564
producto 64 357  ==  3564
producto 1 3     ==  1
producto 7 4     ==  4
```

Comprobar con QuickCheck que la suma y el producto lunar son conmutativos.

Soluciones

```
import Test.QuickCheck

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

producto :: Integer -> Integer -> Integer
producto 0 _ = 0
producto x y = productoDigitoNumero x2 y `suma` (10 * producto x1 y)
  where (x1, x2) = x `divMod` 10

-- (productoDigitoNumero d x) es el producto del dígito d por el número
```

```
-- x. Por ejemplo,
--    productoDigitoNumero 4 357  ==  344
--    productoDigitoNumero 6 357  ==  356
productoDigitoNumero :: Integer -> Integer -> Integer
productoDigitoNumero _ 0 = 0
productoDigitoNumero d x = min d x2 + 10 * productoDigitoNumero d x1
  where (x1, x2) = x `divMod` 10

-- La propiedad es
prop_conmutativa :: Positive Integer -> Positive Integer -> Bool
prop_conmutativa (Positive x) (Positive y) =
  suma x y == suma y x &&
  producto x y == producto y x

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

Ejercicio 66

Término ausente en una progresión aritmética

¡Y esa gran placentería
de ruiseñores que cantan!
Ninguna voz es la mía.

Antonio Machado

Enunciado

Una progresión aritmética es una sucesión de números tales que la diferencia de dos términos sucesivos cualesquiera de la sucesión es constante.

Definir la función

```
ausente :: Integral a => [a] -> a
```

tal que (ausente xs) es el único término ausente de la progresión aritmética xs. Por ejemplo,

```
ausente [3,7,9,11]                ==  5
ausente [3,5,9,11]                ==  7
ausente [3,5,7,11]                ==  9
ausente ([1..9]++[11..])          ==  10
ausente ([1..10^6] ++ [2+10^6])   ==  1000001
```

Nota. Se supone que la lista tiene al menos 3 elementos, que puede ser infinita y que sólo hay un término de la progresión aritmética que no está en la lista.

Soluciones

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

-- 1ª solución
ausente :: Integral a => [a] -> a
ausente (x1:xs@(x2:x3:_))
  | d1 == d2     = ausente xs
  | d1 == 2 * d2 = x1 + d2
  | d2 == 2 * d1 = x2 + d1
  where d1 = x2 - x1
        d2 = x3 - x2

-- 2ª solución
ausente2 :: Integral a => [a] -> a
ausente2 s@(x1:x2:x3:_)
  | x1 + x3 /= 2 * x2 = x1 + (x3 - x2)
  | otherwise         = head [a | (a,b) <- zip [x1,x2..] s
                                , a /= b]

-- 3ª solución
ausente3 :: Integral a => [a] -> a
ausente3  xs@(x1:x2:_)
  | null us   = x1 + v
  | otherwise = x2 + u * genericLength (u:us)
  where ((u:us):(v:_):_) = group (zipWith (-) (tail xs) xs)

-- Comparación de eficiencia
--    ghci> let n = 10^6 in ausente1 ([1..n] ++ [n+2])
--    1000001
--    (3.53 secs, 634729880 bytes)
--
--    ghci> let n = 10^6 in ausente2 ([1..n] ++ [n+2])
--    1000001
--    (0.86 secs, 346910784 bytes)
```

```
--
--    ghci> let n = 10^6 in ausente3 ([1..n] ++ [n+2])
--    1000001
--    (1.22 secs, 501521888 bytes)
--
--    ghci> let n = 10^7 in ausente2 ([1..n] ++ [n+2])
--    10000001
--    (8.68 secs, 3444142568 bytes)
--
--    ghci> let n = 10^7 in ausente3 ([1..n] ++ [n+2])
--    10000001
--    (12.59 secs, 4975932088 bytes)
```

Ejercicio 67

Particiones de enteros positivos

Fe empirista. Ni somos ni seremos.
Todo nuestro vivir es emprestado.
Nada trajimos, nada llevaremos.

Antonio Machado

Enunciado

Una partición de un entero positivo n es una manera de escribir n como una suma de enteros positivos. Dos sumas que sólo difieren en el orden de sus sumandos se consideran la misma partición. Por ejemplo, 4 tiene cinco particiones: 4, 3+1, 2+2, 2+1+1 y 1+1+1+1.

Definir la función

```
particiones :: Int -> [[Int]]
```

tal que (particiones n) es la lista de las particiones del número n. Por ejemplo,

```
particiones 4  ==  [[4],[3,1],[2,2],[2,1,1],[1,1,1,1]]
particiones 5  ==  [[5],[4,1],[3,2],[3,1,1],[2,2,1],[2,1,1,1],[1,1,1,1,1]]
length (particiones 50)  ==  204226
```

Soluciones

```haskell
-- 1ª solución
particiones :: Int -> [[Int]]
particiones 0 = [[]]
particiones n = [x:y | x <- [n,n-1..1],
                       y <- particiones (n-x),
                       [x] >= take 1 y]

-- 2ª solución
particiones2 :: Int -> [[Int]]
particiones2 n = aux !! n where
  aux = [] : map particiones' [1..]
  particiones' n' = [n'] : [x:p | x <- [n',n'-1..1],
                                  p <- aux !! (n'-x),
                                  x >= head p]

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

-- La comparación es
--    ghci> length (particiones 20)
--    627
--    (4.93 secs, 875288184 bytes)
--
--    ghci> length (particiones2 20)
--    627
--    (0.02 secs, 2091056 bytes)
```

Ejercicio 68

Sucesión de sumas de dos números abundantes

¿Dices que nada se crea?
Alfarero, a tus cacharros.
Haz tu copa y no te importe
si no puedes hacer barro.

Antonio Machado

Enunciado

Un número n es abundante si la suma de los divisores propios de n es mayor que n. El primer número abundante es el 12 (cuyos divisores propios son 1, 2, 3, 4 y 6 cuya suma es 16). Por tanto, el menor número que es la suma de dos números abundantes es el 24.

Definir la sucesión

```
sumasDeDosAbundantes :: [Integer]
```

cuyos elementos son los números que se pueden escribir como suma de dos números abundantes. Por ejemplo,

```
take 10 sumasDeDosAbundantes  ==  [24,30,32,36,38,40,42,44,48,50]
```

Soluciones

```haskell
sumasDeDosAbundantes :: [Integer]
sumasDeDosAbundantes = [n | n <- [1..], esSumaDeDosAbundantes n]

-- (esSumaDeDosAbundantes n) se verifica si n es suma de dos números
-- abundantes. Por ejemplo,
--    esSumaDeDosAbundantes 24          ==  True
--    any esSumaDeDosAbundantes [1..22]  ==  False
esSumaDeDosAbundantes :: Integer -> Bool
esSumaDeDosAbundantes n = (not . null) [x | x <- xs, n-x `elem` xs]
  where xs = takeWhile (<n) abundantes

-- abundantes es la lista de los números abundantes. Por ejemplo,
--    take 10 abundantes  ==  [12,18,20,24,30,36,40,42,48,54]
abundantes :: [Integer]
abundantes = [n | n <- [2..], abundante n]

-- (abundante n) se verifica si n es abundante. Por ejemplo,
--    abundante 12  ==  True
--    abundante 11  ==  False
abundante :: Integer -> Bool
abundante n = sum (divisores n) > n

-- (divisores n) es la lista de los divisores propios de n. Por ejemplo,
--    divisores 12  ==  [1,2,3,4,6]
divisores :: Integer -> [Integer]
divisores n = [x | x <- [1..n `div` 2], n `mod` x == 0]
```

Ejercicio 69

Cálculo de pi mediante la serie de Nilakantha

Bueno es saber que los vasos
nos sirven para beber;
lo malo es que no sabemos
para que sirve la sed.

Antonio Machado

Enunciado

Una serie infinita para el cálculo de pi, publicada por Nilakantha en el siglo XV, es

$$\pi = 3 + \frac{4}{2\times 3\times 4} - \frac{4}{4\times 5\times 6} + \frac{4}{6\times 7\times 8} - \frac{4}{8\times 8\times 10} + \dots$$

Definir las funciones

```
aproximacionPi :: Int -> Double
tabla          :: FilePath -> [Int] -> IO ()
```

tales que

- (aproximacionPi n) es la n-ésima aproximación de pi obtenida sumando los n primeros términos de la serie de Nilakantha. Por ejemplo,

```
aproximacionPi 0        ==  3.0
aproximacionPi 1        ==  3.1666666666666665
aproximacionPi 2        ==  3.1333333333333333
aproximacionPi 3        ==  3.145238095238095
aproximacionPi 4        ==  3.1396825396825396
aproximacionPi 5        ==  3.1427128427128426
aproximacionPi 10       ==  3.1414067184965018
aproximacionPi 100      ==  3.1415924109719824
aproximacionPi 1000     ==  3.141592653340544
aproximacionPi 10000    ==  3.141592653589538
aproximacionPi 100000   ==  3.1415926535897865
aproximacionPi 1000000  ==  3.141592653589787
pi                      ==  3.141592653589793
```

- (tabla f ns) escribe en el fichero f las n–ésimas aproximaciones de pi, donde n toma los valores de la lista ns, junto con sus errores. Por ejemplo, al evaluar la expresión

  ```
  tabla "AproximacionesPi.txt" [0,10..100]
  ```

  hace que el contenido del fichero .^AproximacionesPi.txt”sea

  ```
  +------+----------------+----------------+
  | n    | Aproximación   | Error          |
  +------+----------------+----------------+
  |    0 | 3.000000000000 | 0.141592653590 |
  |   10 | 3.141406718497 | 0.000185935093 |
  |   20 | 3.141565734659 | 0.000026918931 |
  |   30 | 3.141584272675 | 0.000008380915 |
  |   40 | 3.141589028941 | 0.000003624649 |
  |   50 | 3.141590769850 | 0.000001883740 |
  |   60 | 3.141591552546 | 0.000001101044 |
  |   70 | 3.141591955265 | 0.000000698325 |
  |   80 | 3.141592183260 | 0.000000470330 |
  |   90 | 3.141592321886 | 0.000000331704 |
  |  100 | 3.141592410972 | 0.000000242618 |
  +------+----------------+----------------+
  ```

  al evaluar la expresión

  ```
  tabla "AproximacionesPi.txt" [0,500..5000]
  ```

hace que el contenido del fichero .ªproximacionesPi.txt”sea

```
+------+----------------+----------------+
| n    | Aproximación   | Error          |
+------+----------------+----------------+
|    0 | 3.000000000000 | 0.141592653590 |
|  500 | 3.141592651602 | 0.000000001988 |
| 1000 | 3.141592653341 | 0.000000000249 |
| 1500 | 3.141592653516 | 0.000000000074 |
| 2000 | 3.141592653559 | 0.000000000031 |
| 2500 | 3.141592653574 | 0.000000000016 |
| 3000 | 3.141592653581 | 0.000000000009 |
| 3500 | 3.141592653584 | 0.000000000006 |
| 4000 | 3.141592653586 | 0.000000000004 |
| 4500 | 3.141592653587 | 0.000000000003 |
| 5000 | 3.141592653588 | 0.000000000002 |
+------+----------------+----------------+
```

Soluciones

```
import Text.Printf

-- 1ª solución
-- ===========

aproximacionPi :: Int -> Double
aproximacionPi n = serieNilakantha !! n

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

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

-- 2ª solución
-- ===========
```

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

-- 3ª solución
-- ===========

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


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

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

-- Definicioń de tabla
-- ===================

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

tablaAux :: [Int] -> String
tablaAux ns =
     linea
  ++ cabecera
  ++ linea
  ++ concat [printf "| %4d | %.12f | %.12f |\n" n a e
```

```
            | n <- ns
            , let a = aproximacionPi n
            , let e = abs (pi - a)]
  ++ linea

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

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

Ejercicio 70

Simplificación de expresiones booleanas

¿Dices que nada se pierde?
Si esta copa de cristal
se me rompe, nunca en ella
beberé, nunca jamás.

Antonio Machado

Enunciado

El siguiente tipo de dato algebraico representa las expresiones booleanas construidas con una variable (X), las constantes verdadera (V) y falsa (F), la negación (Neg) y la disyunción (Dis):

```
data Expr = X
          | V
          | F
          | Neg Expr
          | Dis Expr Expr
          deriving (Eq, Ord)
```

Por ejemplo, la fórmula $\neg X \vee V$ se representa por (Dis (Neg X) V).

Definir las funciones

```
valor      :: Expr -> Bool -> Bool
simplifica :: Expr -> Expr
```

tales que (valor p i) es el valor de la fórmula p cuando se le asigna a X el valor i. Por ejemplo,

```
valor (Neg X) True            ==  False
valor (Neg F) True            ==  True
valor (Dis X (Neg X)) True    ==  True
valor (Dis X (Neg X)) False   ==  True
```

y (simplifica p) es una expresión obtenida aplicándole a p las siguientes reglas de simplificación:

```
Neg V       = F
Neg F       = V
Neg (Neg q) = q
Dis V q     = V
Dis F q     = q
Dis q V     = V
Dis q F     = F
Dis q q     = q
```

Por ejemplo,

```
simplifica (Dis X (Neg (Neg X)))                     ==  X
simplifica (Neg (Dis (Neg (Neg X)) F))               ==  Neg X
simplifica (Dis (Neg F) F)                           ==  V
simplifica (Dis (Neg V) (Neg (Dis (Neg X) F)))       ==  X
simplifica (Dis (Neg V) (Neg (Dis (Neg (Neg X)) F))) ==  Neg X
```

Comprobar con QuickCheck que para cualquier fórmula p y cualquier booleano i se verifica que (valor (simplifica p) i) es igual a (valor p i).

Para la comprobación, de define el generador

```
instance Arbitrary Expr where
  arbitrary = sized expr
    where expr n | n <= 0     = atom
```

```
                     | otherwise = oneof [ atom
                                         , liftM Neg subexpr
                                         , liftM2 Dis subexpr subexpr ]
               where atom    = oneof [elements [X,V,F]]
                     subexpr = expr (n `div` 2)
```

que usa las funciones liftM y liftM2 de la librería Control.Monad que hay que importar al principio.

Soluciones

```
import Test.QuickCheck
import Control.Monad

data Expr = X
          | V
          | F
          | Neg Expr
          | Dis Expr Expr
          deriving (Eq, Ord, Show)

valor :: Expr -> Bool -> Bool
valor X i         = i
valor V _         = True
valor F _         = False
valor (Neg p) i   = not (valor p i)
valor (Dis p q) i = valor p i || valor q i

simplifica :: Expr -> Expr
simplifica X = X
simplifica V = V
simplifica F = F
simplifica (Neg p) = negacion (simplifica p)
  where negacion V        = F
        negacion F        = V
        negacion (Neg p') = p'
        negacion p'       = Neg p'
simplifica (Dis p q) = disyuncion (simplifica p) (simplifica q)
```

```
  where disyuncion V _  = V
        disyuncion F q' = q'
        disyuncion _ V  = V
        disyuncion q' F = q'
        disyuncion p' q' | p' == q'  = p'
                         | otherwise = Dis p' q'

-- La propiedad es
prop_simplifica :: Expr -> Bool -> Bool
prop_simplifica p i =
  valor (simplifica p) i == valor p i

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

-- Generador de fórmulas
instance Arbitrary Expr where
  arbitrary = sized expr
    where expr n | n <= 0    = atom
                 | otherwise = oneof [ atom
                                     , liftM Neg subexpr
                                     , liftM2 Dis subexpr subexpr ]
            where atom    = oneof [elements [X,V,F]]
                  subexpr = expr (n `div` 2)
```

Ejercicio 71

Sucesión de Cantor de números innombrables

Dices que nada se pierde
y acaso dices verdad;
pero todo lo perdemos
y todo nos perderá.

Antonio Machado

Enunciado

Un número es **innombrable** si es divisible por 7 o alguno de sus dígitos es un 7. Un juego infantil consiste en contar saltándose los números innombrables:

```
1 2 3 4 5 6 ( ) 8 9 10 11 12 13 ( ) 15 16 ( ) 18 ...
```

La sucesión de Cantor se obtiene llenando los huecos de la sucesión anterior:

```
1 2 3 4 5 6 (1) 8 9 10 11 12 13 (2) 15 16 (3) 18 19 20 (4) 22 23
24 25 26 (5) (6) 29 30 31 32 33 34 (1) 36 (8) 38 39 40 41  (9) 43
44 45 46 (10) 48 (11) 50 51 52 53 54 55 (12) (13) 58 59 60 61 62
(2) 64 65 66 (15) 68 69 (16) (3) (18) (19) (20) (4) (22) (23) (24)
```

```
(25) 80 81 82 83 (26) 85 86 (5) 88 89 90 (6) 92 93 94 95 96 (29)
(30) 99 100
```

Definir las funciones

```
sucCantor         :: [Integer]
graficaSucCantor :: Int -> IO ()
```

tales que

- sucCantor es la lista cuyos elementos son los términos de la sucesión de Cantor. Por ejemplo,

```
λ> take 100 sucCantor
[1,2,3,4,5,6, 1 ,8,9,10,11,12,13, 2, 15,16, 3, 18,19,20, 4,
 22,23,24,25,26, 5 , 6 ,29,30,31,32,33,34, 1 ,36 , 8 ,38,39,
 40,41, 9 ,43,44,45,46, 10 ,48, 11 ,50,51,52,53,54,55 , 12 ,
 13, 58,59,60,61,62, 2 ,64,65,66, 15 ,68,69, 16 , 3 , 18, 19,
 20, 4, 22, 23, 24 ,25 ,80,81,82,83, 26 ,85,86, 5 ,88,89,90,
 6, 92,93,94,95,96, 29, 30 ,99,100]
λ> sucCantor2 !! (5+10^6)
544480
λ> sucCantor2 !! (6+10^6)
266086
```

- (graficaSucCantor n) es la gráfica de los n primeros términos de la sucesión de Cantor. Por ejemplo, (graficaSucCantor 200) dibuja la Figura 71.1

Soluciones

```
import Graphics.Gnuplot.Simple

-- 1ª solución
-- ===========

sucCantor1 :: [Integer]
sucCantor1 = map fst $ scanl f (1,0) [2..]
  where f (_,i) x
```

```haskell
          | esInnombrable x = (sucCantor1 !! i, i+1)
          | otherwise       = (x,i)

esInnombrable :: Integer -> Bool
esInnombrable x =
  rem x 7 == 0 || '7' `elem` show x

-- 2ª solución
-- ===========

sucCantor2 :: [Integer]
sucCantor2 = aux 0 1
  where aux i x
          | esInnombrable x = sucCantor2 !! i : aux (i+1) (x+1)
          | otherwise       = x : aux i (x+1)

-- 3ª solución
-- ===========

sucCantor3 :: [Integer]
sucCantor3 = 1 : aux [2..] sucCantor3
  where aux [] _ = []
        aux (x:xs) a@(y:ys)
          | esInnombrable x = y : aux xs ys
          | otherwise       = x : aux xs a

-- Definición de graficaSucCantor
-- ========================================

graficaSucCantor :: Int -> IO ()
graficaSucCantor n =
  plotList [ Key Nothing
           , PNG ("Sucesion_de_Cantor_de_numeros_innombrables.png")
           ]
           (take n sucCantor3)
```

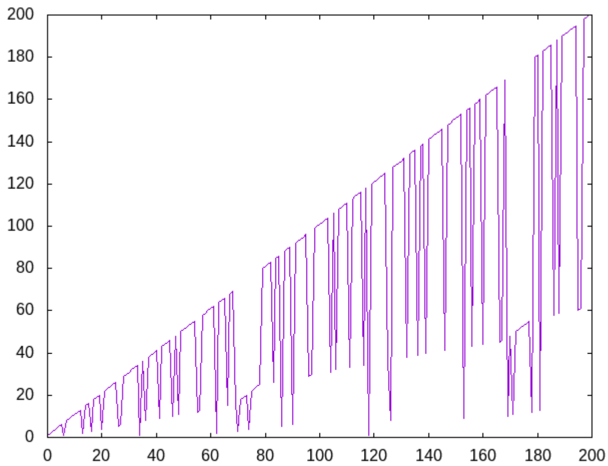


Figura 71.1: (graficaSucCantor 200)

Ejercicio 72

Ternas euclídeas

Todo pasa y todo queda,
pero lo nuestro es pasar,
pasar haciendo caminos,
caminos sobre la mar.

Antonio Machado

Enunciado

Uno de los problemas planteados por Euclides en los Elementos consiste en encontrar tres números tales que cada uno de sus productos, dos a dos, aumentados en la unidad sea un cuadrado perfecto.

Diremos que (x,y,z) es una terna euclídea si es una solución del problema; es decir, si $x \leq y \leq z$ y $xy+1, yz+1$ y $zx+1$ son cuadrados. Por ejemplo, (4,6,20) es una terna euclídea ya que

$$4 \times 6 + 1 = 5^2, 6 \times 20 + 1 = 11^2, 20 \times 4 + 1 = 9^2$$

Definir la funciones

```
ternasEuclideas         :: [(Integer,Integer,Integer)]
esMayorDeTernaEuclidea :: Integer -> Bool
```

tales que

- ternasEuclideas es la lista de las ternas euclídeas. Por ejemplo,

```
λ> take 7 ternasEuclideas
[(1,3,8),(2,4,12),(1,8,15),(3,5,16),(4,6,20),(3,8,21),(5,7,24)]
```

- (esMayorDeTernaEuclidea z) se verifica si existen x, y tales que (x,y,z) es una terna euclídea. Por ejemplo,

```
esMayorDeTernaEuclidea 20  ==  True
esMayorDeTernaEuclidea 22  ==  False
```

Comprobar con QuickCheck que z es el mayor de una terna euclídea si, y sólo si, existe un número natural x tal que $1 < x < z - 1$ y x^2 es congruente con 1 módulo z.

Soluciones

```
import Test.QuickCheck

ternasEuclideas :: [(Integer,Integer,Integer)]
ternasEuclideas =
  [(x,y,z) | z <- [1..]
           , y <- [1..z]
           , esCuadrado (y * z + 1)
           , x <- [1..y]
           , esCuadrado (x * y + 1)
           , esCuadrado (z * x + 1)]

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

esMayorDeTernaEuclidea :: Integer -> Bool
esMayorDeTernaEuclidea z =
  not (null [(x,y) | y <- [1..z]
```

```haskell
                    , esCuadrado (y * z + 1)
                    , x <- [1..y]
                    , esCuadrado (x * y + 1)
                    , esCuadrado (z * x + 1)])

-- La propiedad es
prop_esMayorDeTernaEuclidea :: Positive Integer -> Bool
prop_esMayorDeTernaEuclidea (Positive z) =
  esMayorDeTernaEuclidea z == any (\x -> (x^2) `mod` z == 1) [2..z-2]

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

Ejercicio 73

Mezcla de listas

Cuatro cosas tiene el hombre
que no sirven en la mar:
ancla, gobernalle y remos,
y miedo de naufragar.

Antonio Machado

Enunciado

Definir la función

```
mezcla :: [[a]] -> [a]
```

tal que (mezcla xss) es la lista tomando sucesivamente los elementos de xss en la misma posición. Cuando una de las listas de xss es vacía, se continua con las restantes. por ejemplo,

```
mezcla [[1,2],[3..7],[8..10]]             ==  [1,3,8,2,4,9,5,10,6,7]
mezcla ["Estamos","en","2019"]            ==  "Ee2sn0t1a9mos"
take 9 (mezcla [[3,6..],[5,7..],[0,1]])  ==  [3,5,0,6,7,1,9,9,12]
```

Soluciones

```haskell
import Data.List (transpose)

-- 1ª solución
mezcla :: [[a]] -> [a]
mezcla xss = aux (filter (not . null) xss)
  where
    aux []  = []
    aux yss = map head yss ++ mezcla (map tail yss)

-- 2ª solución
mezcla2 :: [[a]] -> [a]
mezcla2 = aux . filter (not . null)
  where
    aux []  = []
    aux yss = map head yss ++ mezcla (map tail yss)

-- 3ª solución
mezcla3 :: [[a]] -> [a]
mezcla3 = concatMap primeros . takeWhile (not . null) . iterate restos
  where primeros = map head . filter (not . null)
        restos   = map tail . filter (not . null)

-- 4ª solución
mezcla4 :: [[a]] -> [a]
mezcla4 = concat . transpose
```

Ejercicio 74

Mayor exponente

Mirando mi calavera
un nuevo Hamlet dirá:
He aquí un lindo fósil de una
careta de carnaval.

Antonio Machado

Enunciado

Definir las funciones

```
mayorExponente        :: Integer -> Integer
graficaMayorExponente :: Integer -> IO ()
```

tales que

- (mayorExponente n) es el mayor número b para el que existe un a tal que $n = a^b$. Se supone que $n > 1$. Por ejemplo,

```
mayorExponente 9    ==  2
mayorExponente 8    ==  3
mayorExponente 7    ==  1
mayorExponente 18   ==  1
mayorExponente 36   ==  2
```

```
mayorExponente4 144  ==  2
mayorExponente (10^(10^5))  ==  100000
```

- (graficaMayorExponente n) dibuja la gráfica de los mayores exponentes de los números entre 2 y n. Por ejemplo, (graficaMayorExponente 50) dibuja la Figura 74.1

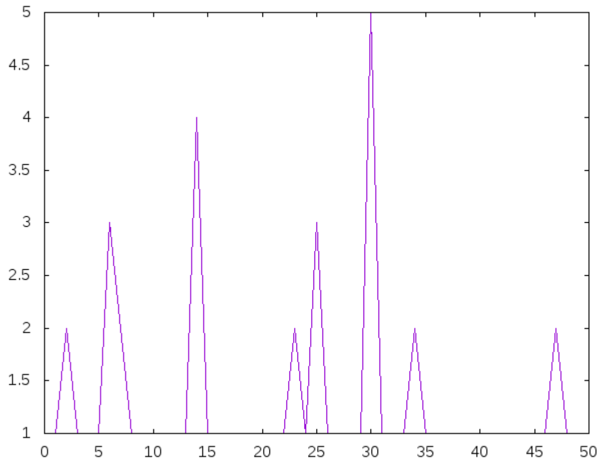


Figura 74.1: (graficaMayorExponente 50)

Soluciones

```
import Data.List (genericLength, group)
import Data.Numbers.Primes (primeFactors)
import Test.QuickCheck
import Graphics.Gnuplot.Simple

-- 1ª solución
-- ===========

mayorExponente :: Integer -> Integer
mayorExponente x =
  last [b | b <- [1..x]
          , a <- [1..x]
          , a^b == x]
```

```
-- 2ª solución
-- ===========

mayorExponente2 :: Integer -> Integer
mayorExponente2 x =
  head [b | b <- [x,x-1..1]
          , a <- [1..x]
          , a^b == x]

-- 3ª solución
-- ===========

mayorExponente3 :: Integer -> Integer
mayorExponente3 x = aux x
  where aux 1 = 1
        aux b | any (\a -> a^b == x) [1..x] = b
              | otherwise                   = aux (b-1)

-- 4ª solución
-- ===========

mayorExponente4 :: Integer -> Integer
mayorExponente4 x =
  mcd (exponentes x)

-- (exponentes x) es la lista de los exponentes en la factorizacioń de
-- x. por ejemplo.
--    exponentes 720  ==  [4,2,1]
exponentes :: Integer -> [Integer]
exponentes x =
  map genericLength (group (primeFactors x))

-- (mcd xs) es el máximo común divisor de xs. Por ejemplo,
--    mcd [4,6,10]  ==  2
mcd :: [Integer] -> Integer
mcd = foldr1 gcd

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

```haskell
-- La propiedad es
prop_mayorExponente :: Integer -> Property
prop_mayorExponente n =
  n >= 0 ==>
  mayorExponente  n == mayorExponente2 n &&
  mayorExponente2 n == mayorExponente3 n

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

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

--    λ> mayorExponente (10^3)
--    3
--    (3.96 secs, 4,671,928,464 bytes)
--    λ> mayorExponente2 (10^3)
--    3
--    (3.99 secs, 4,670,107,024 bytes)
--    λ> mayorExponente3 (10^3)
--    3
--    (3.90 secs, 4,686,383,952 bytes)
--    λ> mayorExponente4 (10^3)
--    3
--    (0.02 secs, 131,272 bytes)

-- Definición de graficaMayorExponente
-- ===================================

graficaMayorExponente :: Integer -> IO ()
graficaMayorExponente n =
  plotList [ Key Nothing
           , PNG ("MayorExponente.png")
           ]
           (map mayorExponente3 [2..n])
```

Ejercicio 75

Número de sumandos en suma de cuadrados

– Nuestro español bosteza. ¿Es hambre? ¿Sueño? ¿Hastío?
Doctor, ¿tendrá el estómago vacío?
– El vacío es más bien en la cabeza.

Antonio Machado

Enunciado

El teorema de Lagrange de los cuatro cuadrados asegura que cualquier número entero positivo es la suma de, como máximo,cuatro cuadrados de números enteros. Por ejemplo,

```
16 = 4²
29 = 2² + 5²
14 = 1² + 2² + 3²
15 = 1² + 1² + 2² + 3²
```

Definir las funciones

```haskell
ordenLagrange        :: Integer -> Int
graficaOrdenLagrange :: Integer -> IO ()
```

tales que

- (ordenLagrange n) es el menor número de cuadrados necesarios para escribir n como suma de cuadrados. Por ejemplo.

```
ordenLagrange 16     ==  1
ordenLagrange 29     ==  2
ordenLagrange 14     ==  3
ordenLagrange 15     ==  4
ordenLagrange 10000  ==  1
ordenLagrange 10001  ==  2
ordenLagrange 10002  ==  3
ordenLagrange 10007  ==  4
```

- (graficaOrdenLagrange n) dibuja la gráfica de los órdenes de Lagrange de los n primeros números naturales. Por ejemplo, (graficaOrdenLagrange 100) dibuja la Figura 75.1

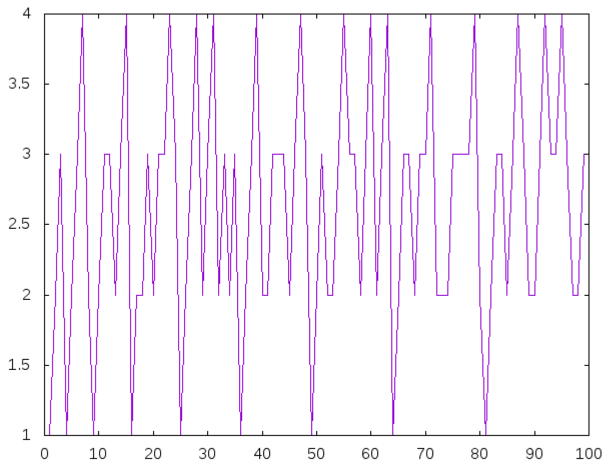


Figura 75.1: (graficaOrdenLagrange 100)

Comprobar con QuickCheck que. para todo entero positivo k, el orden de Lagrange de k es menos o igual que 4, el de 4k+3 es distinto de 2 y el de 8k+7 es distinto de 3.

Soluciones

```haskell
import Data.Array (Array, (!), array)
import Graphics.Gnuplot.Simple

import Test.QuickCheck

-- 1ª definición
-- =============

ordenLagrange :: Integer -> Int
ordenLagrange n
  | esCuadrado n = 1
  | otherwise    = 1 + minimum [ ordenLagrange (n - x^2)
                               | x <- [1..raizEntera n]]

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

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

-- 2ª definición
-- =============

ordenLagrange2 :: Integer -> Int
ordenLagrange2 n = (vectorOrdenLagrange n) ! n

vectorOrdenLagrange :: Integer -> Array Integer Int
vectorOrdenLagrange n = v where
  v = array (0,n) [(i,f i) | i <- [0..n]]
```

```
  f i | esCuadrado i = 1
      | otherwise    = 1 + minimum [ v ! (i - j^2)
                                   | j <- [1..raizEntera i]]

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

-- La comparación es
--    λ> ordenLagrange 50
--    2
--    (10.39 secs, 1,704,144,464 bytes)
--    λ> ordenLagrange2 50
--    2
--    (0.01 secs, 341,920 bytes)

-- Definición de graficaOrdenLagrange
-- ==================================

graficaOrdenLagrange :: Integer -> IO ()
graficaOrdenLagrange n =
  plotList [ Key Nothing
           , PNG ("Numero_de_sumandos_en_suma_de_cuadrados.png")
           ]
           (map ordenLagrange2 [0..n-1])

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_OrdenLagrange :: Positive Integer -> Bool
prop_OrdenLagrange (Positive k) =
  ordenLagrange2 k <= 4 &&
  ordenLagrange2 (4*k+3) /= 2 &&
  ordenLagrange2 (8*k+7) /= 3

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

Ejercicio 76

Divisiones del círculo

...Y si la vida es corta
y no llega la mar a tu galera,
aguarda sin partir y siempre espera,
que el arte es largo y, además no importa.

Antonio Machado

Enunciado

Dado 4 puntos de un círculo se pueden dibujar 2 cuerdas entre ellos de forma que no se corten. En efecto, si se enumeran los puntos del 1 al 4 en sentido de las agujas del reloj una forma tiene las cuerdas {1–2, 3–4} y la otra {1–4, 2–3}.

Definir la función

```
numeroFormas :: Integer -> Integer
```

tal que (numeroFormas n) es el número de formas que se pueden dibujar n cuerdas entre 2xn puntos de un círculo sin que se corten. Por ejemplo,

```
numeroFormas 1   ==  1
numeroFormas 2   ==  2
numeroFormas 4   ==  14
numeroFormas 75  ==  1221395654430378811828760722007962130791020
length (show (numeroFormas 1000))  ==  598
```

Soluciones

```haskell
import Data.Array

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

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

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

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

Ejercicio 77

Inserciones en una lista de listas

...De la mar al percepto,
del percepto al concepto,
del concepto a la idea
– ¡oh, la linda tarea! –
de la idea a la mar.
¡Y otra vez al empezar!

Antonio Machado

Enunciado

Definir la función

```
inserta :: a -> [[a]] -> [[[a]]]
```

tal que (inserta x yss) es la lista obtenida insertando x en cada uno de los elementos de yss. Por ejemplo,

```
λ> inserta 1 [[2,3],[4],[5,6,7]]
[[[1,2,3],[4],[5,6,7]],[[2,3],[1,4],[5,6,7]],[[2,3],[4],[1,5,6,7]]]
λ> inserta 'a' ["hoy","es","lunes"]
[["ahoy","es","lunes"],["hoy","aes","lunes"],["hoy","es","alunes"]]
```

Soluciones

```haskell
-- 1ª solución
inserta :: a -> [[a]] -> [[[a]]]
inserta _ []       = []
inserta x (ys:yss) = ((x:ys):yss) : [ys : zs | zs <- inserta x yss]

-- 2ª solución
inserta2 :: a -> [[a]] -> [[[a]]]
inserta2 _ []       = []
inserta2 x (ys:yss) = ((x:ys):yss) : map (ys:) (inserta2 x yss)
```

Ejercicio 78

Particiones de un conjunto

A quien nos justifica nuestra desconfianza
llamamos enemigo, ladrón de una esperanza.
Jamás perdona el necio si ve la nuez vacía
que dio a cascar al diente de la sabiduría.

Antonio Machado

Enunciado

Una partición de un conjunto A es un conjunto de subconjuntos no vacíos de A, disjuntos dos a dos y cuya unión es A. Por ejemplo, el conjunto {1, 2, 3} tiene exactamente 5 particiones:

```
{{1}, {2}, {3}}
{{1,2}, {3}}
{{1,3}, {2}}
{{1}, {2,3}}
{{1,2,3}}
```

Definir la función

```
particiones :: [a] -> [[[a]]]
```

tal que (particiones xs) es el conjunto de las particiones de xs. Por ejemplo,

```
λ> particiones [1,2]
[[[1,2]],[[1],[2]]]
λ> particiones [1,2,3]
[[[1,2,3]],[[1],[2,3]],[[1,2],[3]],[[2],[1,3]],[[1],[2],[3]]]
λ> particiones "abcd"
[["abcd"],["a","bcd"],["ab","cd"],["b","acd"],["abc","d"],["bc","ad"],
 ["ac","bd"],["c","abd"],["a","b","cd"],["a","bc","d"],["a","c","bd"],
 ["ab","c","d"],["b","ac","d"],["b","c","ad"],["a","b","c","d"]]
```

Soluciones

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

-- 1ª solución
-- ===========

particiones :: [a] -> [[[a]]]
particiones [] = [[]]
particiones (x:xs) =
  concat [([x] : yss) : inserta x yss | yss <- ysss]
  where ysss = particiones xs

-- (inserta x yss) es la lista obtenida insertando x en cada uno de los
-- elementos de yss. Por ejemplo,
--    λ> inserta 1 [[2,3],[4],[5,6,7]]
--    [[[1,2,3],[4],[5,6,7]],[[2,3],[1,4],[5,6,7]],[[2,3],[4],[1,5,6,7]]]
inserta :: a -> [[a]] -> [[[a]]]
inserta _ []       = []
inserta x (ys:yss) = ((x:ys):yss) : [ys : zs | zs <- inserta x yss]

-- 2ª solución
-- ===========

particiones2 :: [a] -> [[[a]]]
particiones2 [] = [[]]
```

```haskell
particiones2 xs =
  concat [particionesFijas xs k | k <- [0..length xs]]

-- (particionesFijas xs k) es el conjunto de las particiones de xs en k
-- subconjuntos. Por ejemplo,
--    particionesFijas [1,2,3] 0  ==  []
--    particionesFijas [1,2,3] 1  ==  [[[1,2,3]]]
--    particionesFijas [1,2,3] 2  ==  [[[1],[2,3]],[[1,2],[3]],[[2],[1,3]]]
--    particionesFijas [1,2,3] 3  ==  [[[1],[2],[3]]]
--    particionesFijas [1,2,3] 4  ==  []
particionesFijas :: [a] -> Int -> [[[a]]]
particionesFijas [] _ = []
particionesFijas xs 1 = [[xs]]
particionesFijas (x:xs) k =
   [[x]:ys | ys <- particionesFijas xs (k-1)] ++
   concat [inserta x ys | ys <- particionesFijas xs k]

-- 3ª solución
-- ===========

particiones3 :: [a] -> [[[a]]]
particiones3 xs = concat [a ! (n,k) | k <- [0..n]]
  where a = matrizParticiones xs
        n = length xs

-- (matrizParticiones xs) es la matriz de dimensión ((0,0),(n,n)) que en
-- la posición (i,j) tiene el conjunto de particiones de los i primeros
-- elementoa de xs en j subconjuntos. Por ejemplo,
--   λ> elems (matrizParticiones [1,2,3])
--   [[[]],[],         [],                                    [],
--    [],  [[[1]]],    [],                                    [],
--    [],  [[[1,2]]],  [[[2],[1]]],                           [],
--    [],  [[[1,2,3]]],[[[3],[1,2]],[[3,2],[1]],[[2],[3,1]]],[[[3],[2],[1]]]]
matrizParticiones :: [a] -> Array (Int,Int) [[[a]]]
matrizParticiones xs = a
  where
    n = length xs
    v = listArray (1,n) xs
    a = array ((0,0),(n,n)) [((i,j), f i j) | i <- [0..n], j <- [0..n]]
    f 0 0 = [[]]
```

```
    f 0 _ = []
    f _ 0 = []
    f i 1 = [[[v!k | k <- [1..i]]]]
    f i j = [[v!i] : ys | ys <- a!(i-1,j-1)] ++
            concat [inserta (v!i) ys | ys <- a!(i-1,j)]

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

-- La propiedad es
prop_Particiones :: [Int] -> Bool
prop_Particiones xs =
  all (== (ordenada . particiones) xs)
      [(ordenada . f )xs | f <- [ particiones2
                                , particiones3]]

ordenada :: Ord a => [[[a]]] -> [[[a]]]
ordenada xsss =
  sort [sort (map sort xss) | xss <- xsss]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=10}) prop_Particiones
--    +++ OK, passed 100 tests.

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

--    λ> length (particiones [1..12])
--    4213597
--    (2.74 secs, 2,903,492,120 bytes)
--    λ> length (particiones2 [1..12])
--    4213597
--    (4.63 secs, 3,878,003,920 bytes)
--    λ> length (particiones3 [1..12])
--    4213597
--    (6.21 secs, 3,199,076,464 bytes)
```

Ejercicio 79

Número de particiones de un conjunto

Yo he visto garras fieras en las pulidas manos;
conozco grajos mélicos y líricos marranos ...
El más truhán se lleva la mano al corazón,
y el bruto más espeso se carga de razón.

Antonio Machado

Enunciado

Una partición de un conjunto A es un conjunto de subconjuntos no vacíos de A, disjuntos dos a dos y cuya unión es A. Por ejemplo, el conjunto {1, 2, 3} tiene exactamente 5 particiones:

```
{{1}, {2}, {3}}
{{1,2}, {3}}
{{1,3}, {2}}
{{1}, {2,3}}
{{1,2,3}}
```

Definir la función

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

tal que (nParticiones xs) es el número de particiones de xs. Por ejemplo,

```
nParticiones [1,2]                      ==  2
nParticiones [1,2,3]                    ==  5
nParticiones "abcd"                     ==  15
length (show (nParticiones [1..500]))  ==  844
```

Soluciones

```haskell
import Data.List  ( genericLength
                  )
import Data.Array ( Array
                  , (!)
                  , array
                  )

-- 1ª definición
-- =============

nParticiones :: [a] -> Integer
nParticiones xs =
  genericLength (particiones xs)

-- (particiones xs) es el conjunto de las particiones de xs. Por
-- ejemplo,
--    λ> particiones [1,2]
--    [[[1,2]],[[1],[2]]]
--    λ> particiones [1,2,3]
--    [[[1,2,3]],[[1],[2,3]],[[1,2],[3]],[[2],[1,3]],[[1],[2],[3]]]
particiones :: [a] -> [[[a]]]
particiones [] = [[]]
particiones xs =
  concat [particionesFijas xs k | k <- [0..length xs]]

-- (particionesFijas xs k) es el conjunto de las particiones de xs en k
-- subconjuntos. Por ejemplo,
--    particionesFijas [1,2,3] 0  ==  []
--    particionesFijas [1,2,3] 1  ==  [[[1,2,3]]]
--    particionesFijas [1,2,3] 2  ==  [[[1],[2,3]],[[1,2],[3]],[[2],[1,3]]]
```

```
--    particionesFijas [1,2,3] 3  ==  [[[1],[2],[3]]]
--    particionesFijas [1,2,3] 4  ==  []
particionesFijas :: [a] -> Int -> [[[a]]]
particionesFijas [] _ = []
particionesFijas xs 1 = [[xs]]
particionesFijas (x:xs) k =
   [[x]:ys | ys <- particionesFijas xs (k-1)] ++
   concat [inserta x ys | ys <- particionesFijas xs k]

-- (inserta x yss) es la lista obtenida insertando x en cada uno de los
-- elementos de yss. Por ejemplo,
--    λ> inserta 1 [[2,3],[4],[5,6,7]]
--    [[[1,2,3],[4],[5,6,7]],[[2,3],[1,4],[5,6,7]],[[2,3],[4],[1,5,6,7]]]
inserta :: a -> [[a]] -> [[[a]]]
inserta _ []       = []
inserta x (ys:yss) = ((x:ys):yss) : [ys : zs | zs <- inserta x yss]

-- 2ª definición
-- =============

nParticiones2 :: [a] -> Integer
nParticiones2 xs = sum [nParticionesFijas n k | k <- [0..n]]
  where n = genericLength xs

-- nPparticionesFijas n k) es el número de las particiones de un
-- conjunto con n elementos en k subconjuntos. Por ejemplo,
--    nParticionesFijas 3 0  ==  0
--    nParticionesFijas 3 1  ==  1
--    nParticionesFijas 3 2  ==  3
--    nParticionesFijas 3 3  ==  1
--    nParticionesFijas 3 4  ==  0
nParticionesFijas :: Integer -> Integer -> Integer
nParticionesFijas 0 0 = 1
nParticionesFijas 0 _ = 0
nParticionesFijas _ 1 = 1
nParticionesFijas n k =
  nParticionesFijas (n-1) (k-1) + k * nParticionesFijas (n-1) k

-- 3ª definición
-- =============
```

```haskell
nParticiones3 :: [a] -> Integer
nParticiones3 xs = sum [a ! (n,k) | k <- [0..n]]
  where n = genericLength xs
        a = matrizNParticiones n

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

-- 4ª definición
-- =============

nParticiones4 :: [a] -> Integer
nParticiones4 xs = sum [a ! (n,k) | k <- [0..n]]
  where
    n = genericLength xs
    a = array ((0,0),(n,n)) [((i,j), f i j) | i <- [0..n], j <- [0..n]]
```

```
    f 0 0 = 1
    f 0 _ = 0
    f _ 0 = 0
    f _ 1 = 1
    f i j = a ! (i-1,j-1) + j * a ! (i-1,j)

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

--    λ> nParticiones [1..11]
--    678570
--    (3.77 secs, 705,537,480 bytes)
--    λ> nParticiones2 [1..11]
--    678570
--    (0.07 secs, 6,656,584 bytes)
--    λ> nParticiones3 [1..11]
--    678570
--    (0.01 secs, 262,176 bytes)
--    λ> nParticiones4 [1..11]
--    678570
--    (0.01 secs, 262,264 bytes)
--
--    λ> nParticiones2 [1..16]
--    10480142147
--    (2.24 secs, 289,774,408 bytes)
--    λ> nParticiones3 [1..16]
--    10480142147
--    (0.01 secs, 437,688 bytes)
--    λ> nParticiones4 [1..16]
--    10480142147
--    (0.01 secs, 437,688 bytes)
--
--    λ> length (show (nParticiones3 [1..500]))
--    844
--    (2.23 secs, 357,169,528 bytes)
--    λ> length (show (nParticiones4 [1..500]))
--    844
--    (2.20 secs, 357,172,680 bytes)
```

Ejercicio 80

Descomposiciones en sumas de cuadrados

No extrañéis, dulces amigos,
que esté mi frente arrugada;
yo vivo en paz con los hombres
y en guerra con mis entrañas.

Antonio Machado

Enunciado

Definir la función

```haskell
descomposiciones :: Int -> [[Int]]
```

tal que (descomposiciones x) es la lista de las listas de los cuadrados de cuatro números enteros positivos cuya suma es x. Por ejemplo.

```haskell
λ> descomposiciones 4
[[1,1,1,1]]
λ> descomposiciones 5
[]
λ> descomposiciones 7
[[1,1,1,4],[1,1,4,1],[1,4,1,1],[4,1,1,1]]
```

```
λ> descomposiciones 10
[[1,1,4,4],[1,4,1,4],[1,4,4,1],[4,1,1,4],[4,1,4,1],[4,4,1,1]]
λ> descomposiciones 15
[[1,1,4,9],[1,1,9,4],[1,4,1,9],[1,4,9,1],[1,9,1,4],[1,9,4,1],
 [4,1,1,9],[4,1,9,1],[4,9,1,1],[9,1,1,4],[9,1,4,1],[9,4,1,1]]
λ> length (descomposiciones 50000)
5682
```

Soluciones

```
import Data.Array
import Test.QuickCheck

-- 1ª definición
-- =============

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

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

-- (raizEntera n) es el mayor entero cuya raíz cuadrada es menor o igual
-- que n. Por ejemplo,
--    raizEntera 15  ==  3
```

```haskell
--    raizEntera 16  ==  4
--    raizEntera 17  ==  4
raizEntera :: Int -> Int
raizEntera = floor . sqrt . fromIntegral

-- 2ª definición
-- =============

descomposiciones2 :: Int -> [[Int]]
descomposiciones2 x = a ! (x,4)
  where
    a = array ((0,1),(x,4)) [((i,j), f i j) | i <- [0..x], j <- [1..4]]
    f 0 1 = []
    f 1 1 = [[1]]
    f 2 1 = []
    f 3 1 = []
    f i 1 | esCuadrado i = [[i]]
          | otherwise    = []
    f i j = [z^2 : zs | z <- [1..raizEntera i]
                      , zs <- a ! (i - z^2,j-1)]

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

-- La propiedad es
prop_descomposiciones :: Positive Int -> Bool
prop_descomposiciones (Positive x) =
  descomposiciones x == descomposiciones2 x

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

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

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

```
--    λ> length (descomposiciones2 (2*10^4))
--    1068
--    (0.72 secs, 678,416,144 bytes)
```

Ejercicio 81

Número de descomposiciones en sumas de cuadrados

Ya habrá cigüeñas al sol,
mirando la tarde roja,
entre Moncayo y Urbión.

Antonio Machado

Enunciado

Definir las funciones

```
nDescomposiciones       :: Int -> Int
graficaDescomposiciones :: Int -> IO ()
```

tales que

- (nDescomposiciones x) es el número de listas de los cuadrados de cuatro números enteros positivos cuya suma es x. Por ejemplo.

```
nDescomposiciones 4      ==  1
nDescomposiciones 5      ==  0
nDescomposiciones 7      ==  4
nDescomposiciones 10     ==  6
```

```
nDescomposiciones 15     ==  12
nDescomposiciones 50000  ==  5682
```

- (graficaDescomposiciones n) dibuja la gráfica del número de descomposiciones de los n primeros números naturales. Por ejemplo. (graficaDescomposiciones 500) dibuja la Figura 81.1

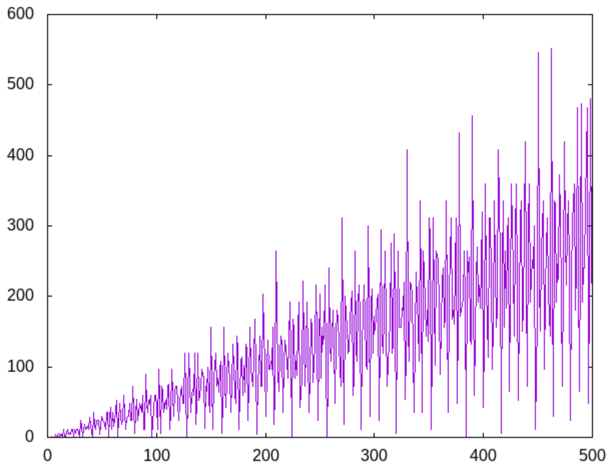


Figura 81.1: (graficaDescomposiciones 500)

Soluciones

```haskell
import Data.Array
import Graphics.Gnuplot.Simple
import Test.QuickCheck

-- 1ª solución
-- ===========

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

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

```haskell
--    λ> descomposiciones 5
--    []
--    λ> descomposiciones 7
--    [[1,1,1,4],[1,1,4,1],[1,4,1,1],[4,1,1,1]]
--    λ> descomposiciones 10
--    [[1,1,4,4],[1,4,1,4],[1,4,4,1],[4,1,1,4],[4,1,4,1],[4,4,1,1]]
--    λ> descomposiciones 15
--    [[1,1,4,9],[1,1,9,4],[1,4,1,9],[1,4,9,1],[1,9,1,4],[1,9,4,1],
--     [4,1,1,9],[4,1,9,1],[4,9,1,1],[9,1,1,4],[9,1,4,1],[9,4,1,1]]
descomposiciones :: Int -> [[Int]]
descomposiciones x = aux x 4
  where
    aux 0 1 = []
    aux 1 1 = [[1]]
    aux 2 1 = []
    aux 3 1 = []
    aux y 1 | esCuadrado y = [[y]]
            | otherwise    = []
    aux y n = [z^2 : zs | z <- [1..raizEntera y]
                        , zs <- aux (y - z^2) (n-1)]

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

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

-- 2ª solución
-- =============

nDescomposiciones2 :: Int -> Int
```

```haskell
nDescomposiciones2 = length . descomposiciones2

descomposiciones2 :: Int -> [[Int]]
descomposiciones2 x = a ! (x,4)
  where
    a = array ((0,1),(x,4)) [((i,j), f i j) | i <- [0..x], j <- [1..4]]
    f 0 1 = []
    f 1 1 = [[1]]
    f 2 1 = []
    f 3 1 = []
    f i 1 | esCuadrado i = [[i]]
          | otherwise    = []
    f i j = [z^2 : zs | z <- [1..raizEntera i]
                      , zs <- a ! (i - z^2,j-1)]

-- 3ª solución
-- ===========

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

-- 4ª solución
-- ===========

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

```
    f i 1 | esCuadrado i = 1
          | otherwise    = 0
    f i j = sum [a ! (i- z^2,j-1) | z <- [1..raizEntera i]]

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

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

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

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

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

```
-- Definición de graficaDescomposiciones
-- ======================================

graficaDescomposiciones :: Int -> IO ()
graficaDescomposiciones n =
  plotList [ Key Nothing
           , PNG ("Numero_de_descomposiciones_en_sumas_de_cuadrados.png")
           ]
           (map nDescomposiciones3 [0..n])
```

Ejercicio 82

Hojas con caminos no decrecientes

Para dialogar,
preguntad, primero;
después …escuchad.

Antonio Machado

Enunciado

Los árboles se pueden representar mediante el siguiente tipo de datos

```haskell
data Arbol = N Int [Arbol]
  deriving Show
```

Por ejemplo, los árboles

```
         1               1              1
        / \             / \            / \
       /   \           8   3          8   3
      2     6             /|\        /|\  |
     / \   / \           4 2 6      4 5 6 2
    4   5 5   7
```

se representan por

```
ej1, ej2, ej3 :: Arbol
ej1 = N 1 [N 2 [N 4 [], N 5 []], N 6 [N 5 [], N 7 []]]
ej2 = N 1 [N 8 [], N 3 [N 4 [], N 2 [], N 6 []]]
ej3 = N 1 [N 8 [N 4 [], N 5 [], N 6 []], N 3 [N 2 []]]
```

Definir la función

```
hojasEnNoDecreciente :: Arbol -> [Int]
```

tal que (hojasEnNoDecreciente a) es el conjunto de las hojas de a que se encuentran en alguna rama no decreciente. Por ejemplo,

```
hojasEnNoDecreciente ej1  ==  [4,5,7]
hojasEnNoDecreciente ej2  ==  [4,6,8]
hojasEnNoDecreciente ej3  ==  []
```

Soluciones

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

data Arbol = N Int [Arbol]
  deriving Show

ej1, ej2, ej3 :: Arbol
ej1 = N 1 [N 2 [N 4 [], N 5 []], N 6 [N 5 [], N 7 []]]
ej2 = N 1 [N 8 [], N 3 [N 4 [], N 2 [], N 6 []]]
ej3 = N 1 [N 8 [N 4 [], N 5 [], N 6 []], N 3 [N 2 []]]

-- 1ª solución
-- ===========

hojasEnNoDecreciente :: Arbol -> [Int]
hojasEnNoDecreciente a =
  sort (nub (map last (ramasNoDecrecientes a)))

--    ramasNoDecrecientes ej1  ==  [[1,2,4],[1,2,5],[1,6,7]]
--    ramasNoDecrecientes ej2  ==  [[1,8],[1,3,4],[1,3,6]]
```

```haskell
--    ramasNoDecrecientes ej3  ==  []
ramasNoDecrecientes :: Arbol -> [[Int]]
ramasNoDecrecientes a =
  filter esNoDecreciente (ramas a)

-- (ramas a) es la lista de las ramas del árbol a. Por ejemplo,
--    λ> ramas ej1
--    [[1,2,4],[1,2,5],[1,6,5],[1,6,7]]
--    λ> ramas ej2
--    [[1,8],[1,3,4],[1,3,2],[1,3,6]]
--    λ> ramas ej3
--    [[1,8,4],[1,8,5],[1,8,6],[1,3,2]]
ramas :: Arbol -> [[Int]]
ramas (N x []) = [[x]]
ramas (N x as) = map (x:) (concatMap ramas as)

-- (esNoDecreciente xs) se verifica si la lista xs es no
-- decreciente. Por ejemplo,
--    esNoDecreciente [1,3,3,5]  ==  True
--    esNoDecreciente [1,3,5,3]  ==  False
esNoDecreciente :: [Int] -> Bool
esNoDecreciente xs =
  and (zipWith (<=) xs (tail xs))

-- 2ª solución
-- ===========

--    hojasEnNoDecreciente ej1  ==  [4,5,7]
--    hojasEnNoDecreciente ej2  ==  [4,6,8]
--    hojasEnNoDecreciente ej3  ==  []
hojasEnNoDecreciente2 :: Arbol -> [Int]
hojasEnNoDecreciente2 = sort . nub . aux
  where
    aux (N x []) = [x]
    aux (N x as) = concat [aux (N y bs) | (N y bs) <- as, x <= y]
```

Ejercicio 83

Las torres de Hanói

En preguntar lo que sabes
el tiempo no has de perder …
Y a preguntas sin respuesta
¿quién te podrá responder?

Antonio Machado

Enunciado

Las torres de Hanoi es un rompecabeza que consta de tres postes que llamaremos A, B y C. Hay N discos de distintos tamaños en el poste A, de forma que no hay un disco situado sobre otro de menor tamaño. Los postes B y C están vacíos. Sólo puede moverse un disco a la vez y todos los discos deben de estar ensartados en algún poste. Ningún disco puede situarse sobre otro de menor tamaño. El problema consiste en colocar los N discos en el poste C.

Los postes se pueden representar mediante el siguiente tipo de datos

```
data Poste = A | B | C
  deriving Show
```

Definir las funciones

```
movimientos :: Integer -> [(Integer,Poste,Poste)]
hanoi       :: Integer -> IO ()
```

tales que

- (movimientos n) es la lista de los movimientos para resolver el problema de las torres de hanoi con n discos. Por ejemplo,

```
λ> movimientos 1
[(1,A,C)]
λ> movimientos 2
[(1,A,B),(2,A,C),(1,B,C)]
λ> movimientos 3
[(1,A,C),(2,A,B),(1,C,B),(3,A,C),(1,B,A),(2,B,C),(1,A,C)]
```

- (hanoi n) escribe los mensajes de los movimientos para resolver el problema de las torres de hanoi con n discos. Por ejemplo,

```
λ> hanoi 3
Mueve el disco 1 de A a C
Mueve el disco 2 de A a B
Mueve el disco 1 de C a B
Mueve el disco 3 de A a C
Mueve el disco 1 de B a A
Mueve el disco 2 de B a C
Mueve el disco 1 de A a C
```

Soluciones

```haskell
data Poste = A | B | C
  deriving (Eq, Show)

movimientos :: Integer -> [(Integer,Poste,Poste)]
movimientos n = aux n A B C
  where
    aux k a b c
      | k == 1    = [(1,a,c)]
      | otherwise = aux (k-1) a c b ++ (k,a,c) : aux (k-1) b a c

hanoi :: Integer -> IO ()
hanoi n =
  putStrLn (unlines (map mensaje (movimientos n)))
```

```haskell
-- (mensaje (n.x.y)) es la cadena indicando que el disco n se ha movido
-- desde el poste x al poste y. Por ejemplo,
--    λ> mensaje (1,A,B)
--    "Mueve el disco 1 de A a B"
mensaje :: (Integer,Poste,Poste) -> String
mensaje (n,x,y) =
  "Mueve el disco " ++ show n ++ " de " ++ show x ++ " a " ++ show y
```

Ejercicio 84

Número como suma de sus dígitos

Caminante, son tus huellas
el camino, y nada más;
caminante no hay camino,
se hace camino al andar.

Antonio Machado

Enunciado

El número 23 se puede escribir de 4 formas como suma de sus dígitos

```
2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 3
2 + 2 + 2 + 2 + 2 + 2 + 2 + 3 + 3 + 3
2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 3
2 + 3 + 3 + 3 + 3 + 3 + 3 + 3
```

La de menor número de sumando es la última, que tiene 8 sumandos.

Definir las funciones

```
minimoSumandosDigitos        :: Integer -> Integer
graficaMinimoSumandosDigitos :: Integer -> IO ()
```

tales que

- (minimoSumandosDigitos n) es el menor número de dígitos de n cuya suma es n. Por ejemplo,

```
minimoSumandosDigitos 23    ==  8
minimoSumandosDigitos 232   ==  78
minimoSumandosDigitos 2323  ==  775
map minimoSumandosDigitos [10..20] == [10,11,6,5,5,3,6,5,4,3,10]
```

- (graficaMinimoSumandosDigitos n) dibuja la gráfica de (minimoSumandosDigitos k) par los k primeros números naturales. Por ejemplo, (graficaMinimoSumandosDigitos 300) dibuja

  la Figura 84.1

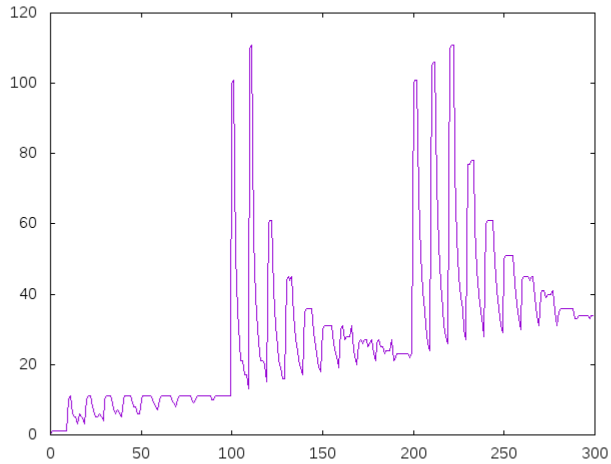


Figura 84.1: (graficaMinimoSumandosDigitos 300)

Soluciones

```
import Test.QuickCheck
import Graphics.Gnuplot.Simple
import Data.List (nub, genericLength, sort)

minimoSumandosDigitos :: Integer -> Integer
minimoSumandosDigitos n =
  minimoSumandos (digitos n) n
```

```haskell
-- (digitos n) es el conjunto de los dígitos no nulos de n. Por ejemplo,
--    digitos 2032  ==  [2,3]
digitos :: Integer -> [Integer]
digitos n =
  nub [read [c] | c <- show n, c /= '0']

-- (minimoSumandos xs n) es el menor número de elementos de la lista de
-- enteros positivos xs (con posibles repeticiones) cuya suma es n. Por
-- ejemplo,
--    minimoSumandos [7,2,4] 11  ==  2
minimoSumandos :: [Integer] -> Integer -> Integer
minimoSumandos xs n =
  minimum (map genericLength (sumas xs n))

-- (sumas xs n) es la lista de elementos de la lista de enteros
-- positivos xs (con posibles repeticiones) cuya suma es n. Por ejemplo,
--    sumas [7,2,4] 11  ==  [[7,2,2],[7,4]]
sumas :: [Integer] -> Integer -> [[Integer]]
sumas [] 0 = [[]]
sumas [] _ = []
sumas (x:xs) n
  | x <= n    = map (x:) (sumas (x:xs) (n-x)) ++ sumas xs n
  | otherwise = sumas xs n

-- 2ª solución
-- ===========

minimoSumandosDigitos2 :: Integer -> Integer
minimoSumandosDigitos2 n =
  minimoSumandos (digitos2 n) n

-- (digitos n) es el conjunto de los dígitos no nulos de n. Por ejemplo,
--    digitos 2032  ==  [2,3]
digitos2 :: Integer -> [Integer]
digitos2 n =
  reverse (sort (nub [read [c] | c <- show n, c /= '0']))

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

```haskell
-- La propiedad es
prop_minimoSumandosDigitos :: Positive Integer -> Bool
prop_minimoSumandosDigitos (Positive n) =
  minimoSumandosDigitos n == minimoSumandosDigitos2 n

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

-- Definición de graficaMinimoSumandosDigitos
-- ==========================================

graficaMinimoSumandosDigitos :: Integer -> IO ()
graficaMinimoSumandosDigitos n =
  plotList [ Key Nothing
           -- , PNG "Numero_como_suma_de_sus_digitos.png"
           ]
           [minimoSumandosDigitos k | k <- [0..n-1]]
```

Ejercicio 85

Suma de segmentos iniciales

Al andar se hace camino,
y al volver la vista atrás
se ve la senda que nunca
se ha de volver a pisar.

Antonio Machado

Enunciado

Los segmentos iniciales de [3,1,2,5] son [3], [3,1], [3,1,2] y [3,1,2,5]. Sus sumas son 3, 4, 6 y 9, respectivamente. La suma de dichas sumas es 24.

Definir la función

```
sumaSegmentosIniciales :: [Integer] -> Integer
```

tal que (sumaSegmentosIniciales xs) es la suma de las sumas de los segmentos iniciales de xs. Por ejemplo,

```
sumaSegmentosIniciales [3,1,2,5]     ==  24
sumaSegmentosIniciales3 [1..3*10^6]  ==  4500004500001000000
```

Comprobar con QuickCheck que la suma de las sumas de los segmentos iniciales de la lista formada por n veces el número uno es el n-ésimo número triangular; es decir que

```
  sumaSegmentosIniciales (genericReplicate n 1)
```

es igual a

```
  n * (n + 1) `div` 2
```

Soluciones

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

-- 1ª solución
-- ===========

sumaSegmentosIniciales :: [Integer] -> Integer
sumaSegmentosIniciales xs =
  sum [sum (take k xs) | k <- [1.. length xs]]

-- 2ª solución
-- ===========

sumaSegmentosIniciales2 :: [Integer] -> Integer
sumaSegmentosIniciales2 xs =
  sum (zipWith (*) [n,n-1..1] xs)
  where n = genericLength xs

-- 3ª solución
-- ===========

sumaSegmentosIniciales3 :: [Integer] -> Integer
sumaSegmentosIniciales3 xs =
  sum (scanl1 (+) xs)

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

-- La propiedad es
```

```haskell
prop_sumaSegmentosInicialesEquiv :: [Integer] -> Bool
prop_sumaSegmentosInicialesEquiv xs =
  all (== sumaSegmentosIniciales xs) [f xs | f <- [ sumaSegmentosIniciales2
                                                  , sumaSegmentosIniciales3]]

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

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

--    λ> sumaSegmentosIniciales [1..10^4]
--    166716670000
--    (2.42 secs, 7,377,926,824 bytes)
--    λ> sumaSegmentosIniciales2 [1..10^4]
--    166716670000
--    (0.01 secs, 4,855,176 bytes)
--
--    λ> sumaSegmentosIniciales2 [1..3*10^6]
--    4500004500001000000
--    (2.68 secs, 1,424,404,168 bytes)
--    λ> sumaSegmentosIniciales3 [1..3*10^6]
--    4500004500001000000
--    (1.54 secs, 943,500,384 bytes)

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_sumaSegmentosIniciales :: Positive Integer -> Bool
prop_sumaSegmentosIniciales (Positive n) =
  sumaSegmentosIniciales3 (genericReplicate n 1) ==
  n * (n + 1) `div` 2

-- La compronación es
--    λ> quickCheck prop_sumaSegmentosIniciales
--    +++ OK, passed 100 tests.
```

Ejercicio 86

Camino de máxima suma en una matriz

Caminante, no hay camino,
sino estelas en la mar.

Antonio Machado

Enunciado

Los caminos desde el extremo superior izquierdo (posición (1,1)) hasta el extremo inferior derecho (posición (3,4)) en la matriz

```
( 1  6 11  2 )
( 7 12  3  8 )
( 3  8  4  9 )
```

moviéndose en cada paso una casilla hacia abajo o hacia la derecha, son los siguientes:

```
1, 7,  3, 8, 4, 9
1, 7, 12, 8, 4, 9
1, 7, 12, 3, 4, 9
1, 7, 12, 3, 8, 9
1, 6, 12, 8, 4, 9
1, 6, 12, 3, 4, 9
```

```
1, 6, 12, 3, 8, 9
1, 6, 11, 3, 4, 9
1, 6, 11, 3, 8, 9
1, 6, 11, 2, 8, 9
```

Las sumas de los caminos son 32, 41, 36, 40, 40, 35, 39, 34, 38 y 37, respectivamente. El camino de máxima suma es el segundo (1, 7, 12, 8, 4, 9) que tiene una suma de 41.

Definir la función

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

tal que (caminoMaxSuma m) es un camino de máxima suma en la matriz m desde el extremo superior izquierdo hasta el extremo inferior derecho, moviéndose en cada paso una casilla hacia abajo o hacia la derecha. Por ejemplo,

```
λ> caminoMaxSuma (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
[1,7,12,8,4,9]
λ> sum (caminoMaxSuma (fromList 800 800 [1..]))
766721999
```

Soluciones

```
import Data.Matrix
import Test.QuickCheck

-- 1ª definición
-- =============

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

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

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

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

-- 2ª definición
-- =============

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

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

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

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

caminoMaxSuma3 :: Matrix Int -> [Int]
caminoMaxSuma3 m = reverse (snd (q ! (nf,nc)))
```

```
  where nf = nrows m
        nc = ncols m
        q  = caminoMaxSumaAux m

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

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

-- El generador es
instance Arbitrary a => Arbitrary (Matrix a) where
  arbitrary =  do
    m <- choose (1,7)
    n <- choose (1,7)
    xs <- Test.QuickCheck.vector (n*m)
    return (fromList m n xs)


-- Por ejemplo,
--    λ> sample' (arbitrary :: Gen (Matrix Int))
--    [( 0 )
--     ( 0 )
--     ( 0 )
--     ( 0 )
--    ,(  1  2 )
```

```
--      ( -1  1 )
--      ( -1 -2 )
--      (  1 -1 )
--      (  1  0 )
--      (  2  0 )
--      (  2 -2 )
--     ,( -4  4 -2 )
--      ( -2  0 -2 )
--      (  0 -1 -2 )
--      ( -4 -1  2 )
--     ,( -2  7 -3  1 -5 -3  5 )
--      (  0  2  7 -1 -5  7 -6 )
--      (  1  7 -8  1  6 -7  5 )
--      ( -4  7 -2 -7 -5  5 -8 )
--     ...

-- La propiedad es
prop_caminoMaxSuma :: Matrix Int -> Bool
prop_caminoMaxSuma m =
  x1 == x2 && x2 == x3
  where x1 = sum (caminoMaxSuma1 m)
        x2 = sum (caminoMaxSuma2 m)
        x3 = sum (caminoMaxSuma1 m)

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


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

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

```
--    (0.01 secs, 0 bytes)
```

Ejercicio 87

Combinaciones divisibles

El que espera desespera,
dice la voz popular.
¡Qué verdad tan verdadera!
La verdad es lo que es,
y sigue siendo verdad
aunque se piense al revés.

Antonio Machado

Enunciado

Definir la función

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

tal que (tieneCombinacionDivisible xs m) se verifica si existe alguna forma de combinar todos los elementos de la lista (con las operaciones suma o resta) de forma que el resultado sea divisible por m. Por ejemplo,

```
tieneCombinacionDivisible [1,3,4,6] 4  ==  True
tieneCombinacionDivisible [1,3,9]   2  ==  False
```

En el primer ejemplo, 1 - 2 + 3 + 4 + 6 = 12 es una combinación divisible por 4. En el segundo ejemplo, las combinaciones de [1,3,9] son

```
  1 + 3 + 9 =  13
 -1 + 3 + 9 =  11
  1 - 3 + 9 =   7
 -1 - 3 + 9 =   5
  1 + 3 - 9 =  -5
 -1 + 3 - 9 =  -7
  1 - 3 - 9 = -11
 -1 - 3 - 9 = -13
```

y ninguna de las 4 es divisible por 2.

Soluciones

```haskell
import Test.QuickCheck

-- 1ª solución
-- ===========

tieneCombinacionDivisible :: [Int] -> Int -> Bool
tieneCombinacionDivisible xs m =
  any esDivisible (valoresCombinaciones xs)
  where esDivisible x = x `mod` m == 0

-- (valoresCombinaciones xs) es la lista de los valores de todas las
-- combinaciones de todos los elementos de la lista con las operaciones
-- suma o resta. Por ejemplo,
--    λ> valoresCombinaciones [1,3,4,6]
--    [14,12,8,6,6,4,0,-2,2,0,-4,-6,-6,-8,-12,-14]
--    λ> valoresCombinaciones [1,3,-4,6]
--    [6,4,0,-2,14,12,8,6,-6,-8,-12,-14,2,0,-4,-6]
valoresCombinaciones :: [Int] -> [Int]
valoresCombinaciones []     = []
valoresCombinaciones [x]    = [x,-x]
valoresCombinaciones (x:xs) = concat [[y + x, y - x] | y <- ys]
  where ys = valoresCombinaciones xs

-- 2ª solución
-- ===========
```

```haskell
tieneCombinacionDivisible2 :: [Int] -> Int -> Bool
tieneCombinacionDivisible2 xs m =
  tieneCombinacionCongruente xs m 0

-- (tieneCombinacionCongruente xs m a) se verifica si existe alguna
-- forma de combinar todos los elementos de la lista xs (con las
-- operaciones suma o resta) de forma que el resultado sea congruente
-- con a módulo m. Por ejemplo,
--    tieneCombinacionCongruente [1,3,4,6] 4 0  ==  True
--    tieneCombinacionCongruente [1,3,4,6] 4 1  ==  False
--    tieneCombinacionCongruente [1,3,9] 2 0    ==  False
--    tieneCombinacionCongruente [1,3,9] 2 1    ==  True
tieneCombinacionCongruente :: [Int] -> Int -> Int -> Bool
tieneCombinacionCongruente []  _  _ = False
tieneCombinacionCongruente [x] m  a = (x - a) `mod` m == 0
tieneCombinacionCongruente (x:xs) m a =
  tieneCombinacionCongruente xs m (a-x) ||
  tieneCombinacionCongruente xs m (a+x)

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

-- La propiedad es
prop_tieneCombinacionDivisible :: [Int] -> Positive Int -> Bool
prop_tieneCombinacionDivisible xs (Positive m) =
  tieneCombinacionDivisible xs m == tieneCombinacionDivisible2 xs m

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=25}) prop_tieneCombinacionDivisible
--    +++ OK, passed 100 tests.

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

--    λ> (n,xs,m) = (200,[-n..n],sum [1..n])
--    (0.00 secs, 0 bytes)
--    λ> and [tieneCombinacionDivisible xs a | a <- [1..m]]
--    True
--    (4.74 secs, 6,536,494,976 bytes)
```

```
--    λ> and [tieneCombinacionDivisible2 xs a | a <- [1..m]]
--    True
--    (2.97 secs, 3,381,932,664 bytes)
```