Exercitium

Ejercicios de programación funcional con Haskell

(Volumen 1: curso 2013–14)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 14 de octubre de 2025

Índice general

1. Iguales al siguiente 9
2. Ordenación por el máximo 15
3. La bandera tricolor 19
4. Determinación de los elementos minimales 23
5. Mastermind 29
6. Primos consecutivos con media capicúa 35
7. Anagramas 39
8. Primos equidistantes 45
9. Suma si todos los valores son justos 51
10. Matriz de Toeplitz 57
11. Máximos locales 63
12. Lista cuadrada 67
13. Segmentos maximales con elementos consecutivos 71
14. Valores de polinomios representados con vectores 77
15. Ramas de un árbol 85
16. Alfabeto comenzando en un carácter 89
17. Numeración de ternas 93
18. Ordenación de estructuras 99
19. Emparejamiento binario 103

20. Amplia columnas **107**

21. Regiones determinadas por n rectas del plano **111**

22. Elemento más repetido de manera consecutiva **115**

23. Número de pares de elementos adyacentes iguales en una matriz **121**

24. Mayor producto de las ramas de un árbol **127**

25. Biparticiones de una lista **133**

26. Trenzado de listas **137**

27. Números triangulares con n cifras distintas **141**

28. Enumeración de árboles binarios **147**

29. Algún vecino menor **153**

30. Reiteración de una función **159**

31. Pim, Pam, Pum y divisibilidad **163**

32. Código de las alergias **167**

33. Índices de valores verdaderos **173**

34. Descomposiciones triangulares **179**

35. Número de inversiones **185**

36. Separación por posición **191**

37. Emparejamiento de árboles **197**

38. Eliminación de las ocurrencias aisladas **201**

39. Ordenada cíclicamente **205**

40. Órbita prima **211**

41. Divisores de un número con final dado **215**

42. Descomposiciones de x como sumas de n sumandos de una lista **221**

43. Selección hasta el primero que falla inclusive 225

44. Buscaminas 229

45. Mayor sucesión del problema 3n+1 235

46. Filtro booleano 239

47. Entero positivo de la cadena 243

48. N gramas 251

49. Sopa de letras 255

50. Intercalación de n copias 261

51. Eliminación de n elementos 265

52. Límite de sucesiones 271

53. Empiezan con mayúscula 275

54. Renombramiento de un árbol 281

55. Divide si todos son múltiplos 287

56. Ventana deslizante 293

57. Representación de Zeckendorf 297

58. Elemento más cercano que cumple una propiedad 303

59. Producto cartesiano de una familia de conjuntos 309

60. Todas tienen par 315

61. Sucesiones pucelanas 319

62. Producto de matrices como listas de listas 323

63. Inserción en árboles binarios de búsqueda 327

64. Matriz permutación 331

65. Números con todos sus dígitos primos 337

66. Cadenas de ceros y unos 343

67. Clausura de un conjunto respecto de una función **347**

68. Sustitución en una expresión aritmética **351**

69. Laberinto numérico **355**

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 2013–14.

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.

En cada ejercicio se muestra distintas soluciones, se comprueba con QuickCheck su equivalencia y se compara su eficiencia.

El código de los ejercicios del libro se encuentra en GitHub [4]

[1] https://jaalonso.github.io/exercitium

[2] https://jaalonso.github.io/cursos/i1m-13

[3] https://web.archive.org/web/20250614171615/https://www.cs.us.es/~jalonso/cursos/i1m-13/ejercicios/ejercicios-I1M-2013.pdf

[4] https://github.com/jaalonso/Exercitium

Ejercicio 1

Iguales al siguiente

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    igualesAlSiguiente :: Eq a => [a] -> [a]
-- tal que (igualesAlSiguiente xs) es la lista de los elementos de xs
-- que son iguales a su siguiente. Por ejemplo,
--    igualesAlSiguiente [1,2,2,2,3,3,4]  ==  [2,2,3]
--    igualesAlSiguiente [1..10]          ==  []
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Iguales_al_siguiente where

import Data.List (group)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

igualesAlSiguiente1 :: Eq a => [a] -> [a]
igualesAlSiguiente1 xs =
  [x | (x,y) <- zip xs (tail xs), x == y]

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

igualesAlSiguiente2 :: Eq a => [a] -> [a]
igualesAlSiguiente2 (x:y:zs) | x == y    = x : igualesAlSiguiente2 (y:zs)
                             | otherwise = igualesAlSiguiente2 (y:zs)
igualesAlSiguiente2 _                    = []
```

```haskell
-- 3ª solución
-- ===========

igualesAlSiguiente3 :: Eq a => [a] -> [a]
igualesAlSiguiente3 xs = concat [ys | (_:ys) <- group xs]

-- 4ª solución
-- ===========

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

-- 5ª solución
-- ===========

igualesAlSiguiente5 :: Eq a => [a] -> [a]
igualesAlSiguiente5 xs = tail =<< (group xs)

-- Nota: En la solución anterior se usa el operador (=<<) ya que
--    f =<< xs
-- es equivalente a
--    concat (map f xs)
-- Por ejemplo,
--    ghci> tail =<< [[1],[2,3,4],[9,7],[6]]
--    [3,4,7]
--    ghci> init =<< [[1],[2,3,4],[9,7],[6]]
--    [2,3,9]
--    ghci> reverse =<< [[1],[2,3,4],[9,7],[6]]
--    [1,4,3,2,7,9,6]
--    ghci> (take 2) =<< [[1],[2,3,4],[9,7],[6]]
--    [1,2,3,9,7,6]
--    ghci> (drop 2) =<< [[1],[2,3,4],[9,7],[6]]
--    [4]
--    ghci> (++[0]) =<< [[1],[2,3,4],[3,7],[6]]
--    [1,0,2,3,4,0,3,7,0,6,0]

-- 6ª solución
-- ===========

igualesAlSiguiente6 :: Eq a => [a] -> [a]
igualesAlSiguiente6 = (tail =<<) . group

-- 7ª solución
-- ===========
```

```haskell
igualesAlSiguiente7 :: Eq a => [a] -> [a]
igualesAlSiguiente7 xs = concatMap tail (group xs)

-- Nota: En la solución anterior se usa la función ya que
--    concatMap f xs
-- es equivalente a
--    concat (map f xs)
-- Por ejemplo,
--    ghci> concatMap tail [[1],[2,3,4],[9,7],[6]]
--    [3,4,7]

-- 8ª solución
-- ===========

igualesAlSiguiente8 :: Eq a => [a] -> [a]
igualesAlSiguiente8 = (concatMap tail) . group

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [Int])  -> Spec
specG igualesAlSiguiente = do
  it "e1" $
    igualesAlSiguiente [1,2,2,2,3,3,4]  `shouldBe`  [2,2,3]
  it "e2" $
    igualesAlSiguiente [1..10] `shouldBe` []

spec :: Spec
spec = do
  describe "def. 1" $ specG igualesAlSiguiente1
  describe "def. 2" $ specG igualesAlSiguiente2
  describe "def. 3" $ specG igualesAlSiguiente3
  describe "def. 4" $ specG igualesAlSiguiente4
  describe "def. 5" $ specG igualesAlSiguiente5
  describe "def. 6" $ specG igualesAlSiguiente6
  describe "def. 7" $ specG igualesAlSiguiente7
  describe "def. 8" $ specG igualesAlSiguiente8

-- La verificación es
--    λ> verifica
--    16 examples, 0 failures
```

```haskell
-- Equivalencia de las definiciones
-- ================================

-- La propiedad es
prop_igualesAlSiguiente :: [Int] -> Bool
prop_igualesAlSiguiente xs =
  all (== igualesAlSiguiente1 xs)
      [igualesAlSiguiente2 xs,
       igualesAlSiguiente3 xs,
       igualesAlSiguiente4 xs,
       igualesAlSiguiente5 xs,
       igualesAlSiguiente6 xs,
       igualesAlSiguiente7 xs,
       igualesAlSiguiente8 xs]

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

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

-- La comparación es
--    > ej = concatMap show [1..10^6]
--    (0.01 secs, 446,752 bytes)
--    λ> length ej
--    5888896
--    (0.16 secs, 669,787,856 bytes)
--    λ> length (show (igualesAlSiguiente1 ej))
--    588895
--    (1.60 secs, 886,142,944 bytes)
--    λ> length (show (igualesAlSiguiente2 ej))
--    588895
--    (1.95 secs, 1,734,143,816 bytes)
--    λ> length (show (igualesAlSiguiente3 ej))
--    588895
--    (1.81 secs, 1,178,232,104 bytes)
--    λ> length (show (igualesAlSiguiente4 ej))
--    588895
--    (1.43 secs, 1,932,010,304 bytes)
--    λ> length (show (igualesAlSiguiente5 ej))
--    588895
--    (0.40 secs, 2,016,810,320 bytes)
--    λ> length (show (igualesAlSiguiente6 ej))
--    588895
```

```
--    (0.32 secs, 1,550,409,984 bytes)
--    λ> length (show (igualesAlSiguiente7 ej))
--    588895
--    (0.34 secs, 1,550,410,104 bytes)
--    λ> length (show (igualesAlSiguiente8 ej))
--    588895
--    (0.33 secs, 1,550,410,024 bytes)
```

Ejercicio 2

Ordenación por el máximo

```
-- ---------------------------------------------------------------------
-- Definir la función
--    ordenadosPorMaximo :: Ord a => [[a]] -> [[a]]
-- tal que (ordenadosPorMaximo xss) es la lista de los elementos de xss
-- ordenada por sus máximos. Por ejemplo,
--    λ> ordenadosPorMaximo [[3,2],[6,7,5],[1,4]]
--    [[3,2],[1,4],[6,7,5]]
--    λ> ordenadosPorMaximo [[1],[0,1]]
--    [[1],[0,1]]
--    λ> ordenadosPorMaximo [[0,1],[1]]
--    [[0,1],[1]]
--    λ> ordenadosPorMaximo ["este","es","el","primero"]
--    ["el","primero","es","este"]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
module Ordenados_por_maximo where

import Data.List (sort, sortOn, sortBy)
import Data.Ord (comparing)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

ordenadosPorMaximo1 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo1 xss =
  [ys | (_,_,ys) <- sort [(maximum xs,n,xs) | (n,xs) <- zip [0..] xss]]

-- 2ª solución
-- ===========
```

```haskell
ordenadosPorMaximo2 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo2 xss =
  map (\(_,_,ys) -> ys) (sort [(maximum xs,n,xs) | (n,xs) <- zip [0..] xss])

-- 3ª solución
-- ===========

ordenadosPorMaximo3 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo3 = sortBy (comparing maximum)

-- 4ª solución
-- ===========

ordenadosPorMaximo4 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo4 = sortOn maximum

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([[Int]] -> [[Int]]) -> Spec
specG ordenadosPorMaximo = do
  it "e1" $
    ordenadosPorMaximo [[3,2],[6,7,5],[1,4]] `shouldBe`
      [[3,2],[1,4],[6,7,5]]
  it "e2" $
    ordenadosPorMaximo [[1],[0,1]] `shouldBe`
      [[1],[0,1]]
  it "e3" $
    ordenadosPorMaximo [[0,1],[1]] `shouldBe`
      [[0,1],[1]]

spec :: Spec
spec = do
  describe "def. 1" $ specG ordenadosPorMaximo1
  describe "def. 2" $ specG ordenadosPorMaximo2
  describe "def. 3" $ specG ordenadosPorMaximo3
  describe "def. 4" $ specG ordenadosPorMaximo4

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures
```

```haskell
-- Equivalencia de las definiciones
-- ================================

-- La propiedad es
prop_ordenadosPorMaximo :: [[Int]] -> Bool
prop_ordenadosPorMaximo xss =
  all (== ordenadosPorMaximo1 yss)
      [ordenadosPorMaximo2 yss,
       ordenadosPorMaximo3 yss,
       ordenadosPorMaximo4 yss]
  where yss = filter (not .null) xss

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

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

-- La comparación es
--    λ> length (ordenadosPorMaximo1 [[1..n] | n <- [1..10^4]])
--    10000
--    (6.52 secs, 3,607,440,800 bytes)
--    λ> length (ordenadosPorMaximo2 [[1..n] | n <- [1..10^4]])
--    10000
--    (7.30 secs, 3,607,600,776 bytes)
--    λ> length (ordenadosPorMaximo3 [[1..n] | n <- [1..10^4]])
--    10000
--    (7.53 secs, 3,604,559,928 bytes)
--    λ> length (ordenadosPorMaximo4 [[1..n] | n <- [1..10^4]])
--    10000
--    (7.09 secs, 3,606,159,952 bytes)
```

Ejercicio 3

La bandera tricolor

```
-- ---------------------------------------------------------------------
-- El problema de la bandera tricolor consiste en lo siguiente: Dada un
-- lista de objetos xs que pueden ser rojos, amarillos o morados, se pide
-- devolver una lista ys que contiene los elementos de xs, primero los
-- rojos, luego los amarillos y por último los morados.
--
-- Definir el tipo de dato Color para representar los colores con los
-- constructores R, A y M correspondientes al rojo, azul y morado y la
-- función
--    banderaTricolor :: [Color] -> [Color]
-- tal que (banderaTricolor xs) es la bandera tricolor formada con los
-- elementos de xs. Por ejemplo,
--    banderaTricolor [M,R,A,A,R,R,A,M,M]  ==  [R,R,R,A,A,A,M,M,M]
--    banderaTricolor [M,R,A,R,R,A]        ==  [R,R,R,A,A,M]
-- ---------------------------------------------------------------------

module Bandera_tricolor where

import Data.List (sort)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

data Color = R | A | M
  deriving (Show, Eq, Ord, Enum)

-- 1ª solución
-- ===========

banderaTricolor1 :: [Color] -> [Color]
banderaTricolor1 = sort

-- 2ª solución
```

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

banderaTricolor2 :: [Color] -> [Color]
banderaTricolor2 xs =
  [x | x <- xs, x == R] ++ [x | x <- xs, x == A] ++ [x | x <- xs, x == M]

-- 3ª solución
-- ===========

banderaTricolor3 :: [Color] -> [Color]
banderaTricolor3 xs =
  concat [[x | x <- xs, x == c] | c <- [R,A,M]]

-- 4ª solución
-- ===========

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

-- 5ª solución
-- ===========

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Color] -> [Color])  -> Spec
specG banderaTricolor = do
  it "e1" $
    banderaTricolor [M,R,A,A,R,R,A,M,M] `shouldBe` [R,R,R,A,A,A,M,M,M]
```

```
  it "e2" $
    banderaTricolor [M,R,A,R,R,A] `shouldBe` [R,R,R,A,A,M]

spec :: Spec
spec = do
  describe "def. 1" $ specG banderaTricolor1
  describe "def. 2" $ specG banderaTricolor2
  describe "def. 3" $ specG banderaTricolor3
  describe "def. 4" $ specG banderaTricolor4
  describe "def. 5" $ specG banderaTricolor5

-- La verificación es
--    λ> verifica
--    10 examples, 0 failures

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

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

-- La propiedad es
prop_banderaTricolor :: [Color] -> Bool
prop_banderaTricolor xs =
  all (== banderaTricolor1 xs)
      [banderaTricolor2 xs,
       banderaTricolor3 xs,
       banderaTricolor4 xs,
       banderaTricolor5 xs]

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

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

-- La comparación es
--    λ> bandera n = concat [replicate n c | c <- [M,R,A]]
--    λ> :set +s
--    λ> length (banderaTricolor1 (bandera 1000000))
--    3000000
--    (2.92 secs, 1,024,602,024 bytes)
--    λ> length (banderaTricolor2 (bandera 1000000))
--    3000000
```

```
--    (1.91 secs, 1,168,601,536 bytes)
--    λ> length (banderaTricolor3 (bandera 1000000))
--    3000000
--    (3.55 secs, 1,440,602,120 bytes)
--    λ> length (banderaTricolor4 (bandera 1000000))
--    3000000
--    (1.30 secs, 1,000,601,376 bytes)
--    λ> length (banderaTricolor5 (bandera 1000000))
--    3000000
--    (1.56 secs, 1,245,461,400 bytes)
```

Ejercicio 4

Determinación de los elementos minimales

```
-- ----------------------------------------------------------------------
-- Definir la función
--    minimales :: Eq a => [[a]] -> [[a]]
-- tal que (minimales xss) es la lista de los elementos de xss que no
-- están contenidos en otros elementos de xss. Por ejemplo,
--    minimales [[1,3],[2,3,1],[3,2,5]]        ==  [[2,3,1],[3,2,5]]
--    minimales [[1,3],[2,3,1],[3,2,5],[3,1]]  ==  [[2,3,1],[3,2,5]]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module ElementosMinimales where

import Data.List (nub, delete, (\\))
import Data.Set (isSubsetOf, fromList)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

minimales1 :: Eq a => [[a]] -> [[a]]
minimales1 xss =
  [xs | xs <- xss,
        null [ys | ys <- xss, subconjuntoPropio xs ys]]

-- (subconjuntoPropio xs ys) se verifica si xs es un subconjunto propio
-- de ys. Por ejemplo,
--    subconjuntoPropio [1,3] [3,1,3]    ==  False
```

```
--    subconjuntoPropio [1,3,1] [3,1,2]  ==  True
subconjuntoPropio :: Eq a => [a] -> [a] -> Bool
subconjuntoPropio xs ys = subconjuntoPropio' (nub xs) (nub ys)
  where
    subconjuntoPropio' _ [] = False
    subconjuntoPropio' [] _ = True
    subconjuntoPropio' (x:xs') ys' =
      x `elem` ys' && subconjuntoPropio xs' (delete x ys')

-- 2ª solución
-- ===========

minimales2 :: Eq a => [[a]] -> [[a]]
minimales2 xss = filter noTieneSubconjunto xss
  where
    noTieneSubconjunto xs = not (any (subconjuntoPropio xs) xss)

-- 3ª solución
-- ===========

minimales3 :: Eq a => [[a]] -> [[a]]
minimales3 xss =
  [xs | xs <- xss, all (not . subconjuntoPropio2 xs) xss]

subconjuntoPropio2 :: Eq a => [a] -> [a] -> Bool
subconjuntoPropio2 xs ys =
  xs' /= ys' && null (xs' \\ ys')
  where xs' = nub xs
        ys' = nub ys

-- 4ª solución
-- ===========

minimales4 :: Eq a => [[a]] -> [[a]]
minimales4 xss =
  [xs | xs <- xss, not (any (subconjuntoPropio2 xs) xss)]

-- 5ª solución
-- ===========

minimales5 :: Ord a => [[a]] -> [[a]]
minimales5 xss =
  [xs | xs <- xss, (not . any (subconjuntoPropio3 xs)) xss]

subconjuntoPropio3 :: Ord a => [a] -> [a] -> Bool
```

```haskell
subconjuntoPropio3 xs ys =
  setXs /= setYs && setXs `isSubsetOf` setYs
  where setXs = fromList xs
        setYs = fromList ys

-- 6ª solución
-- ===========

minimales6 :: Eq a => [[a]] -> [[a]]
minimales6 = foldr agregarSiMinimal []
  where
    agregarSiMinimal xs yss
      | any (subconjuntoPropio xs) yss = yss
      | otherwise = xs : filter (not . flip subconjuntoPropio xs) yss

-- 7ª solución
-- ===========

minimales7 :: Eq a => [[a]] -> [[a]]
minimales7 xss =
  [xs | xs <- xss,
        not (or [subconjuntoPropio xs ys | ys <- xss])]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([[Int]] -> [[Int]]) -> Spec
specG minimales = do
  it "e1" $
    minimales [[1,3],[2,3,1],[3,2,5]]        `shouldBe`  [[2,3,1],[3,2,5]]
  it "e2" $
    minimales [[1,3],[2,3,1],[3,2,5],[3,1]]  `shouldBe`  [[2,3,1],[3,2,5]]

spec :: Spec
spec = do
  describe "def. 1" $ specG minimales1
  describe "def. 2" $ specG minimales2
  describe "def. 3" $ specG minimales3
  describe "def. 4" $ specG minimales4
  describe "def. 5" $ specG minimales5
  describe "def. 6" $ specG minimales6
  describe "def. 7" $ specG minimales7
```

```
-- La verificación es
--    λ> verifica
--    14 examples, 0 failures

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

-- La propiedad es
prop_minimales :: [[Int]] -> Bool
prop_minimales xss =
  all (== minimales1 xss)
      [minimales2 xss,
       minimales3 xss,
       minimales4 xss,
       minimales5 xss,
       minimales6 xss,
       minimales7 xss]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=40}) prop_minimales
--    +++ OK, passed 100 tests.

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

-- La comparación es
--    λ> length (minimales1 [[1..n] | n <- [1..100]])
--    1
--    (2.27 secs, 973,011,248 bytes)
--    λ> length (minimales2 [[1..n] | n <- [1..100]])
--    1
--    (2.19 secs, 973,170,440 bytes)
--    λ> length (minimales3 [[1..n] | n <- [1..100]])
--    1
--    (0.14 secs, 37,972,696 bytes)
--    λ> length (minimales4 [[1..n] | n <- [1..100]])
--    1
--    (0.12 secs, 37,804,728 bytes)
--    λ> length (minimales5 [[1..n] | n <- [1..100]])
--    1
--    (0.05 secs, 39,607,648 bytes)
--    λ> length (minimales6 [[1..n] | n <- [1..100]])
--    1
--    (0.12 secs, 36,018,976 bytes)
```

```
--    λ> length (minimales7 [[1..n] | n <- [1..100]])
--    1
--    (2.16 secs, 973,840,760 bytes)
--
--    λ> length (minimales3 [[1..n] | n <- [1..250]])
--    1
--    (3.47 secs, 533,688,496 bytes)
--    λ> length (minimales4 [[1..n] | n <- [1..250]])
--    1
--    (3.46 secs, 532,668,528 bytes)
--    λ> length (minimales5 [[1..n] | n <- [1..250]])
--    1
--    (0.27 secs, 579,544,232 bytes)
--    λ> length (minimales6 [[1..n] | n <- [1..250]])
--    1
--    (3.30 secs, 740,152,496 bytes)
```

Ejercicio 5

Mastermind

```
-- ---------------------------------------------------------------------
-- El Mastermind es un juego que consiste en deducir un código
-- numérico formado por una lista de números distintos. Cada vez que se
-- empieza una partida, el programa debe elegir un código, que será lo
-- que el jugador debe adivinar en la menor cantidad de intentos
-- posibles. Cada intento consiste en una propuesta de un código posible
-- que propone el jugador, y una respuesta del programa. Las respuestas
-- le darán pistas al jugador para que pueda deducir el código.
--
-- Estas pistas indican cuán cerca estuvo el número propuesto de la
-- solución a través de dos valores: la cantidad de aciertos es la
-- cantidad de dígitos que propuso el jugador que también están en el
-- código en la misma posición. La cantidad de coincidencias es la
-- cantidad de digitos que propuso el jugador que también están en el
-- código pero en una posición distinta.
--
-- Por ejemplo, si el código que eligió el programa es el [2,6,0,7], y
-- el jugador propone el [1,4,0,6], el programa le debe responder un
-- acierto (el 0, que está en el código original en el mismo lugar, el
-- tercero), y una coincidencia (el 6, que también está en el código
-- original, pero en la segunda posición, no en el cuarto como fue
-- propuesto). Si el jugador hubiera propuesto el [3,5,9,1], habría
-- obtenido como respuesta ningún acierto y ninguna coincidencia, ya que
-- no hay números en común con el código original, y si se obtienen
-- cuatro aciertos es porque el jugador adivinó el código y ganó el
-- juego.
--
-- Definir la función
--    mastermind :: [Int] -> [Int] -> (Int,Int)
-- tal que (mastermind xs ys) es el par formado por los números de
-- aciertos y de coincidencias entre xs e ys. Por ejemplo,
--    mastermind [2,6,0,7] [1,4,0,6]  ==  (1,1)
```

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

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Mastermind where

import Data.List (nub)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========
mastermind1 :: [Int] -> [Int] -> (Int,Int)
mastermind1 xs ys =
  (length (aciertos xs ys),length (coincidencias xs ys))

-- (aciertos xs ys) es la lista de aciertos entre xs e ys. Por ejemplo,
--    aciertos [2,6,0,7] [1,4,0,6]  ==  [0]
aciertos :: Eq a => [a] -> [a] -> [a]
aciertos xs ys =
  [x | (x,y) <- zip xs ys, x == y]

-- (coincidencia xs ys) es la lista de coincidencias entre xs e ys. Por
-- ejemplo,
--    coincidencias [2,6,0,7] [1,4,0,6]  ==  [6]
coincidencias :: Eq a => [a] -> [a] -> [a]
coincidencias xs ys =
  [x | x <- xs, x `elem` ys, x `notElem` zs]
  where zs = aciertos xs ys

-- 2ª solución
-- ===========
mastermind2 :: [Int] -> [Int] -> (Int,Int)
mastermind2 xs ys = aux xs ys
  where aux [] [] = (0,0)
        aux (x:xs') (z:zs)
          | x == z      = (a+1,b)
          | x `elem` ys = (a,b+1)
          | otherwise   = (a,b)
          where (a,b) = aux xs' zs
```

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [Int] -> (Int,Int))  -> Spec
specG mastermind = do
  it "e1" $
    mastermind [2,6,0,7] [1,4,0,6] `shouldBe` (1,1)
  it "e2" $
    mastermind [2,6,0,7] [3,5,9,1] `shouldBe` (0,0)
  it "e3" $
    mastermind [2,6,0,7] [1,6,0,4] `shouldBe` (2,0)
  it "e4" $
    mastermind [2,6,0,7] [2,6,0,7] `shouldBe` (4,0)

spec :: Spec
spec = do
  describe "def. 1" $ specG mastermind1
  describe "def. 2" $ specG mastermind2
  describe "def. 3" $ specG mastermind3

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- Generador para listas de 4 elementos elgidos del 1 al 6. Por ejemplo,
--    λ> sample genMastermind
--    [2,4,6,1]
--    [6,5,1,2]
--    [3,2,6,4]
--    [1,3,2,5]
--    [5,2,6,4]
--    [3,2,4,1]
```

```
--    [2,6,5,4]
--    [4,1,2,3]
--    [6,4,2,5]
--    [4,5,3,2]
--    [6,3,4,2]
genMastermind :: Gen [Int]
genMastermind = do
  elementos <- shuffle [1..6]
  return (take 4 elementos)

-- Generador para el juego clásico de Mastermind (4 posiciones, 6
-- colores). Por ejemplo,
--    λ> sample genParesMastermind
--    ([3,5,6,2],[6,3,4,2])
--    ([1,4,3,2],[4,2,1,3])
--    ([1,5,3,6],[2,6,4,1])
--    ([5,6,1,4],[1,6,4,2])
--    ([2,6,4,3],[3,6,4,1])
--    ([6,1,3,5],[6,1,3,5])
--    ([6,4,5,1],[3,2,4,5])
--    ([5,6,4,2],[1,4,6,2])
--    ([1,5,3,6],[3,4,2,6])
--    ([6,1,5,3],[1,4,5,6])
--    ([1,2,4,5],[4,1,2,5])
genParesMastermind :: Gen ([Int], [Int])
genParesMastermind = do
  xs <- genMastermind
  ys <- genMastermind
  return (xs, ys)

prop_Mastermind :: Property
prop_Mastermind = forAll genParesMastermind $ \(xs, ys) ->
  all (== mastermind1 xs ys)
      [mastermind2 xs ys,
       mastermind3 xs ys]

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

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

-- La comparación es
--    λ> mastermind1 [1..20000] [1,3..40000]
```

```
--    (1,9999)
--    (2.14 secs, 12,442,128 bytes)
--    λ> mastermind2 [1..20000] [1,3..40000]
--    (1,9999)
--    (2.01 secs, 11,969,360 bytes)
--    λ> mastermind3 [1..20000] [1,3..40000]
--    (1,9999)
--    (5.60 secs, 11,322,176 bytes)
--
--    λ> mastermind1 [1..20000] [1..20000]
--    (20000,0)
--    (5.02 secs, 15,962,800 bytes)
--    λ> mastermind2 [1..20000] [1..20000]
--    (20000,0)
--    (0.02 secs, 12,978,344 bytes)
--    λ> mastermind3 [1..20000] [1..20000]
--    (20000,0)
--    (3.89 secs, 12,122,840 bytes)
```

Ejercicio 6

Primos consecutivos con media capicúa

```haskell
-- ---------------------------------------------------------------------
-- Definir la lista
--    primosConsecutivosConMediaCapicua :: [(Int,Int,Int)]
-- formada por las ternas (x,y,z) tales que x e y son primos
-- consecutivos cuya media, z, es capicúa. Por ejemplo,
--    λ> take 5 primosConsecutivosConMediaCapicua
--    [(3,5,4),(5,7,6),(7,11,9),(97,101,99),(109,113,111)]
--
-- Calcular cuántos hay anteriores a 2014.
-- ---------------------------------------------------------------------

module Primos_consecutivos_con_media_capicua where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Data.Numbers.Primes (primes)

-- 1ª solución
-- ===========

primosConsecutivosConMediaCapicua1 :: [(Int,Int,Int)]
primosConsecutivosConMediaCapicua1 =
  [(x,y,z) | (x,y) <- zip primos (tail primos),
             let z = (x + y) `div` 2,
             capicua z]

-- (primo x) se verifica si x es primo. Por ejemplo,
--    primo 7  ==  True
--    primo 8  ==  False
primo :: Int -> Bool
```

```
primo x = [y | y <- [1..x], x `rem` y == 0] == [1,x]

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

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

-- 2ª solución
-- ===========

primosConsecutivosConMediaCapicua2 :: [(Int,Int,Int)]
primosConsecutivosConMediaCapicua2 =
  [(x,y,z) | (x,y) <- zip (tail primes) (drop 2 primes),
             let z = (x + y) `div` 2,
             capicua z]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: [(Int,Int,Int)] -> Spec
specG primosConsecutivosConMediaCapicua = do
  it "e1" $
    take 5 primosConsecutivosConMediaCapicua `shouldBe`
    [(3,5,4),(5,7,6),(7,11,9),(97,101,99),(109,113,111)]

spec :: Spec
spec = do
  describe "def. 1" $ specG primosConsecutivosConMediaCapicua1
  describe "def. 2" $ specG primosConsecutivosConMediaCapicua2

-- La verificación es
--    λ> verifica
--    2 examples, 0 failures

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

```haskell
prop_equivalencia :: Int -> Bool
prop_equivalencia n =
  take n primosConsecutivosConMediaCapicua1 == take n primosConsecutivosConMediaCapicua2

-- La comprobación es
--    λ> prop_equivalencia 30
--    True

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

-- La comparación es
--    λ> primosConsecutivosConMediaCapicua1 !! 30
--    (12919,12923,12921)
--    (3.16 secs, 1,877,340,488 bytes)
--    λ> primosConsecutivosConMediaCapicua2 !! 30
--    (12919,12923,12921)
--    (0.01 secs, 9,236,008 bytes)

-- Cálculo
-- =======

-- El cálculo es
--    λ> length (takeWhile (\(x,y,z) -> y < 2014) primosConsecutivosConMediaCapicua2)
--    20
```

Ejercicio 7

Anagramas

```haskell
-- ----------------------------------------------------------------------
-- Una palabra es una anagrama de otra si se puede obtener permutando
-- sus letras. Por ejemplo, mora y roma son anagramas de amor.
--
-- Definir la función
--    anagramas :: String -> [String] -> [String]
-- tal que (anagramas x ys) es la lista de los elementos de ys que son
-- anagramas de x. Por ejemplo,
--    λ> anagramas "amor" ["Roma","mola","loma","moRa", "rama"]
--    ["Roma","moRa"]
--    λ> anagramas "rama" ["aMar","amaRa","roMa","marr","aRma"]
--    ["aMar","aRma"]
-- ----------------------------------------------------------------------

module Anagramas where

import Data.List (sort)
import Data.Char (toLower)
import Data.Function (on)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

anagramas1 :: String -> [String] -> [String]
anagramas1 _ [] = []
anagramas1 x (y:ys) | sonAnagramas1 x y = y : anagramas1 x ys
                    | otherwise         = anagramas1 x ys

-- (sonAnagramas1 xs ys) se verifica si xs e ys son anagramas. Por
-- ejemplo,
```

```
--    sonAnagramas1 "amor" "Roma"  ==  True
--    sonAnagramas1 "amor" "mola"  ==  False
sonAnagramas1 :: String -> String -> Bool
sonAnagramas1 xs ys =
  sort (map toLower xs) == sort (map toLower ys)

-- 2ª solución
-- ===========

anagramas2 :: String -> [String] -> [String]
anagramas2 _ [] = []
anagramas2 x (y:ys) | sonAnagramas2 x y = y : anagramas2 x ys
                    | otherwise         = anagramas2 x ys

sonAnagramas2 :: String -> String -> Bool
sonAnagramas2 xs ys =
  (sort . map toLower) xs == (sort . map toLower) ys

-- 3ª solución
-- ===========

anagramas3 :: String -> [String] -> [String]
anagramas3 _ [] = []
anagramas3 x (y:ys) | sonAnagramas3 x y = y : anagramas3 x ys
                    | otherwise         = anagramas3 x ys

sonAnagramas3 :: String -> String -> Bool
sonAnagramas3 = (==) `on` (sort . map toLower)

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

-- 4ª solución
-- ===========

anagramas4 :: String -> [String] -> [String]
anagramas4 x ys = [y | y <- ys, sonAnagramas1 x y]

-- 5ª solución
-- ===========
```

```haskell
anagramas5 :: String -> [String] -> [String]
anagramas5 x = filter (`sonAnagramas1` x)

-- 6ª solución
-- ===========

anagramas6 :: String -> [String] -> [String]
anagramas6 x = filter (((==) `on` (sort . map toLower)) x)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (String -> [String] -> [String]) -> Spec
specG anagramas = do
  it "e1" $
    anagramas "amor" ["Roma","mola","loma","moRa", "rama"] `shouldBe`
      ["Roma","moRa"]
  it "e2" $
    anagramas "rama" ["aMar","amaRa","roMa","marr","aRma"] `shouldBe`
      ["aMar","aRma"]

spec :: Spec
spec = do
  describe "def. 1" $ specG anagramas1
  describe "def. 2" $ specG anagramas2
  describe "def. 3" $ specG anagramas3
  describe "def. 4" $ specG anagramas4
  describe "def. 5" $ specG anagramas5
  describe "def. 6" $ specG anagramas6

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- Genera un carácter. Por ejemplo.
--    λ> sample genCaracter
--    'b'
--    'O'
```

```
--    'Y'
--    't'
--    'B'
--    'm'
--    'S'
--    'z'
--    'e'
--    'c'
--    'c'
genCaracter :: Gen Char
genCaracter = elements (['a'..'z'] ++ ['A'..'Z'])

-- Genera una palabra. Por ejemplo.
--    λ> sample genPalabra
--    ""
--    "U"
--    ""
--    ""
--    "SBNrDsv"
--    "L"
--    ""
--    "WEbZepkEMiOT"
--    "TPC"
--    "MQKbhuiOSoiLhGKc"
--    "TTUujizoQZxLtyKTH"
genPalabra :: Gen String
genPalabra = listOf genCaracter

-- Genera una lista de palabras.
genPalabras :: Gen [String]
genPalabras = listOf genPalabra

-- Propiedad con generador personalizado
prop_equivalencia :: Property
prop_equivalencia =
  forAll genPalabra $ \x ->
  forAll genPalabras $ \ys ->
  all (== anagramas1 x ys)
      [anagramas2 x ys,
       anagramas3 x ys,
       anagramas4 x ys,
       anagramas5 x ys,
       anagramas6 x ys]

-- La comprobación es
```

```
--    λ> quickCheck prop_equivalencia
--    +++ OK, passed 100 tests.
--    (0.16 secs, 240,359,144 bytes)
```

Ejercicio 8

Primos equidistantes

```
-- ---------------------------------------------------------------------
-- Definir la función
--    primosEquidistantes :: Integer -> [(Integer,Integer)]
-- tal que (primosEquidistantes k) es la lista de los pares de primos
-- cuya diferencia es k. Por ejemplo,
--    take 3 (primosEquidistantes 2)  ==  [(3,5),(5,7),(11,13)]
--    take 3 (primosEquidistantes 4)  ==  [(7,11),(13,17),(19,23)]
--    take 3 (primosEquidistantes 6)  ==  [(23,29),(31,37),(47,53)]
--    take 3 (primosEquidistantes 8)  ==  [(89,97),(359,367),(389,397)]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Primos_equidistantes where

import Data.List (unfoldr, tails)
import Data.Numbers.Primes (primes)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)

-- 1ª solución
-- ===========

primosEquidistantes1 :: Integer -> [(Integer,Integer)]
primosEquidistantes1 k = aux primos
  where aux (x:y:ps) | y - x == k = (x,y) : aux (y:ps)
                     | otherwise  = aux (y:ps)

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

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

-- 2ª solución
-- ===========

primosEquidistantes2 :: Integer -> [(Integer,Integer)]
primosEquidistantes2 k = aux primos2
  where aux (x:y:ps) | y - x == k = (x,y) : aux (y:ps)
                     | otherwise  = aux (y:ps)

primos2 :: [Integer]
primos2 = criba [2..]
  where criba (p:ps) = p : criba [n | n <- ps, mod n p /= 0]

-- 3ª solución
-- ===========

primosEquidistantes3 :: Integer -> [(Integer,Integer)]
primosEquidistantes3 k =
  [(x,y) | (x,y) <- zip primos2 (tail primos2)
         , y - x == k]

-- 4ª solución
-- ===========

primosEquidistantes4 :: Integer -> [(Integer,Integer)]
primosEquidistantes4 k = aux primes
  where aux (x:y:ps) | y - x == k = (x,y) : aux (y:ps)
                     | otherwise  = aux (y:ps)

-- 5ª solución
-- ===========

primosEquidistantes5 :: Integer -> [(Integer,Integer)]
primosEquidistantes5 k =
  [(x,y) | (x,y) <- zip primes (tail primes)
         , y - x == k]

-- 6ª solución
-- ===========
```

```haskell
primosEquidistantes6 :: Integer -> [(Integer,Integer)]
primosEquidistantes6 k =
  [(p, q) | (p:q:_) <- tails primes, q - p == k]

-- 7ª solución
-- ===========

primosEquidistantes7 :: Integer -> [(Integer, Integer)]
primosEquidistantes7 k = unfoldr aux (primes)
  where
    aux (p:q:ps)
      | q - p == k  = Just ((p,q), q:ps)
      | otherwise   = aux (q:ps)
    aux _ = Nothing

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Integer -> [(Integer,Integer)]) -> Spec
specG primosEquidistantes = do
  it "e1" $
    take 3 (primosEquidistantes 2) `shouldBe` [(3,5),(5,7),(11,13)]
  it "e2" $
    take 3 (primosEquidistantes 4) `shouldBe` [(7,11),(13,17),(19,23)]
  it "e3" $
    take 3 (primosEquidistantes 6) `shouldBe` [(23,29),(31,37),(47,53)]
  it "e4" $
    take 3 (primosEquidistantes 8) `shouldBe` [(89,97),(359,367),(389,397)]

spec :: Spec
spec = do
  describe "def. 1" $ specG primosEquidistantes1
  describe "def. 2" $ specG primosEquidistantes2
  describe "def. 3" $ specG primosEquidistantes3
  describe "def. 4" $ specG primosEquidistantes4
  describe "def. 5" $ specG primosEquidistantes5
  describe "def. 6" $ specG primosEquidistantes6
  describe "def. 7" $ specG primosEquidistantes7

-- La verificación es
--    λ> verifica
--    28 examples, 0 failures
```

```haskell
-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_primosEquidistantes :: Int -> Integer -> Bool
prop_primosEquidistantes n k =
  all (== take n (primosEquidistantes1 k))
      [take n (f k) | f <- [primosEquidistantes2,
                            primosEquidistantes3,
                            primosEquidistantes4,
                            primosEquidistantes5,
                            primosEquidistantes6,
                            primosEquidistantes7]]

-- La comprobación es
--    λ> prop_primosEquidistantes 100 4
--    True

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

-- La comparación es
--    λ> primosEquidistantes1 4 !! 200
--    (9829,9833)
--    (2.60 secs, 1,126,458,272 bytes)
--    λ> primosEquidistantes2 4 !! 200
--    (9829,9833)
--    (0.44 secs, 249,622,048 bytes)
--    λ> primosEquidistantes3 4 !! 200
--    (9829,9833)
--    (0.36 secs, 207,549,592 bytes)
--    λ> primosEquidistantes4 4 !! 200
--    (9829,9833)
--    (0.02 secs, 4,012,848 bytes)
--    λ> primosEquidistantes5 4 !! 200
--    (9829,9833)
--    (0.01 secs, 7,085,072 bytes)
--    λ> primosEquidistantes6 4 !! 200
--    (9829,9833)
--    (0.02 secs, 3,628,024 bytes)
--    λ> primosEquidistantes7 4 !! 200
--    (9829,9833)
--    (0.02 secs, 3,729,816 bytes)
--
```

```
--    λ> primosEquidistantes2 4 !! 600
--    (41617,41621)
--    (5.67 secs, 3,340,313,480 bytes)
--    λ> primosEquidistantes3 4 !! 600
--    (41617,41621)
--    (5.43 secs, 3,090,994,096 bytes)
--    λ> primosEquidistantes4 4 !! 600
--    (41617,41621)
--    (0.03 secs, 15,465,824 bytes)
--    λ> primosEquidistantes5 4 !! 600
--    (41617,41621)
--    (0.04 secs, 28,858,232 bytes)
--    λ> primosEquidistantes6 4 !! 600
--    (41617,41621)
--    (0.04 secs, 13,449,344 bytes)
--    λ> primosEquidistantes7 4 !! 600
--    (41617,41621)
--    (0.04 secs, 13,811,992 bytes)
--
--    λ> primosEquidistantes4 4 !! (10^5)
--    (18467047,18467051)
--    (3.99 secs, 9,565,715,488 bytes)
--    λ> primosEquidistantes5 4 !! (10^5)
--    (18467047,18467051)
--    (7.95 secs, 18,712,469,144 bytes)
--    λ> primosEquidistantes6 4 !! (10^5)
--    (18467047,18467051)
--    (3.71 secs, 8,405,806,400 bytes)
--    λ> primosEquidistantes7 4 !! (10^5)
--    (18467047,18467051)
--    (3.72 secs, 8,502,545,368 bytes)
```

Ejercicio 9

Suma si todos los valores son justos

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    sumaSiTodosJustos :: (Num a, Eq a) => [Maybe a] -> Maybe a
-- tal que (sumaSiTodosJustos xs) es justo la suma de todos los
-- elementos de xs si todos son justos (es decir, si Nothing no
-- pertenece a xs) y Nothing en caso contrario. Por ejemplo,
--    sumaSiTodosJustos [Just 2, Just 5]           == Just 7
--    sumaSiTodosJustos [Just 2, Just 5, Nothing]  == Nothing
-- ---------------------------------------------------------------------

module Suma_si_todos_justos where

import Data.Maybe (catMaybes, isJust, fromJust)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

sumaSiTodosJustos1 :: (Num a, Eq a) => [Maybe a] -> Maybe a
sumaSiTodosJustos1 xs
  | todosJustos xs = Just (sum [x | (Just x) <- xs])
  | otherwise      = Nothing

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

```haskell
-- 1ª definición de todosJustos:
todosJustos1 :: Eq a => [Maybe a] -> Bool
todosJustos1 = notElem Nothing

-- 2ª definición de todosJustos:
todosJustos :: Eq a => [Maybe a] -> Bool
todosJustos = all isJust

-- 2ª solución
-- ===========

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

-- 3ª solución
-- ===========

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

-- 4ª solución

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

-- 5ª solución
-- ===========

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

-- Nota. En la solución anterior se usa la función
--    sequence :: Monad m => [m a] -> m [a]
-- tal que (sequence xs) es la mónada obtenida evaluando cada una de las
-- de xs de izquierda a derecha. Por ejemplo,
--    sequence [Just 2, Just 5]   ==  Just [2,5]
```

```haskell
--    sequence [Just 2, Nothing]  ==  Nothing
--    sequence [[2,4],[5,7]]      ==  [[2,5],[2,7],[4,5],[4,7]]
--    sequence [[2,4],[5,7],[6]]  ==  [[2,5,6],[2,7,6],[4,5,6],[4,7,6]]
--    sequence [[2,4],[5,7],[]]   ==  []

-- 6ª solución
-- ===========

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

-- 7ª solución
-- ===========

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Maybe Integer] -> Maybe Integer) -> Spec
specG sumaSiTodosJustos = do
  it "e1" $
    sumaSiTodosJustos [Just 2, Just 5]           `shouldBe` Just 7
  it "e2" $
    sumaSiTodosJustos [Just 2, Just 5, Nothing]  `shouldBe` Nothing

spec :: Spec
spec = do
  describe "def. 1" $ specG sumaSiTodosJustos1
  describe "def. 2" $ specG sumaSiTodosJustos2
  describe "def. 3" $ specG sumaSiTodosJustos3
  describe "def. 4" $ specG sumaSiTodosJustos4
  describe "def. 5" $ specG sumaSiTodosJustos5
  describe "def. 6" $ specG sumaSiTodosJustos6
  describe "def. 7" $ specG sumaSiTodosJustos7

-- La verificación es
--    λ> verifica
--    14 examples, 0 failures

-- Equivalencia de las definiciones
```

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

-- La propiedad es
prop_sumaSiTodosJustos :: [Maybe Integer] -> Bool
prop_sumaSiTodosJustos xs =
  all (== sumaSiTodosJustos1 xs)
      [sumaSiTodosJustos2 xs,
       sumaSiTodosJustos3 xs,
       sumaSiTodosJustos4 xs,
       sumaSiTodosJustos5 xs,
       sumaSiTodosJustos6 xs,
       sumaSiTodosJustos7 xs]

verifica_sumaSiTodosJustos :: IO ()
verifica_sumaSiTodosJustos =
  quickCheck prop_sumaSiTodosJustos

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

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

ejemplo1 :: Integer -> [Maybe Integer]
ejemplo1 n = map Just [1..n]

ejemplo2 :: Integer -> [Maybe Integer]
ejemplo2 n = map Just [1..n] ++ [Nothing]

-- La comparación es
--    λ> sumaSiTodosJustos1 (ejemplo1 10000000)
--    Just 50000005000000
--    (4.08 secs, 3,520,610,288 bytes)
--    λ> sumaSiTodosJustos2 (ejemplo1 10000000)
--    Just 50000005000000
--    (4.75 secs, 4,000,610,376 bytes)
--    λ> sumaSiTodosJustos3 (ejemplo1 10000000)
--    Just 50000005000000
--    (2.55 secs, 3,680,610,368 bytes)
--    λ> sumaSiTodosJustos4 (ejemplo1 10000000)
--    Just 50000005000000
--    (2.87 secs, 3,360,610,184 bytes)
--    λ> sumaSiTodosJustos5 (ejemplo1 10000000)
--    Just 50000005000000
```

```
--    (4.96 secs, 3,461,220,272 bytes)
--    λ> sumaSiTodosJustos6 (ejemplo1 10000000)
--    Just 50000005000000
--    (3.30 secs, 3,461,220,152 bytes)
--    λ> sumaSiTodosJustos7 (ejemplo1 10000000)
--    Just 50000005000000
--    (4.10 secs, 3,461,220,304 bytes)
--
--    λ> sumaSiTodosJustos1 (ejemplo2 10000000)
--    Nothing
--    (3.22 secs, 3,200,600,632 bytes)
--    λ> sumaSiTodosJustos2 (ejemplo2 10000000)
--    Nothing
--    (2.12 secs, 3,200,600,688 bytes)
--    λ> sumaSiTodosJustos3 (ejemplo2 10000000)
--    Nothing
--    (3.22 secs, 3,200,600,704 bytes)
--    λ> sumaSiTodosJustos4 (ejemplo2 10000000)
--    Nothing
--    (2.34 secs, 3,200,600,616 bytes)
--    λ> sumaSiTodosJustos5 (ejemplo2 10000000)
--    Nothing
--    (4.02 secs, 3,061,210,672 bytes)
--    λ> sumaSiTodosJustos6 (ejemplo2 10000000)
--    Nothing
--    (2.81 secs, 3,061,210,568 bytes)
--    λ> sumaSiTodosJustos7 (ejemplo2 10000000)
--    Nothing
--    (2.61 secs, 3,061,210,720 bytes)
```

Ejercicio 10

Matriz de Toeplitz

```haskell
-- ---------------------------------------------------------------------
-- Una [matriz de Toeplitz](https://tinyurl.com/zqzokbm) es una matriz
-- cuadrada que es constante a lo largo de las diagonales paralelas a la
-- diagonal principal. Por ejemplo,
--    |2 5 1 6|       |2 5 1 6|
--    |4 2 5 1|       |4 2 6 1|
--    |7 4 2 5|       |7 4 2 5|
--    |9 7 4 2|       |9 7 4 2|
-- la primera es una matriz de Toeplitz y la segunda no lo es.
--
-- Las anteriores matrices se pueden definir por
--    ej1, ej2 :: Array (Int,Int) Int
--    ej1 = listArray ((1,1),(4,4)) [2,5,1,6,4,2,5,1,7,4,2,5,9,7,4,2]
--    ej2 = listArray ((1,1),(4,4)) [2,5,1,6,4,2,6,1,7,4,2,5,9,7,4,2]
--
-- Definir la función
--    esToeplitz :: Eq a => Array (Int,Int) a -> Bool
-- tal que (esToeplitz p) se verifica si la matriz p es de Toeplitz. Por
-- ejemplo,
--    esToeplitz ej1  ==  True
--    esToeplitz ej2  ==  False
-- ---------------------------------------------------------------------

module Matriz_Toeplitz where

import Data.Array (Array, (!), array, bounds, listArray)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

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

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

esToeplitz1 :: Eq a => Array (Int,Int) a -> Bool
esToeplitz1 p =
  esCuadrada p &&
  all todosIguales (diagonalesPrincipales p)

-- (esCuadrada p) se verifica si la matriz p es cuadrada. Por ejemplo,
--    esCuadrada (listArray ((1,1),(4,4)) [1..])  ==  True
--    esCuadrada (listArray ((1,1),(3,4)) [1..])  ==  False
esCuadrada :: Eq a => Array (Int,Int) a -> Bool
esCuadrada p = m == n
  where (_,(m,n)) = bounds p

-- (diagonalesPrincipales p) es la lista de las diagonales principales
-- de p. Por ejemplo,
--    λ> diagonalesPrincipales ej1
--    [[2,2,2,2],[5,5,5],[1,1],[6],[2,2,2,2],[4,4,4],[7,7],[9]]
--    λ> diagonalesPrincipales ej2
--    [[2,2,2,2],[5,6,5],[1,1],[6],[2,2,2,2],[4,4,4],[7,7],[9]]
diagonalesPrincipales :: Array (Int,Int) a -> [[a]]
diagonalesPrincipales p =
  [[p ! i |i <- is] | is <- posicionesDiagonalesPrincipales m n]
  where (_,(m,n)) = bounds p

-- (posicionesDiagonalesPrincipales m n) es la lista de las
-- posiciones de las diagonales principales de una matriz con m filas y
-- n columnas. Por ejemplo,
--   λ> mapM_ print (posicionesDiagonalesPrincipales 3 4)
--   [(3,1)]
--   [(2,1),(3,2)]
--   [(1,1),(2,2),(3,3)]
--   [(1,2),(2,3),(3,4)]
--   [(1,3),(2,4)]
--   [(1,4)]
--   λ> mapM_ print (posicionesDiagonalesPrincipales 4 4)
--   [(4,1)]
--   [(3,1),(4,2)]
--   [(2,1),(3,2),(4,3)]
--   [(1,1),(2,2),(3,3),(4,4)]
--   [(1,2),(2,3),(3,4)]
--   [(1,3),(2,4)]
--   [(1,4)]
```

```
--   λ> mapM_ print (posicionesDiagonalesPrincipales 4 3)
--   [(4,1)]
--   [(3,1),(4,2)]
--   [(2,1),(3,2),(4,3)]
--   [(1,1),(2,2),(3,3)]
--   [(1,2),(2,3)]
--   [(1,3)]
posicionesDiagonalesPrincipales :: Int -> Int -> [[(Int, Int)]]
posicionesDiagonalesPrincipales m n =
  [zip [i..m] [1..n] | i <- [m,m-1..1]] ++
  [zip [1..m] [j..n] | j <- [2..n]]

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    todosIguales [5,5,5]  ==  True
--    todosIguales [5,4,5]  ==  False
todosIguales :: Eq a => [a] -> Bool
todosIguales []     = True
todosIguales (x:xs) = all (== x) xs

-- 2ª solución
-- ===========

esToeplitz2 :: Eq a => Array (Int,Int) a -> Bool
esToeplitz2 p = m == n &&
                and [p!(i,j) == p!(i+1,j+1) |
                     i <- [1..n-1], j <- [1..n-1]]
  where (_,(m,n)) = bounds p

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Array (Int,Int) Int -> Bool) -> Spec
specG esToeplitz = do
  it "e1" $
    esToeplitz ej1 `shouldBe` True
  it "e2" $
    esToeplitz ej2 `shouldBe` False

spec :: Spec
spec = do
  describe "def. 1" $ specG esToeplitz1
```

```
  describe "def. 2" $ specG esToeplitz2

-- La verificación es
--    λ> verifica
--    4 examples, 0 failures

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

newtype Matriz2 = M (Array (Int,Int) Int)
  deriving Show

-- Generador de matrices arbitrarias. Por ejemplo,
--    λ> generate matrizArbitraria
--    M (array ((1,1),(3,4))
--             [((1,1),18),((1,2),6), ((1,3),-23),((1,4),-13),
--              ((2,1),-2),((2,2),22),((2,3),-25),((2,4),-5),
--              ((3,1),2), ((3,2),16),((3,3),-15),((3,4),7)])
matrizArbitraria :: Gen Matriz2
matrizArbitraria = do
  n  <- chooseInt (1,5)
  xs <- vectorOf (n*n) arbitrary
  return (M (listArray ((1,1),(n,n)) xs))

-- Generador de matrices de Toeplitz arbitrarias. Por ejemplo,
--    λ> generate matrizToeplitzArbitraria
--    M (array ((1,1),(3,3)) [((1,1),-28),((1,2), 28),((1,3),  9),
--                            ((2,1),  6),((2,2),-28),((2,3), 28),
--                            ((3,1),-29),((3,2),  6),((3,3),-28)])
matrizToeplitzArbitraria :: Gen Matriz2
matrizToeplitzArbitraria = do
  n <- chooseInt (1, 5)
  primeraFila <- vectorOf n arbitrary
  primeraColumna <- vectorOf (n-1) arbitrary
  let xs = [((i,j), if i <= j
                    then primeraFila !! (j-i)
                    else primeraColumna !! (i-j-1))
            | i <- [1..n], j <- [1..n]]
  return (M (array ((1,1),(n,n)) xs))

-- Matriz es una subclase de Arbitrary.
instance Arbitrary Matriz2 where
  arbitrary = frequency
    [ (1, matrizToeplitzArbitraria)  -- 25% matrices de Toeplitz
    , (3, matrizArbitraria)          -- 75% matrices aleatorias
```

```
    ]

-- La propiedad es
prop_esToeplitz :: Matriz2 -> Bool
prop_esToeplitz (M p) =
  esToeplitz1 p == esToeplitz2 p

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

-- La propiedad para que indique el porcentaje de matrices de Toeplitz
-- generadas.
prop_esToeplitz2 :: Matriz2 -> Property
prop_esToeplitz2 (M p) =
  collect resultado1 $ resultado1 == esToeplitz2 p
  where resultado1 = esToeplitz1 p

-- La comprobación es
--    λ> quickCheck prop_esToeplitz2
--    +++ OK, passed 100 tests:
--    58% False
--    42% True

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

-- La comparación es
--    λ> esToeplitz1 (listArray ((1,1),(2*10^3,2*10^3)) (repeat 1))
--    True
--    (2.26 secs, 2,211,553,888 bytes)
--    λ> esToeplitz2 (listArray ((1,1),(2*10^3,2*10^3)) (repeat 1))
--    True
--    (4.26 secs, 3,421,651,032 bytes)
```

Ejercicio 11

Máximos locales

```
-- ---------------------------------------------------------------------
-- Un máximo local de una lista es un elemento de la lista que es mayor
-- que su predecesor y que su sucesor en la lista. Por ejemplo, 5 es un
-- máximo local de [3,2,5,3,7,7,1,6,2] ya que es mayor que 2 (su
-- predecesor) y que 3 (su sucesor).
--
-- Definir la función
--    maximosLocales :: Ord a => [a] -> [a]
-- tal que (maximosLocales xs) es la lista de los máximos locales de la
-- lista xs. Por ejemplo,
--    maximosLocales [3,2,5,3,7,7,1,6,2]  ==  [5,6]
--    maximosLocales [1..100]             ==  []
--    maximosLocales "adbpmqexyz"         ==  "dpq"
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Maximos_locales where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

maximosLocales1 :: Ord a => [a] -> [a]
maximosLocales1 (x:y:z:xs)
  | y > x && y > z = y : maximosLocales1 (z:xs)
  | otherwise      = maximosLocales1 (y:z:xs)
maximosLocales1 _ = []

-- 2ª solución
```

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

maximosLocales2 :: Ord a => [a] -> [a]
maximosLocales2 xs =
  [y | (x,y,z) <- zip3 xs (tail xs) (drop 2 xs), y > x, y > z]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [Int]) -> Spec
specG maximosLocales = do
  it "e1" $
    maximosLocales [3,2,5,3,7,7,1,6,2]  `shouldBe`  [5,6]
  it "e2" $
    maximosLocales [1..100]             `shouldBe`  []

spec :: Spec
spec = do
  describe "def. 1" $ specG maximosLocales1
  describe "def. 2" $ specG maximosLocales2

-- La verificación es
--    λ> verifica
--
--    4 examples, 0 failures

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

-- La propiedad es
prop_maximosLocales :: [Int] -> Property
prop_maximosLocales xs =
  maximosLocales1 xs === maximosLocales2 xs

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

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

-- La comparación es
```

```
--    λ> last (maximosLocales1 (take (6*10^6) (cycle "abc")))
--    'c'
--    (3.26 secs, 1,904,464,984 bytes)
--    λ> last (maximosLocales2 (take (6*10^6) (cycle "abc")))
--    'c'
--    (2.79 secs, 1,616,465,088 bytes)
```

Ejercicio 12

Lista cuadrada

```
-- ---------------------------------------------------------------------
-- Definir la función
--    listaCuadrada :: Int -> a -> [a] -> [[a]]
-- tal que (listaCuadrada n x xs) es una lista de n listas de longitud n
-- formadas con los elementos de xs completada con x, si no xs no tiene
-- suficientes elementos. Por ejemplo,
--    listaCuadrada 3 7 [0,3,5,2,4]  ==  [[0,3,5],[2,4,7],[7,7,7]]
--    listaCuadrada 3 7 [0..]        ==  [[0,1,2],[3,4,5],[6,7,8]]
--    listaCuadrada 2 'p' "eva"      ==  ["ev","ap"]
--    listaCuadrada 2 'p' ['a'..]    ==  ["ab","cd"]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Lista_cuadrada where

import Data.List.Split (chunksOf)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

listaCuadrada1 :: Int -> a -> [a] -> [[a]]
listaCuadrada1 n x xs =
  take n (grupos n (xs ++ repeat x))

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

```haskell
grupos :: Int -> [a] -> [[a]]
grupos _ [] = []
grupos n xs = take n xs : grupos n (drop n xs)

-- 2ª solución
-- ===========

listaCuadrada2 :: Int -> a -> [a] -> [[a]]
listaCuadrada2 n x xs =
  take n (grupos2 n (xs ++ repeat x))

grupos2 :: Int -> [a] -> [[a]]
grupos2 _ [] = []
grupos2 n xs = ys : grupos2 n zs
  where (ys,zs) = splitAt n xs

-- 3ª solución
-- ===========

listaCuadrada3 :: Int -> a -> [a] -> [[a]]
listaCuadrada3 n x xs =
  take n (chunksOf n (xs ++ repeat x))

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> Int -> [Int] -> [[Int]])  -> Spec
specG listaCuadrada = do
  it "e1" $
    listaCuadrada 3 7 [0,3,5,2,4]  `shouldBe`  [[0,3,5],[2,4,7],[7,7,7]]
  it "e2" $
    listaCuadrada 3 7 [0..]        `shouldBe`  [[0,1,2],[3,4,5],[6,7,8]]

spec :: Spec
spec = do
  describe "def. 1" $ specG listaCuadrada1
  describe "def. 2" $ specG listaCuadrada2
  describe "def. 3" $ specG listaCuadrada3

-- La verificación es
--    λ> verifica
--
```

```
--    6 examples, 0 failures

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

-- La propiedad es
prop_listaCuadrada :: Int -> Int -> [Int] -> Bool
prop_listaCuadrada n x xs =
  all (== listaCuadrada1 n x xs)
      [listaCuadrada2 n x xs,
       listaCuadrada3 n x xs]

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

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

-- La comparación es
--    λ> length (listaCuadrada1 (10^4) 5 [1..])
--    10000
--    (2.02 secs, 12,801,918,616 bytes)
--    λ> length (listaCuadrada2 (10^4) 5 [1..])
--    10000
--    (5.47 secs, 25,600,440,048 bytes)
--    λ> length (listaCuadrada3 (10^4) 5 [1..])
--    10000
--    (2.05 secs, 12,801,518,728 bytes)
```

Ejercicio 13

Segmentos maximales con elementos consecutivos

```
-- ----------------------------------------------------------------------
-- Definir la función
--    segmentos :: (Enum a, Eq a) => [a] -> [[a]]
-- tal que (segmentos xss) es la lista de los segmentos de xss formados
-- por elementos consecutivos. Por ejemplo,
--    segmentos [1,2,5,6,4]     ==  [[1,2],[5,6],[4]]
--    segmentos [1,2,3,4,7,8,9] ==  [[1,2,3,4],[7,8,9]]
--    segmentos "abbccddeeebc"  ==  ["ab","bc","cd","de","e","e","bc"]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
{-# OPTIONS_GHC -fno-warn-incomplete-uni-patterns #-}

module Segmentos_consecutivos where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

segmentos1 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos1 []  = []
segmentos1 xs = ys : segmentos1 zs
  where ys = inicial xs
        n  = length ys
        zs = drop n xs

-- (inicial xs) es el segmento inicial de xs formado por elementos
```

```haskell
-- consecutivos. Por ejemplo,
--    inicial [1,2,5,6,4]    ==  [1,2]
--    inicial "abccddeeebc"  ==  "abc"
inicial :: (Enum a, Eq a) => [a] -> [a]
inicial []      = []
inicial [x]     = [x]
inicial (x:y:xs)
  | succ x == y = x : inicial (y:xs)
  | otherwise   = [x]

-- 2ª solución
-- ===========

segmentos2 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos2 []  = []
segmentos2 xs = ys : segmentos2 zs
  where (ys,zs) = inicialYresto xs

-- (inicialYresto xs) es par formado por el segmento inicial de xs
-- con elementos consecutivos junto con los restantes elementos. Por
-- ejemplo,
--    inicialYresto [1,2,5,6,4]    ==  ([1,2],[5,6,4])
--    inicialYresto "abccddeeebc"  ==  ("abc","cddeeebc")
inicialYresto :: (Enum a, Eq a) => [a] -> ([a],[a])
inicialYresto []      = ([],[])
inicialYresto [x]     = ([x],[])
inicialYresto (x:y:xs)
  | succ x == y = (x:us,vs)
  | otherwise   = ([x],y:xs)
  where (us,vs) = inicialYresto (y:xs)

-- 3ª solución
-- ===========

segmentos3 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos3 []  = []
segmentos3 [x] = [[x]]
segmentos3 (x:xs) | y == succ x = (x:y:ys):zs
                  | otherwise   = [x] : (y:ys):zs
  where ((y:ys):zs) = segmentos3 xs

-- 4ª solución
-- ===========

segmentos4 :: (Enum a, Eq a) => [a] -> [[a]]
```

```
segmentos4 []  = []
segmentos4 xs = ys : segmentos4 zs
  where ys = inicial4 xs
        n  = length ys
        zs = drop n xs

inicial4 :: (Enum a, Eq a) => [a] -> [a]
inicial4 [] = []
inicial4 (x:xs) =
  map fst (takeWhile (\(u,v) -> u == v) (zip (x:xs) [x..]))

-- 5ª solución
-- ===========

segmentos5 :: (Enum a, Eq a) => [a] -> [[a]]
segmentos5 []  = []
segmentos5 xs = ys : segmentos5 zs
  where ys = inicial5 xs
        n  = length ys
        zs = drop n xs

inicial5 :: (Enum a, Eq a) => [a] -> [a]
inicial5 [] = []
inicial5 (x:xs) =
  map fst (takeWhile (uncurry (==)) (zip (x:xs) [x..]))

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [[Int]])  -> Spec
specG segmentos = do
  it "e1" $
    segmentos [1,2,5,6,4] `shouldBe` [[1,2],[5,6],[4]]
  it "e2" $
    segmentos [1,2,3,4,7,8,9] `shouldBe` [[1,2,3,4],[7,8,9]]

spec :: Spec
spec = do
  describe "def. 1" $ specG segmentos1
  describe "def. 2" $ specG segmentos2
  describe "def. 3" $ specG segmentos3
  describe "def. 4" $ specG segmentos4
```

```
  describe "def. 5" $ specG segmentos5

-- La verificación es
--    λ> verifica
--
--    10 examples, 0 failures

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

-- La propiedad es
prop_segmentos :: [Int] -> Bool
prop_segmentos xs =
  all (== segmentos1 xs)
      [segmentos2 xs,
       segmentos3 xs,
       segmentos4 xs,
       segmentos5 xs]

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

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

-- La comparación es
--    λ> length (segmentos1 (take (10^6) (cycle [1..10^3])))
--    1000
--    (0.69 secs, 416,742,208 bytes)
--    λ> length (segmentos2 (take (10^6) (cycle [1..10^3])))
--    1000
--    (0.66 secs, 528,861,976 bytes)
--    λ> length (segmentos3 (take (10^6) (cycle [1..10^3])))
--    1000
--    (2.35 secs, 1,016,276,896 bytes)
--    λ> length (segmentos4 (take (10^6) (cycle [1..10^3])))
--    1000
--    (0.27 secs, 409,438,368 bytes)
--    λ> length (segmentos5 (take (10^6) (cycle [1..10^3])))
--    1000
--    (0.13 secs, 401,510,360 bytes)
--
--    λ> length (segmentos4 (take (10^7) (cycle [1..10^3])))
--    10000
```

```
--    (2.35 secs, 4,088,926,920 bytes)
--    λ> length (segmentos5 (take (10^7) (cycle [1..10^3])))
--    10000
--    (1.02 secs, 4,009,646,928 bytes)
```

Ejercicio 14

Valores de polinomios representados con vectores

```
-- ---------------------------------------------------------------------
-- Los polinomios se pueden representar mediante vectores usando la
-- librería Data.Array. En primer lugar, se define el tipo de los
-- polinomios (con coeficientes de tipo a) mediante
--    type Polinomio a = Array Int a
-- Como ejemplos, definimos el polinomio
--    ej_pol1 :: Array Int Int
--    ej_pol1 = array (0,4) [(0,6),(1,2),(2,-5),(3,0),(4,7)]
-- que representa a 6 + 2x - 5x^2 + 7x^4 y el polinomio
--    ej_pol2 :: Array Int Double
--    ej_pol2 = array (0,4) [(0,6.5),(1,2),(2,-5.2),(3,0),(4,7)]
-- que representa a 6.5 + 2x - 5.2x^2 + 7x^4
--
-- Definir la función
--    valor :: Num a => Polinomio a -> a -> a
-- tal que (valor p b) es el valor del polinomio p en el punto b. Por
-- ejemplo,
--    valor ej_pol1 0  ==  6
--    valor ej_pol1 1  ==  10
--    valor ej_pol1 2  ==  102
--    valor ej_pol2 0  ==  6.5
--    valor ej_pol2 1  ==  10.3
--    valor ej_pol2 3  ==  532.7
--    length (show (valor (listArray (0,5*10^5) (repeat 1)) 2)) == 150516
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Valor_de_un_polinomio where
```

```haskell
import Data.List (foldl')
import Data.Array (Array, (!), array, assocs, bounds, elems, listArray)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

type Polinomio a = Array Int a

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

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

-- 1ª solución
-- ===========

valor1 :: Num a => Polinomio a -> a -> a
valor1 p b = sum [(p!i)*b^i | i <- [0..n]]
  where (_,n) = bounds p

-- 2ª solución
-- ===========

valor2 :: Num a => Polinomio a -> a -> a
valor2 p b = sum [(p!i)*b^i | i <- [0..length p - 1]]

-- 3ª solución
-- ===========

valor3 :: Num a => Polinomio a -> a -> a
valor3 p b = sum [v*b^i | (i,v) <- assocs p]

-- 4ª solución
-- ===========

valor4 :: Num a => Polinomio a -> a -> a
valor4 = valorLista4 . elems

valorLista4 :: Num a => [a] -> a -> a
valorLista4 xs b =
  sum [(xs !! i) * b^i | i <- [0..length xs - 1]]

-- 5ª solución
-- ===========
```

```haskell
valor5 :: Num a => Polinomio a -> a -> a
valor5 = valorLista5 . elems

valorLista5 :: Num a => [a] -> a -> a
valorLista5 []     _ = 0
valorLista5 (x:xs) b = x + b * valorLista5 xs b

-- 6ª solución
-- ===========

valor6 :: Num a => Polinomio a -> a -> a
valor6 = valorLista6 . elems

valorLista6 :: Num a => [a] -> a -> a
valorLista6 xs b = aux xs
  where aux []     = 0
        aux (y:ys) = y + b * aux ys

-- 7ª solución
-- ===========

valor7 :: Num a => Polinomio a -> a -> a
valor7 = valorLista7 . elems

valorLista7 :: Num a => [a] -> a -> a
valorLista7 xs b = foldr (\y r -> y + b * r) 0 xs

-- 8ª solución
-- ===========

valor8 :: Num a => Polinomio a -> a -> a
valor8 = valorLista8 . elems

valorLista8 :: Num a => [a] -> a -> a
valorLista8 xs b = aux 0 (reverse xs)
  where aux r []     = r
        aux r (y:ys) = aux (y + r * b) ys

-- 9ª solución
-- ===========

valor9 :: Num a => Polinomio a -> a -> a
valor9 = valorLista9 . elems
```

```haskell
valorLista9 :: Num a => [a] -> a -> a
valorLista9 xs b = aux 0 (reverse xs)
  where aux = foldl (\ r y -> y + r * b)

-- 10ª solución
-- ============

valor10 :: Num a => Polinomio a -> a -> a
valor10 p b =
  foldl (\ r y -> y + r * b) 0 (reverse (elems p))

-- 11ª solución
-- ============

valor11 :: Num a => Polinomio a -> a -> a
valor11 p b =
  foldl' (\ r y -> y + r * b) 0 (reverse (elems p))

-- 12ª solución
-- ============

valor12 :: Num a => Polinomio a -> a -> a
valor12 p b =
  sum (zipWith (*) (elems p) (iterate (* b) 1))

-- 13ª solución
-- ============

valor13 :: Num a => Polinomio a -> a -> a
valor13 p b =
  foldl' (+) 0 (zipWith (*) (elems p) (iterate (* b) 1))

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Polinomio Int -> Int -> Int) -> Spec
specG valor = do
  it "e1" $
    valor ej_pol1 0  `shouldBe`  6
  it "e2" $
    valor ej_pol1 1  `shouldBe`  10
  it "e3" $
```

```
    valor ej_pol1 2  `shouldBe`  102

spec :: Spec
spec = do
  describe "def. 1"  $ specG valor1
  describe "def. 2"  $ specG valor2
  describe "def. 3"  $ specG valor3
  describe "def. 4"  $ specG valor4
  describe "def. 5"  $ specG valor5
  describe "def. 6"  $ specG valor6
  describe "def. 7"  $ specG valor7
  describe "def. 8"  $ specG valor8
  describe "def. 9"  $ specG valor9
  describe "def. 10" $ specG valor10
  describe "def. 11" $ specG valor11
  describe "def. 12" $ specG valor12
  describe "def. 13" $ specG valor13

-- La verificación es
--    λ> verifica
--
--    39 examples, 0 failures

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

-- La propiedad es
prop_valor :: [Integer] -> Integer -> Bool
prop_valor xs b =
  all (== valor1 p b)
      [f p b | f <- [valor2,
                     valor3,
                     valor4,
                     valor5,
                     valor6,
                     valor7,
                     valor8,
                     valor9,
                     valor10,
                     valor11,
                     valor12,
                     valor13]]
  where p = listArray (0, length xs - 1) xs

-- La comprobación es
```

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

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

-- La comparación es
--    λ> length (show (valor1 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (7.62 secs, 2,953,933,864 bytes)
--    λ> length (show (valor2 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (8.26 secs, 2,953,933,264 bytes)
--    λ> length (show (valor3 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (7.49 secs, 2,954,733,184 bytes)
--    λ> length (show (valor4 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (84.80 secs, 2,956,333,712 bytes)
--    λ> length (show (valor5 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (1.34 secs, 1,307,347,416 bytes)
--    λ> length (show (valor6 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (1.26 secs, 1,308,114,752 bytes)
--    λ> length (show (valor7 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (1.21 secs, 1,296,843,456 bytes)
--    λ> length (show (valor8 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (1.28 secs, 1,309,591,744 bytes)
--    λ> length (show (valor9 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (1.27 secs, 1,299,191,672 bytes)
--    λ> length (show (valor10 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (1.30 secs, 1,299,191,432 bytes)
--    λ> length (show (valor11 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (0.23 secs, 1,287,654,752 bytes)
--    λ> length (show (valor12 (listArray (0,10^5) (repeat 1)) 2))
--    30104
--    (0.75 secs, 1,309,506,968 bytes)
--    λ> length (show (valor13 (listArray (0,10^5) (repeat 1)) 2))
--    30104
```

```
--    (0.22 secs, 1,298,867,128 bytes)
```

Ejercicio 15

Ramas de un árbol

```haskell
-- ----------------------------------------------------------------------
-- Los árboles se pueden representar mediante el siguiente tipo de datos
--    data Arbol a = N a [Arbol a]
--      deriving Show
-- Por ejemplo, los árboles
--      1              3
--     / \            /|\
--    2   3          / | \
--        |         5  4  7
--        4         |     /\
--                  6    2  1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]
--
-- Definir la función
--    ramas :: Arbol b -> [[b]]
-- tal que (ramas a) es la lista de las ramas del árbol a. Por ejemplo,
--    ramas ej1  ==  [[1,2],[1,3,4]]
--    ramas ej2  ==  [[3,5,6],[3,4],[3,7,2],[3,7,1]]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Ramas_de_un_arbol where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

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

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

-- 1ª solución
-- ===========

ramas1 :: Arbol b -> [[b]]
ramas1 (N x []) = [[x]]
ramas1 (N x as) = [x : xs | a <- as, xs <- ramas1 a]

-- 2ª solución
-- ===========

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

-- 3ª solución
-- ===========

ramas3 :: Arbol b -> [[b]]
ramas3 (N x []) = [[x]]
ramas3 (N x as) = concat (map (map (x:) . ramas3) as)

-- 4ª solución
-- ===========

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

-- 5ª solución
-- ===========

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec
```

```haskell
specG :: (Arbol Int -> [[Int]]) -> Spec
specG ramas = do
  it "e1" $
    ramas ej1  `shouldBe`  [[1,2],[1,3,4]]
  it "e2" $
    ramas ej2  `shouldBe`  [[3,5,6],[3,4],[3,7,2],[3,7,1]]

spec :: Spec
spec = do
  describe "def. 1" $ specG ramas1
  describe "def. 2" $ specG ramas2
  describe "def. 3" $ specG ramas3
  describe "def. 4" $ specG ramas4
  describe "def. 5" $ specG ramas5

-- La verificación es
--    λ> verifica
--    10 examples, 0 failures

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

-- (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
--    λ> sample (arbolArbitrario 4 :: Gen (Arbol Int))
--    N 0 [N 0 []]
--    N 1 [N 1 [N (-2) [N (-1) [N (-1) [N (-1) [N 1 []]]]]],N (-1) [N 2 []]]
--    N 1 [N (-2) [],N 0 [N (-4) [N (-2) []]]]
--    N (-4) [N 1 [],N 0 [N 6 [N (-4) []],N 2 [N 3 []]]]
--    N (-7) [N (-7) [N (-3) []]]
--    N (-2) [N (-8) []]
--    N (-3) [N 3 [N 2 []]]
--    N (-12) [N 5 [],N 0 []]
--    N 14 [N 13 [N (-12) []],N 11 [],N 8 [N (-13) []]]
--    N (-12) [N (-6) [N 16 [N (-14) [N (-1) []]]]]
--    N (-5) []
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario n = do
  x  <- arbitrary
  ms <- sublistOf [0 .. n `div` 2]
  as <- mapM arbolArbitrario ms
  return (N x as)

-- Arbol es una subclase de Arbitraria
instance Arbitrary a => Arbitrary (Arbol a) where
```

```
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_arbol :: Arbol Int -> Bool
prop_arbol a =
  all (== ramas1 a)
      [ramas2 a,
       ramas3 a,
       ramas4 a,
       ramas5 a]

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

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

-- La comparación es
--    λ> ej600 <- generate (arbolArbitrario 600 :: Gen (Arbol Int))
--    λ> length (ramas1 ej600)
--    1262732
--    (1.92 secs, 1,700,238,488 bytes)
--    λ> length (ramas2 ej600)
--    1262732
--    (1.94 secs, 2,549,877,280 bytes)
--    λ> length (ramas3 ej600)
--    1262732
--    (1.99 secs, 2,446,508,472 bytes)
--    λ> length (ramas4 ej600)
--    1262732
--    (1.67 secs, 2,090,469,104 bytes)
--    λ> length (ramas5 ej600)
--    1262732
--    (1.66 secs, 2,112,198,232 bytes)
```

Ejercicio 16

Alfabeto comenzando en un carácter

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    alfabetoDesde :: Char -> String
-- tal que (alfabetoDesde c) es el alfabeto, en minúscula, comenzando en
-- el carácter c, si c es una letra minúscula y comenzando en 'a', en
-- caso contrario. Por ejemplo,
--    alfabetoDesde 'e'  ==  "efghijklmnopqrstuvwxyzabcd"
--    alfabetoDesde 'a'  ==  "abcdefghijklmnopqrstuvwxyz"
--    alfabetoDesde '7'  ==  "abcdefghijklmnopqrstuvwxyz"
--    alfabetoDesde '{'  ==  "abcdefghijklmnopqrstuvwxyz"
--    alfabetoDesde 'B'  ==  "abcdefghijklmnopqrstuvwxyz"
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Alfabeto_desde where

import Data.Char (isLower, isAscii)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

alfabetoDesde1 :: Char -> String
alfabetoDesde1 c =
  dropWhile (<c) ['a'..'z'] ++ takeWhile (<c) ['a'..'z']

-- 2ª solución
```

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

alfabetoDesde2 :: Char -> String
alfabetoDesde2 c = ys ++ xs
  where (xs,ys) = span (<c) ['a'..'z']

-- 3ª solución
-- ===========

alfabetoDesde3 :: Char -> String
alfabetoDesde3 c = ys ++ xs
  where (xs,ys) = break (==c) ['a'..'z']

-- 4ª solución
-- ===========

alfabetoDesde4 :: Char -> String
alfabetoDesde4 c
  | 'a' <= c && c <= 'z' = [c..'z'] ++ ['a'..pred c]
  | otherwise            = ['a'..'z']

-- 5ª solución
-- ===========

alfabetoDesde5 :: Char -> String
alfabetoDesde5 c
  | isLower c = [c..'z'] ++ ['a'..pred c]
  | otherwise = ['a'..'z']

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Char -> String) -> Spec
specG alfabetoDesde = do
  it "e1" $
      alfabetoDesde 'e'  `shouldBe`  "efghijklmnopqrstuvwxyzabcd"
  it "e2" $
      alfabetoDesde 'a'  `shouldBe`  "abcdefghijklmnopqrstuvwxyz"
  it "e3" $
      alfabetoDesde '7'  `shouldBe`  "abcdefghijklmnopqrstuvwxyz"
  it "e4" $
      alfabetoDesde '{'  `shouldBe`  "abcdefghijklmnopqrstuvwxyz"
```

```
  it "e5" $
      alfabetoDesde 'B'  `shouldBe`  "abcdefghijklmnopqrstuvwxyz"
  it "p1" $
    property prop_alfabetoDesde

spec :: Spec
spec = do
  describe "def. 1" $ specG alfabetoDesde1
  describe "def. 2" $ specG alfabetoDesde2
  describe "def. 3" $ specG alfabetoDesde3
  describe "def. 4" $ specG alfabetoDesde4
  describe "def. 5" $ specG alfabetoDesde5

-- La verificación es
--    λ> verifica
--    30 examples, 0 failures

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

-- La propiedad es
prop_alfabetoDesde :: Property
prop_alfabetoDesde =
  forAll (arbitrary `suchThat` isAscii) $ \c ->
  all (== alfabetoDesde1 c)
      [f c | f <- [alfabetoDesde2,
                   alfabetoDesde3,
                   alfabetoDesde4,
                   alfabetoDesde5]]


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

Ejercicio 17

Numeración de ternas

```haskell
-- ----------------------------------------------------------------------
-- 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(x²+6x+11)/6
-- + la posición de (0,y,0) es y(y²+3y+ 8)/6
-- + la posición de (0,0,z) es z(z²+3z+ 2)/6
-- + la posición de (x,x,x) es x(9x²+14x+7)/2
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}
{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Numeracion_de_ternas where

import Data.List (elemIndex)
import Data.Maybe (fromJust)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck
```

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

posicion1 :: (Int,Int,Int) -> Int
posicion1 t = aux 0 ternas
  where aux n (t':ts) | t' == t   = n
                      | otherwise = aux (n+1) ts

-- ternas es la lista ordenada de las ternas de números naturales. Por ejemplo,
--    λ> 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 =
  head [n | (n,t') <- zip [0..] ternas, t' == t]

-- 3ª solución
-- ===========

posicion3 :: (Int,Int,Int) -> Int
posicion3 t = indice t ternas

-- (indice x ys) es el índice de x en ys. Por ejemplo,
--    indice 5 [0..]  ==  5
indice :: Eq a => a -> [a] -> Int
indice x ys = length (takeWhile (/= x) ys)

-- 4ª solución
-- ===========

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

-- 5ª solución
-- ===========

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

```haskell
-- 6ª solución
-- ===========

posicion6 :: (Int,Int,Int) -> Int
posicion6 (x,y,z) =
  sum [((n+2)*(n+1)) `div` 2 | n <- [0..s-1]] +
  sum [s-i+1 | i <- [0..x-1]] + y
  where s = x + y + z

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ((Int,Int,Int) -> Int) -> Spec
specG posicion = do
  it "e1" $
    posicion (0,1,0)  `shouldBe`  2
  it "e2" $
    posicion (0,0,2)  `shouldBe`  4
  it "e3" $
    posicion (0,1,1)  `shouldBe`  5

spec :: Spec
spec = do
  describe "def. 1" $ specG posicion1
  describe "def. 2" $ specG posicion2
  describe "def. 3" $ specG posicion3
  describe "def. 4" $ specG posicion4
  describe "def. 5" $ specG posicion5
  describe "def. 6" $ specG posicion6

-- La verificación es
--    λ> verifica
--    15 examples, 0 failures

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

-- La propiedad es
prop_posicion_equiv :: NonNegative Int
                    -> NonNegative Int
                    -> NonNegative Int
                    -> Bool
```

```
prop_posicion_equiv (NonNegative x) (NonNegative y) (NonNegative z) =
  all (== posicion1 (x,y,z))
      [f (x,y,z) | f <- [ posicion2
                        , posicion3
                        , posicion4
                        , posicion5
                        , posicion6 ]]

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

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

-- La comparación es
--    λ> posicion1 (147,46,116)
--    5000000
--    (5.84 secs, 2,621,428,184 bytes)
--    λ> posicion2 (147,46,116)
--    5000000
--    (3.63 secs, 2,173,230,200 bytes)
--    λ> posicion3 (147,46,116)
--    5000000
--    (2.48 secs, 1,453,229,880 bytes)
--    λ> posicion4 (147,46,116)
--    5000000
--    (1.91 secs, 1,173,229,840 bytes)
--    λ> posicion5 (147,46,116)
--    5000000
--    (1.94 secs, 1,173,229,960 bytes)
--    λ> posicion6 (147,46,116)
--    5000000
--    (0.01 secs, 779,568 bytes)

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

-- La 1ª propiedad es
prop_posicion1 :: NonNegative Int -> Bool
prop_posicion1 (NonNegative x) =
  posicion5 (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) =
  posicion5 (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.

-- La 3ª propiedad es
prop_posicion3 :: NonNegative Int -> Bool
prop_posicion3 (NonNegative z) =
  posicion5 (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) =
  posicion5 (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 18

Ordenación de estructuras

```
-- ---------------------------------------------------------------------
-- Las notas de los dos primeros exámenes se pueden representar mediante
-- el siguiente tipo de dato
--    data Notas = Notas String Int Int
--      deriving (Read, Show, Eq)
-- Por ejemplo, (Notas "Juan" 6 5) representar las notas de un alumno
-- cuyo nombre es Juan, la nota del primer examen es 6 y la del segundo
-- es 5.
--
-- Definir la función
--    ordenadas :: [Notas] -> [Notas]
-- tal que (ordenadas ns) es la lista de las notas ns ordenadas
-- considerando primero la nota del examen 2, a continuación la del
-- examen 1 y finalmente el nombre. Por ejemplo,
--    λ> ordenadas [Notas "Juan" 6 5, Notas "Luis" 3 7]
--    [Notas "Juan" 6 5,Notas "Luis" 3 7]
--    λ> ordenadas [Notas "Juan" 6 5, Notas "Luis" 3 4]
--    [Notas "Luis" 3 4,Notas "Juan" 6 5]
--    λ> ordenadas [Notas "Juan" 6 5, Notas "Luis" 7 4]
--    [Notas "Luis" 7 4,Notas "Juan" 6 5]
--    λ> ordenadas [Notas "Juan" 6 4, Notas "Luis" 7 4]
--    [Notas "Juan" 6 4,Notas "Luis" 7 4]
--    λ> ordenadas [Notas "Juan" 6 4, Notas "Luis" 5 4]
--    [Notas "Luis" 5 4,Notas "Juan" 6 4]
--    λ> ordenadas [Notas "Juan" 5 4, Notas "Luis" 5 4]
--    [Notas "Juan" 5 4,Notas "Luis" 5 4]
--    λ> ordenadas [Notas "Juan" 5 4, Notas "Eva" 5 4]
--    [Notas "Eva" 5 4,Notas "Juan" 5 4]
-- ---------------------------------------------------------------------

module Ordenacion_de_estructuras where
```

```haskell
import Data.List (sort, sortBy)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

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

-- 1ª solución
-- ===========

ordenadas1 :: [Notas] -> [Notas]
ordenadas1 ns =
  [Notas n x y | (y,x,n) <- sort [(y1,x1,n1) | (Notas n1 x1 y1) <- ns]]

-- 2ª solución
-- ===========

ordenadas2 :: [Notas] -> [Notas]
ordenadas2 ns =
  map (\(y,x,n) -> Notas n x y) (sort [(y1,x1,n1) | (Notas n1 x1 y1) <- ns])

-- 3ª solución
-- ===========

ordenadas3 :: [Notas] -> [Notas]
ordenadas3 ns = sortBy (\(Notas n1 x1 y1) (Notas n2 x2 y2) ->
                          compare (y1,x1,n1) (y2,x2,n2))
                       ns

-- 4ª solución
-- ===========

instance Ord Notas where
  Notas n1 x1 y1 <= Notas n2 x2 y2 = (y1,x1,n1) <= (y2,x2,n2)

ordenadas4 :: [Notas] -> [Notas]
ordenadas4 = sort

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Notas] -> [Notas]) -> Spec
```

```
specG ordenadas = do
  it "e1" $
    ordenadas [Notas "Juan" 6 5, Notas "Luis" 3 7]
    `shouldBe` [Notas "Juan" 6 5,Notas "Luis" 3 7]
  it "e2" $
    ordenadas [Notas "Juan" 6 5, Notas "Luis" 3 4]
    `shouldBe` [Notas "Luis" 3 4,Notas "Juan" 6 5]
  it "e3" $
    ordenadas [Notas "Juan" 6 5, Notas "Luis" 7 4]
    `shouldBe` [Notas "Luis" 7 4,Notas "Juan" 6 5]
  it "e4" $
    ordenadas [Notas "Juan" 6 4, Notas "Luis" 7 4]
    `shouldBe` [Notas "Juan" 6 4,Notas "Luis" 7 4]
  it "e5" $
    ordenadas [Notas "Juan" 6 4, Notas "Luis" 5 4]
    `shouldBe` [Notas "Luis" 5 4,Notas "Juan" 6 4]
  it "e6" $
    ordenadas [Notas "Juan" 5 4, Notas "Luis" 5 4]
    `shouldBe` [Notas "Juan" 5 4,Notas "Luis" 5 4]
  it "e7" $
    ordenadas [Notas "Juan" 5 4, Notas "Eva" 5 4]
    `shouldBe` [Notas "Eva" 5 4,Notas "Juan" 5 4]

spec :: Spec
spec = do
  describe "def. 1" $ specG ordenadas1
  describe "def. 2" $ specG ordenadas2
  describe "def. 3" $ specG ordenadas3
  describe "def. 4" $ specG ordenadas4

-- La verificación es
--    λ> verifica
--    28 examples, 0 failures

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

-- notasArbitraria es un generador aleatorio de notas. Por ejemplo,
--    λ> sample notasArbitraria
--    Notas "achjkqruvxy" 3 3
--    Notas "abfgikmptuvy" 10 10
--    Notas "degjmptvwx" 7 9
--    Notas "cdefghjmnoqrsuw" 0 9
--    Notas "bcdfikmstuxz" 1 8
--    Notas "abcdhkopqsvwx" 10 7
```

```
--    Notas "abghiklnoqstvwx" 0 0
--    Notas "abfghklmnoptuvx" 4 9
--    Notas "bdehjkmpqsxyz" 0 4
--    Notas "afghijmopsvwz" 3 7
--    Notas "bdefghjklnoqx" 2 3
notasArbitraria :: Gen Notas
notasArbitraria = do
  n <- sublistOf ['a'..'z']
  x <- chooseInt (0, 10)
  y <- chooseInt (0, 10)
  return (Notas n x y)

-- Notas es una subclase de Arbitrary
instance Arbitrary Notas where
  arbitrary = notasArbitraria

-- La propiedad es
prop_ordenadas :: [Notas] -> Bool
prop_ordenadas ns =
  all (== ordenadas1 ns)
      [f ns | f <- [ordenadas2,
                    ordenadas3,
                    ordenadas4]]

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

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

-- La comparación es
--    λ> ejemplo <- generate (vectorOf 320000 notasArbitraria)
--    λ> length (ordenadas1 ejemplo)
--    320000
--    (2.62 secs, 1,062,514,344 bytes)
--    λ> length (ordenadas2 ejemplo)
--    320000
--    (1.60 secs, 489,138,472 bytes)
--    λ> length (ordenadas3 ejemplo)
--    320000
--    (2.10 secs, 1,271,107,760 bytes)
--    λ> length (ordenadas4 ejemplo)
--    320000
--    (4.77 secs, 2,020,931,872 bytes)
```

Ejercicio 19

Emparejamiento binario

```
-- ----------------------------------------------------------------------
-- Definir la función
--    zipBinario :: [a -> b -> c] -> [a] -> [b] -> [c]
-- tal que (zipBinario fs xs ys) es la lista obtenida aplicando cada una
-- de las operaciones binarias de fs a los correspondientes elementos de
-- xs e ys. Por ejemplo,
--    zipBinario [(+), (*), (*)] [2,2,2] [4,4,4]    == [6,8,8]
--    zipBinario [(+)] [2,2,2] [4,4,4]              == [6]
--    zipBinario [(<), (==), (==)] "coloca" "lobo"  == [True,True,False]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Emparejamiento_binario where

import Test.QuickCheck
import Test.QuickCheck.HigherOrder
import Test.Hspec

-- 1ª solución
-- ===========

zipBinario1 :: [a -> b -> c] -> [a] -> [b] -> [c]
zipBinario1 (f:fs) (x:xs) (y:ys) = f x y : zipBinario1 fs xs ys
zipBinario1 _ _ _                = []

-- 2ª solución
-- ===========

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

```haskell
-- 3ª solución
-- ===========

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

-- 4ª solución
-- ===========

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int -> Int -> Int] -> [Int] -> [Int] -> [Int]) -> Spec
specG zipBinario = do
  it "e1" $ zipBinario [(+), (*), (*)] [2,2,2] [4,4,4]    `shouldBe` [6,8,8]
  it "e2" $ zipBinario [(+)] [2,2,2] [4,4,4]              `shouldBe` [6]

spec :: Spec
spec = do
  describe "def. 1" $ specG zipBinario1
  describe "def. 2" $ specG zipBinario2
  describe "def. 3" $ specG zipBinario3
  describe "def. 4" $ specG zipBinario4

-- La verificación es
--    λ> verifica
--    8 examples, 0 failures

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

-- La propiedad es
prop_zipBinario :: [Int -> Int -> Int] -> [Int] -> [Int] -> Bool
prop_zipBinario fs xs ys =
  all (== zipBinario1 fs xs ys)
      [g fs xs ys | g <- [zipBinario2,
                          zipBinario3,
                          zipBinario4]]
```

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

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

-- La comparación es
--    λ> maximum (zipBinario1 (cycle [(+), (*)]) [1..] [1..2*10^6])
--    4000000000000
--    (2.13 secs, 965,392,072 bytes)
--    λ> maximum (zipBinario2 (cycle [(+), (*)]) [1..] [1..2*10^6])
--    4000000000000
--    (1.86 secs, 1,109,392,176 bytes)
--    λ> maximum (zipBinario3 (cycle [(+), (*)]) [1..] [1..2*10^6])
--    4000000000000
--    (1.93 secs, 981,392,128 bytes)
--    λ> maximum (zipBinario4 (cycle [(+), (*)]) [1..] [1..2*10^6])
--    4000000000000
--    (1.07 secs, 773,392,040 bytes)
```

Ejercicio 20

Amplia columnas

```
-- ---------------------------------------------------------------------
-- Las matrices enteras se pueden representar mediante tablas con
-- índices enteros:
--    type Matriz = Array (Int,Int) Int
--
-- Definir la función
--    ampliaColumnas :: Matriz -> Matriz -> Matriz
-- tal que (ampliaColumnas p q) es la matriz construida añadiendo las
-- columnas de la matriz q a continuación de las de p (se supone que
-- tienen el mismo número de filas). Por ejemplo, si p y q representa
-- las dos primeras matrices, entonces (ampliaColumnas p q) es la
-- tercera
--    |0 1|    |4 5 6|    |0 1 4 5 6|
--    |2 3|    |7 8 9|    |2 3 7 8 9|
-- En Haskell, se definen las dos primeras matrices se definen por
--    ej1 = listArray ((1,1),(2,2)) [0..3]
--    ej2 = listArray ((1,1),(2,3)) [4..9]
-- y el cálculo de la tercera es
--    λ> ampliaColumnas ej1 ej2
--    array ((1,1),(2,5)) [((1,1),0),((1,2),1),((1,3),4),((1,4),5),((1,5),6),
--                         ((2,1),2),((2,2),3),((2,3),7),((2,4),8),((2,5),9)]
--    λ> elems (ampliaColumnas ej1 ej2)
--    [0,1,4,5,6,2,3,7,8,9]
-- ---------------------------------------------------------------------

module Amplia_columnas where

import Data.Array (Array, (!), array, bounds, elems, listArray)
import Data.Matrix (Matrix, (<|>), fromList, ncols, nrows, toList)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck
```

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

ej1, ej2 :: Matriz
ej1 = listArray ((1,1),(2,2)) [0..3]
ej2 = listArray ((1,1),(2,3)) [4..9]

-- 1ª solución
-- ===========

ampliaColumnas1 :: Matriz -> Matriz -> Matriz
ampliaColumnas1 p1 p2 =
  array ((1,1),(m,n1+n2)) [((i,j), f i j) | i <- [1..m], j <- [1..n1+n2]]
    where ((_,_),(m,n1)) = bounds p1
          ((_,_),(_,n2)) = bounds p2
          f i j | j <= n1   = p1!(i,j)
                | otherwise = p2!(i,j-n1)

-- 2ª solución
-- ===========

ampliaColumnas2 :: Matriz -> Matriz -> Matriz
ampliaColumnas2 p1 p2 =
  matriz (matrix p1 <|> matrix p2)

-- (matrix p) es la matriz p en el formato de Data.Matrix. Por ejemplo,
--    λ> ej1
--    array ((1,1),(2,2)) [((1,1),0),((1,2),1),((2,1),2),((2,2),3)]
--    λ> matrix ej1
--    ┌     ┐
--    │ 0 1 │
--    │ 2 3 │
--    └     ┘
--    λ> matrix (ampliaColumnas1 ej1 ej2)
--    ┌           ┐
--    │ 0 1 4 5 6 │
--    │ 2 3 7 8 9 │
--    └           ┘
matrix :: Matriz -> Matrix Int
matrix p = fromList m n (elems p)
  where (_,(m,n)) = bounds p

-- (matriz p) es la matriz p en el formato de Data.Array. Por ejemplo,
--    λ> matriz (fromList 2 3 [1..])
--    array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),((2,1),4),((2,2),5),((2,3),6)]
matriz :: Matrix Int -> Matriz
```

```
matriz p = listArray ((1,1),(nrows p,ncols p)) (toList p)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Matriz -> Matriz -> Matriz) -> Spec
specG ampliaColumnas = do
  it "e1" $
    elems (ampliaColumnas ej1 ej2) `shouldBe` [0,1,4,5,6,2,3,7,8,9]

spec :: Spec
spec = do
  describe "def. 1" $ specG ampliaColumnas1
  describe "def. 2" $ specG ampliaColumnas2

-- La verificación es
--    λ> verifica
--    2 examples, 0 failures

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

data ParMatrices = P Matriz Matriz
  deriving Show

-- parMatricesArbitrario es un generador de pares de matrices con el
-- mismo número de filas.
parMatricesArbitrario :: Gen ParMatrices
parMatricesArbitrario = do
  m  <- arbitrary `suchThat` (> 0)
  n1 <- arbitrary `suchThat` (> 0)
  n2 <- arbitrary `suchThat` (> 0)
  xs <- vector (m * n1)
  ys <- vector (m * n2)
  return (P (listArray ((1,1),(m,n1)) xs)
            (listArray ((1,1),(m,n2)) ys))

-- ParMatrices es una subclase de Arbitrary
instance Arbitrary ParMatrices where
  arbitrary = parMatricesArbitrario

-- La propiedad es
```

```haskell
prop_ampliaColumna :: ParMatrices -> Bool
prop_ampliaColumna (P p q) =
  ampliaColumnas1 p q == ampliaColumnas2 p q

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

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

-- La comparación es
--    λ> ej2000 = listArray ((1,1),(2000,2000)) [1..]
--    λ> maximum (ampliaColumnas1 ej2000 ej2000)
--    4000000
--    (6.26 secs, 6,049,256,416 bytes)
--    λ> maximum (ampliaColumnas2 ej2000 ej2000)
--    4000000
--    (1.90 secs, 2,625,302,000 bytes)
```

Ejercicio 21

Regiones determinadas por n rectas del plano

```
-- ---------------------------------------------------------------------
-- En los siguientes dibujos se observa que el número máximo de regiones
-- en el plano generadas con 1, 2 ó 3 líneas son 2, 4 ó 7,
-- respectivamente.
--
--                        \  |
--                         \5|
--                          \|
--                           \
--                           |\
--                           | \
--                 |         |  \
--     1        1 | 3     1 | 3 \  6
--    ------   ---|---   ---|----\---
--     2        2 | 4     2 | 4   \ 7
--                |         |      \
--
-- Definir la función
--    regiones :: Integer -> Integer
-- tal que (regiones n) es el número máximo de regiones en el plano
-- generadas con n líneas. Por ejemplo,
--    regiones 1     ==  2
--    regiones 2     ==  4
--    regiones 3     ==  7
--    regiones 100   ==  5051
--    regiones 1000  ==  500501
--    regiones 10000 ==  50005001
--    length (show (regiones (10^(10^5)))) ==  200000
--    length (show (regiones (10^(10^6)))) ==  2000000
```

```haskell
--    length (show (regiones (10^(10^7)))) ==  20000000
-- ----------------------------------------------------------------------

module Regiones where

import Data.List (genericIndex)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

regiones1 :: Integer -> Integer
regiones1 0 = 1
regiones1 n = regiones1 (n-1) + n

-- 2ª solución
-- ===========

regiones2 :: Integer -> Integer
regiones2 n = 1 + sum [0..n]

-- 3ª solución
-- ===========

regiones3 :: Integer -> Integer
regiones3 n = 1 + sumas `genericIndex` n

-- (sumas n) es la suma 0 + 1 + 2 +...+ n. Por ejemplo,
--    take 10 sumas  ==  [0,1,3,6,10,15,21,28,36,45]
sumas :: [Integer]
sumas = scanl1 (+) [0..]

-- 4ª solución
-- ===========

regiones4 :: Integer -> Integer
regiones4 n = 1 + n*(n+1) `div` 2

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec
```

```
specG :: (Integer -> Integer) -> Spec
specG regiones = do
  it "e1" $
    regiones 1     `shouldBe`  2
  it "e2" $
    regiones 2     `shouldBe`  4
  it "e3" $
    regiones 3     `shouldBe`  7
  it "e4" $
    regiones 100   `shouldBe`  5051

spec :: Spec
spec = do
  describe "def. 1" $ specG regiones1
  describe "def. 2" $ specG regiones2
  describe "def. 3" $ specG regiones3
  describe "def. 4" $ specG regiones4

-- La verificación es
--    λ> verifica
--    16 examples, 0 failures

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

-- La propiedad es
prop_regiones :: Positive Integer -> Bool
prop_regiones (Positive n) =
  all (== regiones1 n)
      [regiones2 n,
       regiones3 n,
       regiones4 n]

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

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

-- La comparación es
--    λ> regiones1 (4*10^6)
--    8000002000001
--    (2.20 secs, 938,105,888 bytes)
--    λ> regiones2 (4*10^6)
```

```
--    8000002000001
--    (0.77 secs, 645,391,624 bytes)
--    λ> regiones3 (4*10^6)
--    8000002000001
--    (1.22 secs, 1,381,375,296 bytes)
--    λ> regiones4 (4*10^6)
--    8000002000001
--    (0.01 secs, 484,552 bytes)
```

Ejercicio 22

Elemento más repetido de manera consecutiva

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    masRepetido :: Ord a => [a] -> (a,Int)
-- tal que (masRepetido xs) es el elemento de xs que aparece más veces
-- de manera consecutiva en la lista junto con el número de sus
-- apariciones consecutivas; en caso de empate, se devuelve el mayor de
-- dichos elementos. Por ejemplo,
--    masRepetido [1,1,4,4,1]  ==  (4,2)
--    masRepetido [4,4,1,1,5]  ==  (4,2)
--    masRepetido "aadda"      ==  ('d',2)
--    masRepetido "ddaab"      ==  ('d',2)
--    masRepetido (show (product [1..5*10^4]))  ==  ('0',12499)
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Mas_repetido where

import Data.List (group)
import Data.Tuple (swap)
import Control.Arrow ((&&&))
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

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

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

-- 2ª solución
-- ===========

masRepetido2 :: Ord a => [a] -> (a,Int)
masRepetido2 (x:xs)
  | null xs'   = (x,length (x:xs))
  | m > n      = (x,m)
  | m == n     = (max x u,m)
  | otherwise  = (u,n)
  where xs'    = dropWhile (== x) xs
        m      = length (takeWhile (==x) (x:xs))
        (u,n)  = masRepetido2 xs'

-- 3ª solución
-- ===========

masRepetido3 :: Ord a => [a] -> (a,Int)
masRepetido3 xs = (n,z)
  where (z,n) = maximum [(1 + length ys,y) | (y:ys) <- group xs]

-- 4ª solución
-- ===========

masRepetido4 :: Ord a => [a] -> (a,Int)
masRepetido4 xs =
  swap (maximum [(1 + length ys,y) | (y:ys) <- group xs])

-- 5ª solución
-- ===========

masRepetido5 :: Ord a => [a] -> (a,Int)
masRepetido5 xs =
  swap (maximum (map (\ys -> (length ys, head ys)) (group xs)))

-- 6ª solución
-- ===========

masRepetido6 :: Ord a => [a] -> (a,Int)
masRepetido6 =
```

```
  swap . maximum . map ((,) <$> length <*> head) . group

-- 7ª solución
-- ============

masRepetido7 :: Ord a => [a] -> (a,Int)
masRepetido7 =
  swap . maximum . map (length &&& head) . group

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> (Int,Int)) -> Spec
specG masRepetido = do
  it "e1" $
    masRepetido [1,1,4,4,1]  `shouldBe`  (4,2)
  it "e2" $
    masRepetido [4,4,1,1,5]  `shouldBe`  (4,2)

spec :: Spec
spec = do
  describe "def. 1" $ specG masRepetido1
  describe "def. 2" $ specG masRepetido2
  describe "def. 3" $ specG masRepetido3
  describe "def. 4" $ specG masRepetido4
  describe "def. 5" $ specG masRepetido5
  describe "def. 6" $ specG masRepetido6
  describe "def. 7" $ specG masRepetido7

-- La verificación es
--    λ> verifica
--    14 examples, 0 failures

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

-- La propiedad es
prop_masRepetido :: NonEmptyList Int -> Bool
prop_masRepetido (NonEmpty xs) =
  all (== masRepetido1 xs)
      [masRepetido2 xs,
       masRepetido3 xs,
```

```
       masRepetido4 xs,
       masRepetido5 xs,
       masRepetido6 xs,
       masRepetido7 xs]

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

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

-- La comparación es
--    λ> masRepetido1 (show (product [1..3*10^4]))
--    ('0',7498)
--    (3.72 secs, 2,589,930,952 bytes)
--    λ> masRepetido2 (show (product [1..3*10^4]))
--    ('0',7498)
--    (1.27 secs, 991,406,232 bytes)
--    λ> masRepetido3 (show (product [1..3*10^4]))
--    ('0',7498)
--    (0.85 secs, 945,399,976 bytes)
--    λ> masRepetido4 (show (product [1..3*10^4]))
--    ('0',7498)
--    (0.86 secs, 945,399,888 bytes)
--    λ> masRepetido5 (show (product [1..3*10^4]))
--    ('0',7498)
--    (0.80 secs, 943,760,760 bytes)
--    λ> masRepetido6 (show (product [1..3*10^4]))
--    ('0',7498)
--    (0.78 secs, 945,400,400 bytes)
--    λ> masRepetido7 (show (product [1..3*10^4]))
--    ('0',7498)
--    (0.78 secs, 942,122,088 bytes)
--
--    λ> masRepetido2 (show (product [1..5*10^4]))
--    ('0',12499)
--    (3.27 secs, 2,798,156,008 bytes)
--    λ> masRepetido3 (show (product [1..5*10^4]))
--    ('0',12499)
--    (2.20 secs, 2,716,952,408 bytes)
--    λ> masRepetido4 (show (product [1..5*10^4]))
--    ('0',12499)
--    (2.22 secs, 2,716,952,320 bytes)
--    λ> masRepetido5 (show (product [1..5*10^4]))
```

```
--    ('0',12499)
--    (2.18 secs, 2,714,062,328 bytes)
--    λ> masRepetido6 (show (product [1..5*10^4]))
--    ('0',12499)
--    (2.17 secs, 2,716,952,832 bytes)
--    λ> masRepetido7 (show (product [1..5*10^4]))
--    ('0',12499)
--    (2.17 secs, 2,711,172,792 bytes)
```

Ejercicio 23

Número de pares de elementos adyacentes iguales en una matriz

```haskell
-- ----------------------------------------------------------------------
-- Una matriz se puede representar mediante una lista de listas. Por
-- ejemplo, la matriz
--    |2 1 5|
--    |4 3 7|
-- se puede representar mediante la lista
--    [[2,1,5],[4,3,7]]
--
-- Definir la función
--    numeroParesAdyacentesIguales :: Eq a => [[a]] -> Int
-- tal que (numeroParesAdyacentesIguales xss) es el número de pares de
-- elementos consecutivos (en la misma fila o columna) iguales de la
-- matriz xss. Por ejemplo,
--    numeroParesAdyacentesIguales [[0,1],[0,2]]             ==  1
--    numeroParesAdyacentesIguales [[0,0],[1,2]]             ==  1
--    numeroParesAdyacentesIguales [[0,1],[0,0]]             ==  2
--    numeroParesAdyacentesIguales [[1,2],[1,4],[4,4]]       ==  3
--    numeroParesAdyacentesIguales ["ab","aa"]               ==  2
--    numeroParesAdyacentesIguales [[0,0,0],[0,0,0],[0,0,0]] ==  12
--    numeroParesAdyacentesIguales [[0,0,0],[0,1,0],[0,0,0]] ==  8
-- ----------------------------------------------------------------------

module Pares_adyacentes_iguales where

import Data.List (group,transpose)
import Data.Array ((!), listArray)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
```

```haskell
import Test.QuickCheck

-- 1ª solución
-- ===========

numeroParesAdyacentesIguales1 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIguales1 xss =
  length [(i,j) | i <- [1..m-1], j <- [1..n], p!(i,j) == p!(i+1,j)] +
  length [(i,j) | i <- [1..m], j <- [1..n-1], p!(i,j) == p!(i,j+1)]
  where m = length xss
        n = length (head xss)
        p = listArray ((1,1),(m,n)) (concat xss)

-- 2ª solución
-- ===========

numeroParesAdyacentesIguales2 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIguales2 xss =
  numeroParesAdyacentesIgualesFilas xss +
  numeroParesAdyacentesIgualesFilas (transpose xss)

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

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

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

-- (numeroParesAdyacentesIgualesFila xs) es el número de pares de
-- elementos consecutivos de la lista xs. Por ejemplo,
--    numeroParesAdyacentesIgualesFila [5,5,5,2,5] ==  2
```

```haskell
numeroParesAdyacentesIgualesFila :: Eq a => [a] -> Int
numeroParesAdyacentesIgualesFila xs =
  length [(x,y) | (x,y) <- zip xs (tail xs), x == y]

-- 3ª solución
-- ===========

numeroParesAdyacentesIguales3 :: Eq a => [[a]] -> Int
numeroParesAdyacentesIguales3 xss =
  length (concatMap tail (concatMap group (xss ++ transpose xss)))

-- 4ª solución
-- ===========

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([[Int]] -> Int) -> Spec
specG numeroParesAdyacentesIguales = do
  it "e1" $
    numeroParesAdyacentesIguales [[0,1],[0,2]]             `shouldBe` 1
  it "e2" $
    numeroParesAdyacentesIguales [[0,0],[1,2]]             `shouldBe` 1
  it "e3" $
    numeroParesAdyacentesIguales [[0,1],[0,0]]             `shouldBe` 2
  it "e4" $
    numeroParesAdyacentesIguales [[1,2],[1,4],[4,4]]       `shouldBe` 3
  it "e5" $
    numeroParesAdyacentesIguales [[0,0,0],[0,0,0],[0,0,0]] `shouldBe` 12
  it "e6" $
    numeroParesAdyacentesIguales [[0,0,0],[0,1,0],[0,0,0]] `shouldBe` 8

spec :: Spec
spec = do
  describe "def. 1" $ specG numeroParesAdyacentesIguales1
  describe "def. 2" $ specG numeroParesAdyacentesIguales2
  describe "def. 3" $ specG numeroParesAdyacentesIguales3
  describe "def. 4" $ specG numeroParesAdyacentesIguales4
```

```haskell
-- La verificación es
--    λ> verifica
--    24 examples, 0 failures

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

newtype Matriz = M [[Int]]
  deriving Show

-- Generador de matrices arbitrarias. Por ejemplo,
--    λ> generate matrizArbitraria
--    M [[-3,0],[8,-6],[-13,-13],[10,8],[14,29]]
--    λ> generate matrizArbitraria
--    M [[11,9,4,-25,-29,30,-18],[13,8,-2,-22,29,-3,-13]]
matrizArbitraria :: Gen Matriz
matrizArbitraria = do
  m <- chooseInt (1,10)
  n <- chooseInt (1,10)
  xss <- vectorOf m (vectorOf n arbitrary)
  return (M xss)

-- Matriz es una subclase de Arbitrary.
instance Arbitrary Matriz where
  arbitrary = matrizArbitraria

-- La propiedad es
prop_numeroParesAdyacentesIguales :: Matriz -> Bool
prop_numeroParesAdyacentesIguales (M xss) =
  all (== numeroParesAdyacentesIguales1 xss)
      [numeroParesAdyacentesIguales2 xss,
       numeroParesAdyacentesIguales3 xss,
       numeroParesAdyacentesIguales4 xss]

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

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

-- La comparación es
--    λ> numeroParesAdyacentesIguales1 (replicate (3*10^3) (replicate (10^3) 0))
--    5996000
```

```
--    (5.51 secs, 4,751,249,472 bytes)
--    λ> numeroParesAdyacentesIguales2 (replicate (3*10^3) (replicate (10^3) 0))
--    5996000
--    (2.62 secs, 1,681,379,960 bytes)
--    λ> numeroParesAdyacentesIguales3 (replicate (3*10^3) (replicate (10^3) 0))
--    5996000
--    (0.48 secs, 1,393,672,616 bytes)
--    λ> numeroParesAdyacentesIguales4 (replicate (3*10^3) (replicate (10^3) 0))
--    5996000
--    (0.38 secs, 1,393,560,848 bytes)
--
--    λ> ej2 <- generate (vectorOf 1000 (vectorOf 1000 (arbitrary)) :: Gen [[Int]])
--    λ> numeroParesAdyacentesIguales1 ej2
--    32593
--    (1.81 secs, 1,771,121,448 bytes)
--    λ> numeroParesAdyacentesIguales2 ej2
--    32593
--    (0.52 secs, 420,330,176 bytes)
--    λ> numeroParesAdyacentesIguales3 ej2
--    32593
--    (0.33 secs, 780,279,752 bytes)
--    λ> numeroParesAdyacentesIguales4 ej2
--    32593
--    (0.25 secs, 780,280,224 bytes)
```

Ejercicio 24

Mayor producto de las ramas de un árbol

```
-- ---------------------------------------------------------------------
-- Los árboles se pueden representar mediante el siguiente tipo de datos
--    data Arbol a = N a [Arbol a]
--      deriving Show
-- Por ejemplo, los árboles
--       1             3
--     /  \           /|\
--    2   3          / | \
--        |         5  4  7
--        4         |     /\
--                  6    2  1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    ej1 = N 1 [N 2 [],N 3 [N 4 []]]
--    ej2 = N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]]
--
-- Definir la función
--    mayorProducto :: (Ord a, Num a) => Arbol a -> a
-- tal que (mayorProducto a) es el mayor producto de las ramas del árbol
-- a. Por ejemplo,
--    λ> mayorProducto (N 1 [N 2 [], N  3 []])
--    3
--    λ> mayorProducto (N 1 [N 8 [], N  4 [N 3 []]])
--    12
--    λ> mayorProducto (N 1 [N 2 [],N 3 [N 4 []]])
--    12
--    λ> mayorProducto (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
--    90
--    λ> mayorProducto (N (-8) [N 0 [N (-9) []],N 6 []])
```

```haskell
--    0
--    λ> a = N (-4) [N (-7) [],N 14 [N 19 []],N (-1) [N (-6) [],N 21 []],N (-4) []]
--    λ> mayorProducto a
--    84
-- ----------------------------------------------------------------------

module Mayor_producto_de_las_ramas_de_un_arbol where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

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

-- 1ª solución
-- ===========

mayorProducto1 :: (Ord a, Num a) => Arbol a -> a
mayorProducto1 a = maximum [product xs | xs <- ramas a]

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

-- 2ª solución
-- ===========

mayorProducto2 :: (Ord a, Num a) => Arbol a -> a
mayorProducto2 a = maximum (map product (ramas a))

-- 3ª solución
-- ===========

mayorProducto3 :: (Ord a, Num a) => Arbol a -> a
mayorProducto3 = maximum . map product . ramas

-- 4º solución
-- ===========

mayorProducto4 :: (Ord a, Num a) => Arbol a -> a
mayorProducto4 = maximum . productosRamas
```

```haskell
-- (productosRamas a) es la lista de los productos de las ramas
-- del árbol a. Por ejemplo,
--    λ> productosRamas (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
--    [90,12,42,21]
productosRamas :: (Ord a, Num a) => Arbol a -> [a]
productosRamas (N x []) = [x]
productosRamas (N x xs) = [x * y | a <- xs, y <- productosRamas a]

-- 5ª solución
-- ===========

mayorProducto5 :: (Ord a, Num a) => Arbol a -> a
mayorProducto5 (N x []) = x
mayorProducto5 (N x xs)
  | x > 0     = x * maximum (map mayorProducto5 xs)
  | x == 0    = 0
  | otherwise = x * minimum (map menorProducto xs)

-- (menorProducto a) es el menor producto de las ramas del árbol
-- a. Por ejemplo,
--    λ> menorProducto (N 1 [N 2 [], N  3 []])
--    2
--    λ> menorProducto (N 1 [N 8 [], N  4 [N 3 []]])
--    8
--    λ> menorProducto (N 1 [N 2 [],N 3 [N 4 []]])
--    2
--    λ> menorProducto (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
--    12
menorProducto :: (Ord a, Num a) => Arbol a -> a
menorProducto (N x []) = x
menorProducto (N x xs)
  | x > 0     = x * minimum (map menorProducto xs)
  | x == 0    = 0
  | otherwise = x * maximum (map mayorProducto5 xs)

-- 6ª solución
-- ===========

mayorProducto6 :: (Ord a, Num a) => Arbol a -> a
mayorProducto6 = maximum . aux
  where aux (N a []) = [a]
        aux (N a b)  = [v,u]
          where u = maximum g
                v = minimum g
                g = map (*a) (concatMap aux b)
```

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

verifica :: IO ()
verifica = hspec spec

specG :: (Arbol Integer -> Integer) -> Spec
specG mayorProducto = do
  it "e1" $
    mayorProducto (N 1 [N 2 [], N  3 []]) `shouldBe`3
  it "e2" $
    mayorProducto (N 1 [N 8 [], N  4 [N 3 []]]) `shouldBe`12
  it "e3" $
    mayorProducto (N 1 [N 2 [],N 3 [N 4 []]]) `shouldBe`12
  it "e4" $
    mayorProducto (N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]])
    `shouldBe` 90
  it "e5" $
    mayorProducto (N (-8) [N 0 [N (-9) []],N 6 []]) `shouldBe` 0
  it "e6" $ do
    let a = N (-4) [N (-7) [],N 14 [N 19 []],N (-1) [N (-6) [],N 21 []],N (-4) []]
    mayorProducto a `shouldBe` 84

spec :: Spec
spec = do
  describe "def. 1" $ specG mayorProducto1
  describe "def. 2" $ specG mayorProducto2
  describe "def. 3" $ specG mayorProducto3
  describe "def. 4" $ specG mayorProducto4
  describe "def. 5" $ specG mayorProducto5
  describe "def. 6" $ specG mayorProducto6

-- La verificación es
--    λ> verifica
--    36 examples, 0 failures

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

-- (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
--   > sample (arbolArbitrario 5 :: Gen (Arbol Int))
--   N 0 [N 0 []]
--   N (-2) []
--   N 4 []
```

```
--   N 2 [N 4 []]
--   N 8 []
--   N (-2) [N (-9) [],N 7 []]
--   N 11 []
--   N (-11) [N 4 [],N 14 []]
--   N 10 [N (-3) [],N 13 []]
--   N 12 [N 11 []]
--   N 20 [N (-18) [],N (-13) []]
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario n = do
  x  <- arbitrary
  ms <- sublistOf [0 .. n `div` 2]
  as <- mapM arbolArbitrario ms
  return (N x as)

-- Arbol es una subclase de Arbitraria
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_mayorProducto :: Arbol Integer -> Bool
prop_mayorProducto a =
  all (== mayorProducto1 a)
      [f a | f <- [ mayorProducto2
                  , mayorProducto3
                  , mayorProducto4
                  , mayorProducto5
                  , mayorProducto6
                  ]]

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

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

-- La comparación es
--    λ> ejArbol <- generate (arbolArbitrario 600 :: Gen (Arbol Integer))
--    λ> mayorProducto1 ejArbol
--    241972765126624149346713600 0
--    (1.87 secs, 1,082,764,480 bytes)
--    λ> mayorProducto2 ejArbol
--    241972765126624149346713600 0
--    (1.57 secs, 1,023,144,008 bytes)
```

```
--    λ> mayorProducto3 ejArbol
--    241972765126624149346713600 0
--    (1.55 secs, 1,023,144,248 bytes)
--    λ> mayorProducto4 ejArbol
--    241972765126624149346713600 0
--    (1.60 secs, 824,473,800 bytes)
--    λ> mayorProducto5 ejArbol
--    241972765126624149346713600 0
--    (0.83 secs, 732,370,352 bytes)
--    λ> mayorProducto6 ejArbol
--    241972765126624149346713600 0
--    (0.98 secs, 817,473,344 bytes)
--
--    λ> ejArbol2 <- generate (arbolArbitrario 700 :: Gen (Arbol Integer))
--    λ> mayorProducto5 ejArbol2
--    104475893739802671550464000000 0
--    (4.94 secs, 4,170,324,376 bytes)
--    λ> mayorProducto6 ejArbol2
--    104475893739802671550464000000 0
--    (5.88 secs, 4,744,782,024 bytes)
```

Ejercicio 25

Biparticiones de una lista

```
-- ------------------------------------------------------------------
-- Definir la función
--    biparticiones :: [a] -> [([a],[a])]
-- tal que (biparticiones xs) es la lista de pares formados por un
-- prefijo de xs y el resto de xs. Por ejemplo,
--    λ> biparticiones [3,2,5]
--    [([],[3,2,5]),([3],[2,5]),([3,2],[5]),([3,2,5],[])]
--    λ> biparticiones "Roma"
--    [("","Roma"),("R","oma"),("Ro","ma"),("Rom","a"),("Roma","")]
-- ------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Biparticiones_de_una_lista where

import Data.List (inits, tails)
import Control.Applicative (liftA2)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

biparticiones1 :: [a] -> [([a],[a])]
biparticiones1 [] = [([],[])]
biparticiones1 (x:xs) =
  ([],x:xs) : [(x:ys,zs) | (ys,zs) <- biparticiones1 xs]

-- 2ª solución
-- ===========

biparticiones2 :: [a] -> [([a],[a])]
```

```haskell
biparticiones2 xs =
  [(take i xs, drop i xs) | i <- [0..length xs]]

-- 3ª solución
-- ===========

biparticiones3 :: [a] -> [([a],[a])]
biparticiones3 xs =
  [splitAt i xs | i <- [0..length xs]]

-- 4ª solución
-- ===========

biparticiones4 :: [a] -> [([a],[a])]
biparticiones4 xs =
  zip (inits xs) (tails xs)

-- 5ª solución
-- ===========

biparticiones5 :: [a] -> [([a],[a])]
biparticiones5 = liftA2 zip inits tails

-- 6ª solución
-- ===========

biparticiones6 :: [a] -> [([a],[a])]
biparticiones6 = zip <$> inits <*> tails

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [([Int],[Int])]) -> Spec
specG biparticiones = do
  it "e1" $
    biparticiones [3,2,5]
    `shouldBe` [([],[3,2,5]),([3],[2,5]),([3,2],[5]),([3,2,5],[])]

spec :: Spec
spec = do
  describe "def. 1" $ specG biparticiones1
  describe "def. 2" $ specG biparticiones2
```

```
  describe "def. 3" $ specG biparticiones3
  describe "def. 4" $ specG biparticiones4
  describe "def. 5" $ specG biparticiones5
  describe "def. 6" $ specG biparticiones6

-- La verificación es
--    λ> verifica
--    6 examples, 0 failures

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

-- La propiedad es
prop_biparticiones :: [Int] -> Bool
prop_biparticiones xs =
  all (== biparticiones1 xs)
      [biparticiones2 xs,
       biparticiones3 xs,
       biparticiones4 xs,
       biparticiones5 xs,
       biparticiones6 xs]

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

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

-- La comparación es
--    λ> length (biparticiones1 [1..6*10^3])
--    6001
--    (2.21 secs, 3,556,073,552 bytes)
--    λ> length (biparticiones2 [1..6*10^3])
--    6001
--    (0.01 secs, 2,508,448 bytes)
--
--    λ> length (biparticiones2 [1..6*10^6])
--    6000001
--    (2.26 secs, 2,016,494,864 bytes)
--    λ> length (biparticiones3 [1..6*10^6])
--    6000001
--    (2.12 secs, 1,584,494,792 bytes)
--    λ> length (biparticiones4 [1..6*10^6])
--    6000001
```

```
--    (0.78 secs, 1,968,494,704 bytes)
--    λ> length (biparticiones5 [1..6*10^6])
--    6000001
--    (0.79 secs, 1,968,494,688 bytes)
--    λ> length (biparticiones6 [1..6*10^6])
--    6000001
--    (0.77 secs, 1,968,494,720 bytes)
--
--    λ> length (biparticiones4 [1..10^7])
--    10000001
--    (1.30 secs, 3,280,495,432 bytes)
--    λ> length (biparticiones5 [1..10^7])
--    10000001
--    (1.42 secs, 3,280,495,416 bytes)
--    λ> length (biparticiones6 [1..10^7])
--    10000001
--    (1.30 secs, 3,280,495,448 bytes)
```

Ejercicio 26

Trenzado de listas

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    trenza :: [a] -> [a] -> [a]
-- tal que (trenza xs ys) es la lista obtenida intercalando los
-- elementos de xs e ys. Por ejemplo,
--    trenza [5,1] [2,7,4]             ==  [5,2,1,7]
--    trenza [5,1,7] [2..]             ==  [5,2,1,3,7,4]
--    trenza [2..] [5,1,7]             ==  [2,5,3,1,4,7]
--    take 8 (trenza [2,4..] [1,5..])  ==  [2,1,4,5,6,9,8,13]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Trenzado_de_listas where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

trenza1 :: [a] -> [a] -> [a]
trenza1 []     _      = []
trenza1 _      []     = []
trenza1 (x:xs) (y:ys) = x : y : trenza1 xs ys

-- 2ª solución
-- ===========

trenza2 :: [a] -> [a] -> [a]
trenza2 (x:xs) (y:ys) = x : y : trenza2 xs ys
trenza2 _      _      = []
```

```haskell
-- 3ª solución
-- ===========

trenza3 :: [a] -> [a] -> [a]
trenza3 xs ys = concat [[x,y] | (x,y) <- zip xs ys]

-- 4ª solución
-- ===========

trenza4 :: [a] -> [a] -> [a]
trenza4 xs ys = concat (zipWith par xs ys)

par :: a -> a -> [a]
par x y = [x,y]

-- 5ª solución
-- ===========

-- Explicación de eliminación de argumentos en composiciones con varios
-- argumentos:

f :: Int -> Int
f x = x + 1

g :: Int -> Int -> Int
g x y = x + y

h1, h2, h3, h4, h5, h6, h7 :: Int -> Int -> Int
h1 x y  = f (g x y)
h2 x y  = f ((g x) y)
h3 x y  = (f . (g x)) y
h4 x    = f . (g x)
h5 x    = (f .) (g x)
h6 x    = ((f .) . g) x
h7      = (f .) . g

prop_composicion :: Int -> Int -> Bool
prop_composicion x y =
  all (== h1 x y)
      [p x y | p <- [h2, h3, h4, h5, h6, h7]]

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

```haskell
-- En general,
--    f . g             --> \x -> f (g x)
--    (f .) . g         --> \x y -> f (g x y)
--    ((f .) .) . g     --> \x y z -> f (g x y z)
--    (((f .) .) .) . g --> \w x y z -> f (g w x y z)

trenza5 :: [a] -> [a] -> [a]
trenza5 = (concat .) . zipWith par

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [Int] -> [Int]) -> Spec
specG trenza = do
  it "e1" $
    trenza [5,1] [2,7,4]             `shouldBe`  [5,2,1,7]
  it "e2" $
    trenza [5,1,7] [2..]             `shouldBe`  [5,2,1,3,7,4]
  it "e3" $
    trenza [2..] [5,1,7]             `shouldBe`  [2,5,3,1,4,7]
  it "e4" $
    take 8 (trenza [2,4..] [1,5..])  `shouldBe`  [2,1,4,5,6,9,8,13]

spec :: Spec
spec = do
  describe "def. 1" $ specG trenza1
  describe "def. 2" $ specG trenza2
  describe "def. 3" $ specG trenza3
  describe "def. 4" $ specG trenza4
  describe "def. 4" $ specG trenza5

-- La verificación es
--    λ> verifica
--    20 examples, 0 failures

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

-- La propiedad es
prop_trenza :: [Int] -> [Int] -> Bool
prop_trenza xs ys =
  all (== trenza1 xs ys)
```

```
      [trenza2 xs ys,
       trenza3 xs ys,
       trenza4 xs ys,
       trenza5 xs ys]

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

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

-- La comparación es
--    λ> last (trenza1 [1,1..] [1..4*10^6])
--    4000000
--    (2.33 secs, 1,472,494,952 bytes)
--    λ> last (trenza2 [1,1..] [1..4*10^6])
--    4000000
--    (2.24 secs, 1,376,494,928 bytes)
--    λ> last (trenza3 [1,1..] [1..4*10^6])
--    4000000
--    (1.33 secs, 1,888,495,048 bytes)
--    λ> last (trenza4 [1,1..] [1..4*10^6])
--    4000000
--    (0.76 secs, 1,696,494,968 bytes)
--    λ> last (trenza5 [1,1..] [1..4*10^6])
--    4000000
--    (0.76 secs, 1,696,495,064 bytes)
```

Ejercicio 27

Números triangulares con n cifras distintas

```
-- ---------------------------------------------------------------------
-- Los números triangulares se forman como sigue
--
--    *     *      *
--         * *    * *
--               * * *
--    1     3      6
--
-- La sucesión de los números triangulares se obtiene sumando los
-- números naturales. Así, los 5 primeros números triangulares son
--     1 = 1
--     3 = 1 + 2
--     6 = 1 + 2 + 3
--    10 = 1 + 2 + 3 + 4
--    15 = 1 + 2 + 3 + 4 + 5
--
-- Definir la función
--    triangularesConCifras :: Int -> [Integer]
-- tal que (triangularesConCifras n) es la lista de los números
-- triangulares con n cifras distintas. Por  ejemplo,
--    take 6 (triangularesConCifras 1)   ==  [1,3,6,55,66,666]
--    take 6 (triangularesConCifras 2)   ==  [10,15,21,28,36,45]
--    take 6 (triangularesConCifras 3)   ==  [105,120,136,153,190,210]
--    take 5 (triangularesConCifras 4)   ==  [1035,1275,1326,1378,1485]
--    take 2 (triangularesConCifras 10)  ==  [1062489753,1239845706]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```haskell
module Triangulares_con_cifras where

import Data.List (nub)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

triangularesConCifras1 :: Int -> [Integer]
triangularesConCifras1 n =
  [x | x <- triangulares1,
       nCifras x == n]

-- triangulares1 es la lista de los números triangulares. Por ejemplo,
--    take 10 triangulares1 == [1,3,6,10,15,21,28,36,45,55]
triangulares1 :: [Integer]
triangulares1 = map triangular [1..]

triangular :: Integer -> Integer
triangular 1 = 1
triangular n = triangular (n-1) + n

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

-- 2ª solución
-- ===========

triangularesConCifras2 :: Int -> [Integer]
triangularesConCifras2 n =
  [x | x <- triangulares2,
       nCifras x == n]

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

-- 3ª solución
-- ===========

triangularesConCifras3 :: Int -> [Integer]
triangularesConCifras3 n =
```

```haskell
  [x | x <- triangulares3,
       nCifras x == n]

triangulares3 :: [Integer]
triangulares3 = 1 : [x+y | (x,y) <- zip [2..] triangulares3]

-- 4ª solución
-- ===========

triangularesConCifras4 :: Int -> [Integer]
triangularesConCifras4 n =
  [x | x <- triangulares4,
       nCifras x == n]

triangulares4 :: [Integer]
triangulares4 = 1 : zipWith (+) [2..] triangulares4

-- 5ª solución
-- ===========

triangularesConCifras5 :: Int -> [Integer]
triangularesConCifras5 n =
  [x | x <- triangulares5,
       nCifras x == n]

triangulares5 :: [Integer]
triangulares5 = scanl (+) 1 [2..]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> [Integer]) -> Spec
specG triangularesConCifras = do
  it "e1" $
    take 6 (triangularesConCifras 1) `shouldBe` [1,3,6,55,66,666]
  it "e2" $
    take 6 (triangularesConCifras 2) `shouldBe` [10,15,21,28,36,45]
  it "e3" $
    take 6 (triangularesConCifras 3) `shouldBe` [105,120,136,153,190,210]
  it "e4" $
    take 5 (triangularesConCifras 4) `shouldBe` [1035,1275,1326,1378,1485]
```

```
spec :: Spec
spec = do
  describe "def. 1" $ specG triangularesConCifras1
  describe "def. 2" $ specG triangularesConCifras2
  describe "def. 3" $ specG triangularesConCifras3
  describe "def. 4" $ specG triangularesConCifras4
  describe "def. 5" $ specG triangularesConCifras5

-- La verificación es
--    λ> verifica
--    20 examples, 0 failures

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

-- La 1ª propiedad es
prop_triangularesConCifras1 :: Bool
prop_triangularesConCifras1 =
  [take 2 (triangularesConCifras1 n) | n <- [1..7]] ==
  [take 2 (triangularesConCifras2 n) | n <- [1..7]]

-- La comprobación es
--    λ> prop_triangularesConCifras1
--    True

-- La 2ª propiedad es
prop_triangularesConCifras2 :: Int -> Bool
prop_triangularesConCifras2 n =
  all (== take 5 (triangularesConCifras2 n'))
      [take 5 (triangularesConCifras3 n'),
       take 5 (triangularesConCifras4 n'),
       take 5 (triangularesConCifras5 n')]
  where n' = 1 + n `mod` 9

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

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

-- La comparación es
--    λ> (triangularesConCifras1 3) !! 220
--    5456556
--    (2.48 secs, 1,228,690,120 bytes)
```

```
--    λ> (triangularesConCifras2 3) !! 220
--    5456556
--    (0.01 secs, 4,667,288 bytes)
--
--    λ> (triangularesConCifras2 3) !! 600
--    500010500055
--    (1.76 secs, 1,659,299,872 bytes)
--    λ> (triangularesConCifras3 3) !! 600
--    500010500055
--    (1.67 secs, 1,603,298,648 bytes)
--    λ> (triangularesConCifras4 3) !! 600
--    500010500055
--    (1.20 secs, 1,507,298,248 bytes)
--    λ> (triangularesConCifras5 3) !! 600
--    500010500055
--    (1.15 secs, 1,507,298,256 bytes)

-- ----------------------------------------------------------------------
-- §  Referencias                                                       --
-- ----------------------------------------------------------------------

-- + M. Keith [On repdigit polygonal numbers](http://bit.ly/1hzZrDk).
-- + OEIS [A045914: Triangular numbers with all digits the
--   same](https://oeis.org/A045914)
-- + OEIS [A213516: Triangular numbers having only 1 or 2 different
--   digits in base 10](https://oeis.org/A213516).
-- + [Series of natural numbers which has all same
--   digits](http://bit.ly/1hA0Sl4).
```

Ejercicio 28

Enumeración de árboles binarios

```
-- ----------------------------------------------------------------------
-- Los árboles binarios se pueden representar mediante el tipo Arbol
-- definido por
--    data Arbol a = H a
--                 | N (Arbol a) a (Arbol a)
--       deriving Show
-- Por ejemplo, el árbol
--         "B"
--         / \
--        /   \
--       /     \
--     "B"     "A"
--     / \     / \
--   "A" "B" "C" "C"
-- se puede definir por
--    ej1 :: Arbol String
--    ej1 = N (N (H "A") "B" (H "B")) "B" (N (H "C") "A" (H "C"))
--
-- Definir la función
--    enumeraArbol :: Arbol t -> Arbol Int
-- tal que (enumeraArbol a) es el árbol obtenido numerando las hojas y
-- los nodos de a desde la hoja izquierda hasta la raíz. Por ejemplo,
--    λ> enumeraArbol ej1
--    N (N (H 0) 1 (H 2)) 3 (N (H 4) 5 (H 6))
-- Gráficamente,
--          3
--         / \
--        /   \
--       /     \
--      1       5
--     / \     / \
--    0   2   4   6
```

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

{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Enumera_arbol where

import Control.Monad.State (State, evalState, get, modify)
import Test.QuickCheck (Arbitrary, Gen, arbitrary, quickCheck, sized)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)

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

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

-- 1ª solución
-- ===========

enumeraArbol1 :: Arbol t -> Arbol Int
enumeraArbol1 a = fst (aux a 0)
  where
    aux :: Arbol a -> Int -> (Arbol Int,Int)
    aux (H _) n     = (H n, n+1)
    aux (N i _ d) n = (N i' n1 d', n2)
      where (i', n1) = aux i n
            (d', n2) = aux d (n1+1)

-- 2ª solución
-- ===========

enumeraArbol2 :: Arbol t -> Arbol Int
enumeraArbol2 a = evalState (aux a) 0
  where
    aux :: Arbol t -> State Int (Arbol Int)
    aux (H _)     = H <$> enumeraNodo
    aux (N i _ d) = do
      i' <- aux i
      n  <- enumeraNodo
      d' <- aux d
      return (N i' n d')
```

```haskell
enumeraNodo :: State Int Int
enumeraNodo = do
  n <- get
  modify succ
  return n

-- 3ª solución
-- ===========

enumeraArbol3 :: Arbol t -> Arbol Int
enumeraArbol3 a = evalState (aux a) 0
  where
    aux :: Arbol t -> State Int (Arbol Int)
    aux (H _)     = H <$> enumeraNodo
    aux (N i _ d) = N <$> aux i <*> enumeraNodo <*> aux d

-- 4ª solución
-- ===========

enumeraArbol4 :: Arbol t -> Arbol Int
enumeraArbol4 a = evalState (traverse enumeraNodo4 a) 0
  where
    enumeraNodo4 :: t -> State Int Int
    enumeraNodo4 _ = do
      n <- get
      modify succ
      return n

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Arbol String -> Arbol Int) -> Spec
specG enumeraArbol = do
  it "e1" $
    enumeraArbol ej1
    `shouldBe` N (N (H 0) 1 (H 2)) 3 (N (H 4) 5 (H 6))

spec :: Spec
spec = do
  describe "def. 1" $ specG enumeraArbol1
  describe "def. 2" $ specG enumeraArbol2
  describe "def. 3" $ specG enumeraArbol3
```

```
  describe "def. 4" $ specG enumeraArbol4

-- La verificación es
--    λ> verifica
--    4 examples, 0 failures

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

-- (arbolArbitrario n) genera un árbol aleatorio de orden n. Por
-- ejemplo,
--    λ> generate (arbolArbitrario 3 :: Gen (Arbol Int))
--    N (N (H 19) 0 (H (-27))) 21 (N (H 2) 17 (H 26))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario n | n <= 0    = H <$> arbitrary
                  | otherwise = N <$> subarbol <*> arbitrary <*> subarbol
  where subarbol = arbolArbitrario (n `div` 2)

-- Arbol es una subclase de Arbitrary.
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_enumeraArbol :: Arbol Int -> Bool
prop_enumeraArbol a =
  all (== enumeraArbol1 a)
      [enumeraArbol2 a,
       enumeraArbol3 a,
       enumeraArbol4 a]

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

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

-- (arbol n) es el árbol completo de profundidad n. Por ejemplo,
--   λ> arbol 2
--   N (N (H 0) 0 (H 0)) 0 (N (H 0) 0 (H 0))
arbol :: Int -> Arbol Int
arbol 0 = H 0
arbol n = N (arbol (n-1)) 0 (arbol (n-1))

-- (maximo a) es el máximo de los elementos de a. Por ejemplo,
```

```haskell
--    maximo ej1  ==  "C"
maximo :: Ord a => Arbol a -> a
maximo (H x) = x
maximo (N i x d) = maximum [maximo i, x, maximo d]

-- La comparación es
--    λ> maximo (enumeraArbol1 (arbol 19))
--    1048574
--    (1.22 secs, 755,475,496 bytes)
--    λ> maximo (enumeraArbol2 (arbol 19))
--    1048574
--    (2.21 secs, 1,644,666,792 bytes)
--    λ> maximo (enumeraArbol3 (arbol 19))
--    1048574
--    (2.44 secs, 1,799,855,984 bytes)
--    λ> maximo (enumeraArbol4 (arbol 19))
--    1048574
--    (2.89 secs, 1,753,719,616 bytes)
```

Ejercicio 29

Algún vecino menor

```haskell
-- ----------------------------------------------------------------------
-- Las matrices puede representarse mediante tablas cuyos índices son
-- pares de números naturales. Su tipo se define por
--    type Matriz = Array (Int,Int) Int
-- Por ejemplo, la matriz
--    |9 4 6 5|
--    |8 1 7 3|
--    |4 2 5 4|
-- se define por
--    ej :: Matriz
--    ej = listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,4,2,5,4]
--
-- Los vecinos de un elemento son los que están a un paso en la misma
-- fila, columna o diagonal. Por ejemplo, en la matriz anterior, el 1
-- tiene 8 vecinos (el 9, 4, 6, 8, 7, 4, 2 y 5) pero el 9 sólo tiene 3
-- vecinos (el 4, 8 y 1).
--
-- Definir la función
--    algunoMenor :: Matriz -> [Int]
-- tal que (algunoMenor p) es la lista de los elementos de p que tienen
-- algún vecino menor que él. Por ejemplo,
--    algunoMenor ej == [9,4,6,5,8,7,4,2,5,4]
-- pues sólo el 1 y el 3 no tienen ningún vecino menor en la matriz.
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Algun_vecino_menor where

import Data.Array (Array, (!), bounds, indices, inRange, listArray)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (Arbitrary, Gen, arbitrary, chooseInt, quickCheck,
```

```
                      vectorOf)

type Matriz = Array (Int,Int) Int

ej :: Matriz
ej = listArray ((1,1),(3,4)) [9,4,6,5,8,1,7,3,4,2,5,4]

type Pos = (Int,Int)

-- 1ª solución
-- ===========

algunoMenor1 :: Matriz -> [Int]
algunoMenor1 a =
  [a!p| p <- indices a,
        any (< a!p) (vecinos1 a p)]

-- (vecinos q p) es la lista de los vecinos en la matriz a de la
-- posición p. Por ejemplo,
--    vecinos1 ej (2,2)  ==  [9,4,6,8,7,4,2,5]
--    vecinos1 ej (1,1)  ==  [4,8,1]
vecinos1 :: Matriz -> Pos -> [Int]
vecinos1 a p =
  [a!p' | p' <- posicionesVecinos1 a p]

-- (posicionesVecinos a p) es la lista de las posiciones de los
-- vecino de p en la matriz a. Por ejemplo,
--    λ> posicionesVecinos1 3 3 (2,2)
--    [(1,1),(1,2),(1,3),(2,1),(2,3),(3,1),(3,2),(3,3)]
--    λ> posicionesVecinos1 3 3 (1,1)
--    [(1,2),(2,1),(2,2)]
posicionesVecinos1 :: Matriz -> Pos -> [Pos]
posicionesVecinos1 a (i,j) =
  [(i+di,j+dj) | (di,dj) <- [(-1,-1),(-1,0),(-1,1),
                             ( 0,-1),       ( 0,1),
                             ( 1,-1),( 1,0),( 1,1)],
                 inRange (bounds a) (i+di,j+dj)]

-- 2ª solución
-- ===========

algunoMenor2 :: Matriz -> [Int]
algunoMenor2 a =
  [a!p | p <- indices a,
         any (<a!p) (vecinos2 p)]
```

```
  where
    vecinos2 p =
      [a!p' | p' <- posicionesVecinos2 p]
    posicionesVecinos2 (i,j) =
      [(i+di,j+dj) | (di,dj) <- [(-1,-1),(-1,0),(-1,1),
                                 ( 0,-1),       ( 0,1),
                                 ( 1,-1),( 1,0),( 1,1)],
                     inRange (bounds a) (i+di,j+dj)]

-- 3ª solución
-- ===========

algunoMenor3 :: Matriz -> [Int]
algunoMenor3 a =
  [a!p | p <- indices a,
         any (<a!p) (vecinos3 p)]
  where
    vecinos3 p =
      [a!p' | p' <- posicionesVecinos3 p]
    posicionesVecinos3 (i,j) =
      [(i',j') | i' <- [i-1..i+1],
                 j' <- [j-1..j+1],
                 (i',j') /= (i,j),
                 inRange (bounds a) (i',j')]

-- 4ª solución
-- ===========

algunoMenor4 :: Matriz -> [Int]
algunoMenor4 a =
  [a!p | p <- indices a,
         any (<a!p) (vecinos4 p)]
  where
    vecinos4 p =
      [a!p' | p' <- posicionesVecinos4 p]
    posicionesVecinos4 (i,j) =
      [(i',j') | i' <- [max 1 (i-1)..min m (i+1)],
                 j' <- [max 1 (j-1)..min n (j+1)],
                 (i',j') /= (i,j)]
      where (_,(m,n)) = bounds a


-- 5ª solución
-- ===========
```

```haskell
algunoMenor5 :: Matriz -> [Int]
algunoMenor5 a =
  [a!p | p <- indices a,
         any (<a!p) (vecinos5 p)]
  where
    vecinos5 p =
      [a!p' | p' <- posicionesVecinos5 p]
    posicionesVecinos5 (i,j) =
      [(i-1,j-1) | i > 1, j > 1] ++
      [(i-1,j)   | i > 1]        ++
      [(i-1,j+1) | i > 1, j < n] ++
      [(i,j-1)   | j > 1]        ++
      [(i,j+1)   | j < n]        ++
      [(i+1,j-1) | i < m, j > 1] ++
      [(i+1,j)   | i < m]        ++
      [(i+1,j+1) | i < m, j < n]
      where (_,(m,n)) = bounds a

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Matriz -> [Int]) -> Spec
specG algunoMenor = do
  it "e1" $
    algunoMenor ej `shouldBe` [9,4,6,5,8,7,4,2,5,4]

spec :: Spec
spec = do
  describe "def. 1" $ specG algunoMenor1
  describe "def. 2" $ specG algunoMenor2
  describe "def. 3" $ specG algunoMenor3
  describe "def. 4" $ specG algunoMenor4
  describe "def. 5" $ specG algunoMenor5

-- La verificación es
--    λ> verifica
--    5 examples, 0 failures

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

newtype Matriz2 = M Matriz
```

```
  deriving Show

-- Generador de matrices arbitrarias. Por ejemplo,
--    λ> generate matrizArbitraria
--    M (array ((1,1),(3,4))
--             [((1,1),18),((1,2),6), ((1,3),-23),((1,4),-13),
--              ((2,1),-2),((2,2),22),((2,3),-25),((2,4),-5),
--              ((3,1),2), ((3,2),16),((3,3),-15),((3,4),7)])
matrizArbitraria :: Gen Matriz2
matrizArbitraria = do
  m  <- chooseInt (1,10)
  n  <- chooseInt (1,10)
  xs <- vectorOf (m*n) arbitrary
  return (M (listArray ((1,1),(m,n)) xs))

-- Matriz es una subclase de Arbitrary.
instance Arbitrary Matriz2 where
  arbitrary = matrizArbitraria

-- La propiedad es
prop_algunoMenor :: Matriz2 -> Bool
prop_algunoMenor (M p) =
  all (== algunoMenor1 p)
      [algunoMenor2 p,
       algunoMenor3 p,
       algunoMenor4 p,
       algunoMenor5 p]

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

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

-- La comparación es
--    λ> maximum (algunoMenor1 (listArray ((1,1),(600,800)) [0..]))
--    479999
--    (2.20 secs, 1,350,075,240 bytes)
--    λ> maximum (algunoMenor2 (listArray ((1,1),(600,800)) [0..]))
--    479999
--    (2.24 secs, 1,373,139,968 bytes)
--    λ> maximum (algunoMenor3 (listArray ((1,1),(600,800)) [0..]))
--    479999
--    (2.08 secs, 1,200,734,112 bytes)
```

```
--    λ> maximum (algunoMenor4 (listArray ((1,1),(600,800)) [0..]))
--    479999
--    (2.76 secs, 1,287,653,136 bytes)
--    λ> maximum (algunoMenor5 (listArray ((1,1),(600,800)) [0..]))
--    479999
--    (1.67 secs, 953,937,600 bytes)
```

Ejercicio 30

Reiteración de una función

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    reiteracion :: (a -> a) -> Int -> a -> a
-- tal que (reiteracion f n x) es el resultado de aplicar n veces la
-- función f a x. Por ejemplo,
--    reiteracion (+1) 10 5  ==  15
--    reiteracion (+5) 10 0  ==  50
--    reiteracion (*2)  4 1  ==  16
--    reiteracion (5:)  4 [] ==  [5,5,5,5]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Reiteracion_de_funciones where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (Fun (..), Positive (..), quickCheck)

-- 1ª solución
-- ===========

reiteracion1 :: (a -> a) -> Int -> a -> a
reiteracion1 _ 0 x = x
reiteracion1 f n x = f (reiteracion1 f (n-1) x)

-- 2ª solución
-- ===========

reiteracion2 :: (a -> a) -> Int -> a -> a
reiteracion2 _ 0 = id
reiteracion2 f n = f . reiteracion2 f (n-1)
```

```haskell
-- 3ª solución
-- ===========

reiteracion3 :: (a -> a) -> Int -> a -> a
reiteracion3 _ 0 = id
reiteracion3 f n
  | even n    = reiteracion3 (f . f) (n `div` 2)
  | otherwise = f . reiteracion3 (f . f) (n `div` 2)

-- 4ª solución
-- ===========

reiteracion4 :: (a -> a) -> Int -> a -> a
reiteracion4 f n x = reiteraciones f x !! n

reiteraciones :: (a -> a) -> a -> [a]
reiteraciones f x = x : reiteraciones f (f x)

-- 5ª solución
-- ===========

reiteracion5 :: (a -> a) -> Int -> a -> a
reiteracion5 f n x = iterate f x !! n

-- 6ª solución
-- ===========

-- Se puede eliminar los argumentos de la definición anterior como sigue:
--    reiteracion4 f n x = iterate f x !! n
--    reiteracion4 f n x = ((!!) (iterate f x)) n
--    reiteracion4 f n x = (((!!) . (iterate f)) x) n
--    reiteracion4 f n x = ((!!) . (iterate f)) x n
--    reiteracion4 f n x = flip ((!!) . (iterate f)) n x
--    reiteracion4 f = flip ((!!) . (iterate f))
--    reiteracion4 f = flip (((!!) .) (iterate f))
--    reiteracion4 f = flip (((!!) .) . iterate) f
--    reiteracion4 f = (flip . ((!!) .) . iterate) f
--    reiteracion4   = flip . ((!!) .) . iterate

reiteracion6 :: (a -> a) -> Int -> a -> a
reiteracion6 = flip . ((!!) .) . iterate

-- Verificación
-- ============
```

```
verifica :: IO ()
verifica = hspec spec

specG :: ((Int -> Int) -> Int -> Int -> Int) -> Spec
specG reiteracion = do
  it "e1" $
    reiteracion (+1) 10 5  `shouldBe`  15
  it "e2" $
    reiteracion (+5) 10 0  `shouldBe`  50
  it "e3" $
    reiteracion (*2)  4 1  `shouldBe`  16

spec :: Spec
spec = do
  describe "def. 1" $ specG reiteracion1
  describe "def. 2" $ specG reiteracion2
  describe "def. 3" $ specG reiteracion3
  describe "def. 4" $ specG reiteracion4
  describe "def. 5" $ specG reiteracion5
  describe "def. 6" $ specG reiteracion6

-- La verificación es
--    λ> verifica
--    18 examples, 0 failures

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

-- La propiedad es
prop_reiteracion :: Fun Int Int -> Positive Int -> Int -> Bool
prop_reiteracion (Fun _ f) (Positive n) x =
  all (== reiteracion1 f n x)
      [reiteracion2 f n x,
       reiteracion3 f n x,
       reiteracion4 f n x,
       reiteracion5 f n x,
       reiteracion6 f n x]

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

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

```
-- La comparación es
--    λ> reiteracion1 (+1) (10^7) 0
--    10000000
--    (5.09 secs, 2,505,392,792 bytes)
--    λ> reiteracion2 (+1) (10^7) 0
--    10000000
--    (5.45 secs, 2,896,899,728 bytes)
--    λ> reiteracion3 (+1) (10^7) 0
--    10000000
--    (2.14 secs, 816,909,416 bytes)
--    λ> reiteracion4 (+1) (10^7) 0
--    10000000
--    (4.24 secs, 1,696,899,816 bytes)
--    λ> reiteracion5 (+1) (10^7) 0
--    10000000
--    (2.53 secs, 1,376,899,800 bytes)
--    λ> reiteracion6 (+1) (10^7) 0
--    10000000
--    (2.34 secs, 1,376,899,984 bytes)
```

Ejercicio 31

Pim, Pam, Pum y divisibilidad

```
-- ---------------------------------------------------------------------
-- Definir la función
--    sonido :: Int -> String
-- tal que (sonido n) escribe "Pim" si n es divisible por 3, además
-- escribe "Pam" si n es divisible por 5 y también escribe "Pum" si n es
-- divisible por 7. Por ejemplo,
--    sonido   3  ==  "Pim"
--    sonido   5  ==  "Pam"
--    sonido   7  ==  "Pum"
--    sonido   8  ==  ""
--    sonido   9  ==  "Pim"
--    sonido  15  ==  "PimPam"
--    sonido  21  ==  "PimPum"
--    sonido  35  ==  "PamPum"
--    sonido 105  ==  "PimPamPum"
-- ---------------------------------------------------------------------

module PimPamPum where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

sonido1 :: Int -> String
sonido1 x = concat [z | (n,z) <- zs, n == 0]
  where xs = [rem x 3, rem x 5, rem x 7]
        zs = zip xs ["Pim","Pam","Pum"]

-- 2ª solución
-- ===========
```

```haskell
sonido2 :: Int -> String
sonido2 n = concat (["Pim" | rem n 3 == 0] ++
                    ["Pam" | rem n 5 == 0] ++
                    ["Pum" | rem n 7 == 0])

-- 3ª solución
-- ===========

sonido3 :: Int -> String
sonido3 n = f 3 "Pim" ++ f 5 "Pam" ++ f 7 "Pum"
  where f x c = if rem n x == 0
                then c
                else ""

-- 4ª solución
-- ===========

sonido4 :: Int -> String
sonido4 n =
  concat [s | (s,d) <- zip ["Pim","Pam","Pum"] [3,5,7],
              rem n d == 0]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> String) -> Spec
specG sonido = do
  it "e1" $
    sonido   3  `shouldBe`  "Pim"
  it "e2" $
    sonido   5  `shouldBe`  "Pam"
  it "e3" $
    sonido   7  `shouldBe`  "Pum"
  it "e4" $
    sonido   8  `shouldBe`  ""
  it "e5" $
    sonido   9  `shouldBe`  "Pim"
  it "e6" $
    sonido  15  `shouldBe`  "PimPam"
  it "e7" $
    sonido  21  `shouldBe`  "PimPum"
```

```
  it "e8" $
    sonido  35  `shouldBe`  "PamPum"
  it "e9" $
    sonido 105  `shouldBe`  "PimPamPum"

spec :: Spec
spec = do
  describe "def. 1" $ specG sonido1
  describe "def. 2" $ specG sonido2
  describe "def. 3" $ specG sonido3
  describe "def. 4" $ specG sonido4

-- La verificación es
--    λ> verifica
--    36 examples, 0 failures

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

-- La propiedad es
prop_sonido :: Positive Int -> Bool
prop_sonido (Positive n) =
  all (== sonido1 n)
      [sonido2 n,
       sonido3 n,
       sonido4 n]

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

Ejercicio 32

Código de las alergias

```
-- ---------------------------------------------------------------------
-- Para la determinación de las alergia se utiliza los siguientes
-- códigos para los alérgenos:
--    Huevos ........   1
--    Cacahuetes ....   2
--    Mariscos ......   4
--    Fresas ........   8
--    Tomates .......  16
--    Chocolate .....  32
--    Polen .........  64
--    Gatos ......... 128
-- Así, si Juan es alérgico a los cacahuetes y al chocolate, su
-- puntuación es 34 (es decir, 2+32).
--
-- Los alérgenos se representan mediante el siguiente tipo de dato
--   data Alergeno = Huevos
--                 | Cacahuetes
--                 | Mariscos
--                 | Fresas
--                 | Tomates
--                 | Chocolate
--                 | Polen
--                 | Gatos
--     deriving (Enum, Eq, Show, Bounded)
--
-- Definir la función
--    alergias :: Int -> [Alergeno]
-- tal que (alergias n) es la lista de alergias correspondiente a una
-- puntuación n. Por ejemplo,
--    λ> alergias 1
--    [Huevos]
--    λ> alergias 2
```

```
--    [Cacahuetes]
--    λ> alergias 3
--    [Huevos,Cacahuetes]
--    λ> alergias 5
--    [Huevos,Mariscos]
--    λ> alergias 255
--    [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]
-- ----------------------------------------------------------------------

module Alergias where

import Data.List (subsequences)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

data Alergeno =
    Huevos
  | Cacahuetes
  | Mariscos
  | Fresas
  | Tomates
  | Chocolate
  | Polen
  | Gatos
  deriving (Enum, Eq, Show, Bounded)

-- 1ª solución
-- ===========

alergias1 :: Int -> [Alergeno]
alergias1 n =
  [a | (a,c) <- zip alergenos codigos, c `elem` descomposicion n]

-- codigos es la lista de los códigos de los alergenos.
codigos :: [Int]
codigos = [2^x| x <- [0..7]]

-- (descomposicion n) es la descomposición de n como sumas de potencias
-- de 2. Por ejemplo,
--    descomposicion 3    ==  [1,2]
--    descomposicion 5    ==  [1,4]
--    descomposicion 248  ==  [8,16,32,64,128]
--    descomposicion 255  ==  [1,2,4,8,16,32,64,128]
descomposicion :: Int -> [Int]
descomposicion n =
```

```
  head [xs | xs <- subsequences codigos, sum xs == n]

-- 2ª solución
-- ===========

alergias2 :: Int -> [Alergeno]
alergias2 = map toEnum . codigosAlergias

-- (codigosAlergias n) es la lista de códigos de alergias
-- correspondiente a una puntuación n. Por ejemplo,
--    codigosAlergias 1  ==  [0]
--    codigosAlergias 2  ==  [1]
--    codigosAlergias 3  ==  [0,1]
--    codigosAlergias 4  ==  [2]
--    codigosAlergias 5  ==  [0,2]
--    codigosAlergias 6  ==  [1,2]
codigosAlergias :: Int -> [Int]
codigosAlergias = aux [0..7]
  where aux []     _             = []
        aux (x:xs) n | odd n     = x : aux xs (n `div` 2)
                     | otherwise = aux xs (n `div` 2)

-- 3ª solución
-- ===========

alergias3 :: Int -> [Alergeno]
alergias3 = map toEnum . codigosAlergias3

codigosAlergias3 :: Int -> [Int]
codigosAlergias3 n =
  [x | (x,y) <- zip [0..7] (int2bin n), y == 1]

-- (int2bin n) es la representación binaria del número n. Por ejemplo,
--    int2bin 10  ==  [0,1,0,1]
-- ya que 10 = 0*1 + 1*2 + 0*4 + 1*8
int2bin :: Int -> [Int]
int2bin n | n < 2     = [n]
          | otherwise = n `rem` 2 : int2bin (n `div` 2)

-- 4ª solución
-- ===========

alergias4 :: Int -> [Alergeno]
alergias4 = map toEnum . codigosAlergias4
```

```haskell
codigosAlergias4 :: Int -> [Int]
codigosAlergias4 n =
  map fst (filter ((== 1) . snd) (zip  [0..7] (int2bin n)))

-- 5ª solución
-- ===========

alergias5 :: Int -> [Alergeno]
alergias5 = map (toEnum . fst)
          . filter ((1 ==) . snd)
          . zip [0..7]
          . int2bin

-- 6ª solución
-- ===========

alergias6 :: Int -> [Alergeno]
alergias6 = aux alergenos
  where aux []     _             = []
        aux (x:xs) n | odd n     = x : aux xs (n `div` 2)
                     | otherwise = aux xs (n `div` 2)

-- alergenos es la lista de los alergenos. Por ejemplo.
--    take 3 alergenos  ==  [Huevos,Cacahuetes,Mariscos]
alergenos :: [Alergeno]
alergenos = [minBound..maxBound]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> [Alergeno]) -> Spec
specG alergias = do
  it "e1" $
    alergias 1
    `shouldBe` [Huevos]
  it "e2" $
    alergias 2
    `shouldBe` [Cacahuetes]
  it "e3" $
    alergias 3
    `shouldBe` [Huevos,Cacahuetes]
  it "e4" $
```

```
    alergias 5
    `shouldBe` [Huevos,Mariscos]
  it "e5" $
    alergias 255
    `shouldBe` [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]

spec :: Spec
spec = do
  describe "def. 1" $ specG alergias1
  describe "def. 2" $ specG alergias2
  describe "def. 3" $ specG alergias3
  describe "def. 4" $ specG alergias4
  describe "def. 5" $ specG alergias5

-- La verificación es
--    λ> verifica
--    25 examples, 0 failures

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

-- La propiedad es
prop_alergias :: Property
prop_alergias =
  forAll (arbitrary `suchThat` esValido) $ \n ->
  all (== alergias1 n)
      [alergias2 n,
       alergias3 n,
       alergias4 n,
       alergias5 n,
       alergias6 n]
  where esValido x = 1 <= x && x <= 255

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

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

-- La comparación es
--    λ> last (map alergias1 [1..255])
--    [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]
--    (0.02 secs, 1,657,912 bytes)
--    λ> last (map alergias2 [1..255])
```

```
--    [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]
--    (0.01 secs, 597,080 bytes)
--    λ> last (map alergias3 [1..255])
--    [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]
--    (0.01 secs, 597,640 bytes)
--    λ> last (map alergias4 [1..255])
--    [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]
--    (0.01 secs, 598,152 bytes)
--    λ> last (map alergias5 [1..255])
--    [Huevos,Cacahuetes,Mariscos,Fresas,Tomates,Chocolate,Polen,Gatos]
--    (0.01 secs, 596,888 bytes)
```

Ejercicio 33

Índices de valores verdaderos

```
-- ---------------------------------------------------------------------
-- Definir la función
--    indicesVerdaderos :: [Int] -> [Bool]
-- tal que (indicesVerdaderos xs) es la lista infinita de booleanos tal
-- que sólo son verdaderos los elementos cuyos índices pertenecen a la
-- lista estrictamente creciente xs. Por ejemplo,
--    λ> take 6 (indicesVerdaderos [1,4])
--    [False,True,False,False,True,False]
--    λ> take 6 (indicesVerdaderos [0,2..])
--    [True,False,True,False,True,False]
--    λ> take 3 (indicesVerdaderos [])
--    [False,False,False]
--    λ> take 6 (indicesVerdaderos [1..])
--    [False,True,True,True,True,True]
--    λ> last (take (8*10^7) (indicesVerdaderos [0,5..]))
--    False
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Indices_verdaderos where

import Data.List.Ordered (member)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

indicesVerdaderos1 :: [Int] -> [Bool]
indicesVerdaderos1 []     = repeat False
indicesVerdaderos1 (x:ys) =
```

```
  replicate x False ++ [True] ++ indicesVerdaderos1 [y-x-1 | y <- ys]

-- 2ª solución
-- ===========

indicesVerdaderos2 :: [Int] -> [Bool]
indicesVerdaderos2 = aux 0
  where aux _ []     = repeat False
        aux n (x:xs) | x == n    = True  : aux (n+1) xs
                     | otherwise = False : aux (n+1) (x:xs)

-- 3ª solución
-- ===========

indicesVerdaderos3 :: [Int] -> [Bool]
indicesVerdaderos3 = aux [0..]
  where aux _      []                 = repeat False
        aux (i:is) (x:xs) | i == x    = True  : aux is xs
                          | otherwise = False : aux is (x:xs)

-- 4ª solución
-- ===========

indicesVerdaderos4 :: [Int] -> [Bool]
indicesVerdaderos4 xs = [pertenece x xs | x <- [0..]]

-- (pertenece x ys) se verifica si x pertenece a la lista estrictamente
-- creciente (posiblemente infinita) ys. Por ejemplo,
--    pertenece 9 [1,3..]  ==  True
--    pertenece 6 [1,3..]  ==  False
pertenece :: Int -> [Int] -> Bool
pertenece x ys = x `elem` takeWhile (<=x) ys

-- 5ª solución
-- ===========

indicesVerdaderos5 :: [Int] -> [Bool]
indicesVerdaderos5 xs = map (`pertenece2` xs) [0..]

pertenece2 :: Int -> [Int] -> Bool
pertenece2 x = aux
  where aux [] = False
        aux (y:ys) = case compare x y of
                       LT -> False
                       EQ -> True
```

```
                      GT -> aux ys

-- 6ª solución
-- ===========

indicesVerdaderos6 :: [Int] -> [Bool]
indicesVerdaderos6 xs = map (`member` xs) [0..]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> [Bool]) -> Spec
specG indicesVerdaderos = do
  it "e1" $
    take 6 (indicesVerdaderos [1,4])
    `shouldBe` [False,True,False,False,True,False]
  it "e2" $
    take 6 (indicesVerdaderos [0,2..])
    `shouldBe` [True,False,True,False,True,False]
  it "e3" $
    take 3 (indicesVerdaderos [])
    `shouldBe` [False,False,False]
  it "e4" $
    take 6 (indicesVerdaderos [1..])
    `shouldBe` [False,True,True,True,True,True]

spec :: Spec
spec = do
  describe "def. 1" $ specG indicesVerdaderos1
  describe "def. 2" $ specG indicesVerdaderos2
  describe "def. 3" $ specG indicesVerdaderos3
  describe "def. 4" $ specG indicesVerdaderos4
  describe "def. 5" $ specG indicesVerdaderos5
  describe "def. 6" $ specG indicesVerdaderos6

-- La verificación es
--    λ> verifica
--    24 examples, 0 failures

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

```haskell
-- ListaCreciente es un tipo de dato para generar lista de enteros
-- crecientes arbitrarias.
newtype ListaCreciente = LC [Int]
  deriving Show

-- listaCrecienteArbitraria es un generador de lista de enteros
-- crecientes arbitrarias. Por ejemplo,
--    λ> sample listaCrecienteArbitraria
--    LC []
--    LC [2,5]
--    LC [4,8]
--    LC [6,13]
--    LC [7,15,20,28,33]
--    LC [11,15,20,29,35,40]
--    LC [5,17,25,36,42,50,52,64]
--    LC [9,16,31,33,46,59,74,83,85,89,104,113,118]
--    LC [9,22,29,35,37,49,53,62,68,77,83,100]
--    LC []
--    LC [3,22,25,34,36,51,72,75,89]
listaCrecienteArbitraria :: Gen ListaCreciente
listaCrecienteArbitraria =
  LC . listaCreciente <$> arbitrary

-- (listaCreciente xs) es la lista creciente correspondiente a xs. Por ejemplo,
--    listaCreciente [-1,3,-4,3,0]   ==  [2,6,11,15,16]
listaCreciente :: [Int] -> [Int]
listaCreciente xs =
  scanl1 (+) (map (succ . abs) xs)

-- ListaCreciente está contenida en Arbitrary
instance Arbitrary ListaCreciente where
  arbitrary = listaCrecienteArbitraria

-- La propiedad es
prop_indicesVerdaderos :: ListaCreciente -> Bool
prop_indicesVerdaderos (LC xs) =
  all (== take n (indicesVerdaderos1 xs))
      [take n (f xs) | f <-[indicesVerdaderos2,
                            indicesVerdaderos3,
                            indicesVerdaderos4,
                            indicesVerdaderos5,
                            indicesVerdaderos6]]
  where n = length xs

-- La comprobación es
```

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

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

-- La comparación es
--    λ> last (take (2*10^4) (indicesVerdaderos1 [0,5..]))
--    False
--    (2.69 secs, 2,611,031,544 bytes)
--    λ> last (take (2*10^4) (indicesVerdaderos2 [0,5..]))
--    False
--    (0.03 secs, 10,228,880 bytes)
--
--    λ> last (take (4*10^6) (indicesVerdaderos2 [0,5..]))
--    False
--    (2.37 secs, 1,946,100,856 bytes)
--    λ> last (take (4*10^6) (indicesVerdaderos3 [0,5..]))
--    False
--    (1.54 secs, 1,434,100,984 bytes)
--
--    λ> last (take (6*10^6) (indicesVerdaderos3 [0,5..]))
--    False
--    (2.30 secs, 2,150,900,984 bytes)
--    λ> last (take (6*10^6) (indicesVerdaderos4 [0,5..]))
--    False
--    (1.55 secs, 1,651,701,184 bytes)
--    λ> last (take (6*10^6) (indicesVerdaderos5 [0,5..]))
--    False
--    (0.58 secs, 1,584,514,304 bytes)
--
--    λ> last (take (3*10^7) (indicesVerdaderos5 [0,5..]))
--    False
--    (2.74 secs, 7,920,514,360 bytes)
--    λ> last (take (3*10^7) (indicesVerdaderos6 [0,5..]))
--    False
--    (0.82 secs, 6,960,514,136 bytes)

--    λ> last (take (2*10^4) (indicesVerdaderos1 [0,5..]))
--    False
--    (2.69 secs, 2,611,031,544 bytes)
--    λ> last (take (2*10^4) (indicesVerdaderos6 [0,5..]))
--    False
--    (0.01 secs, 5,154,040 bytes)
```

Ejercicio 34

Descomposiciones triangulares

```
-- ---------------------------------------------------------------------
-- Los números triangulares se forman como sigue
--
--    *     *      *
--         * *    * *
--               * * *
--    1     3      6
--
-- La sucesión de los números triangulares se obtiene sumando los
-- números naturales. Así, los 5 primeros números triangulares son
--     1 = 1
--     3 = 1 + 2
--     6 = 1 + 2 + 3
--    10 = 1 + 2 + 3 + 4
--    15 = 1 + 2 + 3 + 4 + 5
--
-- Definir la función
--    descomposicionesTriangulares :: Int -> [(Int, Int, Int)]
-- tal que (descomposicionesTriangulares n) es la lista de las
-- ternas correspondientes a las descomposiciones de n en tres sumandos,
-- como máximo, formados por números triangulares. Por ejemplo,
--    λ> descomposicionesTriangulares 4
--    []
--    λ> descomposicionesTriangulares 5
--    [(1,1,3)]
--    λ> descomposicionesTriangulares 12
--    [(1,1,10),(3,3,6)]
--    λ> descomposicionesTriangulares 30
--    [(1,1,28),(3,6,21),(10,10,10)]
--    λ> descomposicionesTriangulares 61
--    [(1,15,45),(3,3,55),(6,10,45),(10,15,36)]
--    λ> descomposicionesTriangulares 52
```

```
--    [(1,6,45),(1,15,36),(3,21,28),(6,10,36),(10,21,21)]
--    λ> descomposicionesTriangulares 82
--    [(1,3,78),(1,15,66),(1,36,45),(6,10,66),(6,21,55),(10,36,36)]
--    λ> length (descomposicionesTriangulares (5*10^6))
--    390
-- ------------------------------------------------------------------

module Descomposiciones_triangulares where

import Data.Set (fromList, member)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

descomposicionesTriangulares1 :: Int -> [(Int, Int, Int)]
descomposicionesTriangulares1 n =
  [(x,y,z) | x <- xs,
             y <- xs,
             z <- xs,
             x <= y && y <= z,
             x + y + z == n]
  where xs = takeWhile (<=n) triangulares

-- triangulares es la lista de los números triangulares. Por ejemplo,
--    take 9 triangulares  ==  [1,3,6,10,15,21,28,36,45]
triangulares :: [Int]
triangulares = scanl (+) 1 [2..]

-- 2ª solución
-- ===========

descomposicionesTriangulares2 :: Int -> [(Int, Int, Int)]
descomposicionesTriangulares2 n =
  [(x,y,z) | x <- xs,
             y <- xs,
             x <= y,
             z <- xs,
             y <= z,
             x + y + z == n]
  where xs = takeWhile (<=n) triangulares

-- 3ª solución
-- ===========
```

```haskell
descomposicionesTriangulares3 :: Int -> [(Int, Int, Int)]
descomposicionesTriangulares3 n =
  [(x,y,z) | x <- xs,
             y <- xs,
             x <= y,
             let z = n - x - y,
             y <= z,
             z `elem` xs]
  where xs = takeWhile (<=n) triangulares

-- 4ª solución
-- ============

descomposicionesTriangulares4 :: Int -> [(Int, Int, Int)]
descomposicionesTriangulares4 n =
  [(x,y,n-x-y) | x <- xs,
                 y <- dropWhile (<x) xs,
                 let z = n - x - y,
                 y <= z,
                 z `elem` xs]
  where xs = takeWhile (<=n) triangulares

-- 5ª solución
-- ============

descomposicionesTriangulares5 :: Int -> [(Int, Int, Int)]
descomposicionesTriangulares5 n =
  [(x,y,z) | x <- xs,
             y <- dropWhile (<x) xs,
             let z = n - x - y,
             y <= z,
             z `member` ys]
  where
    xs = takeWhile (<=n) triangulares
    ys = fromList xs

-- Verificación
-- =============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> [(Int, Int, Int)]) -> Spec
specG descomposicionesTriangulares = do
```

```
  it "e1" $
    descomposicionesTriangulares  4 `shouldBe`
      []
  it "e2" $
    descomposicionesTriangulares  5 `shouldBe`
      [(1,1,3)]
  it "e3" $
    descomposicionesTriangulares 12 `shouldBe`
      [(1,1,10),(3,3,6)]
  it "e4" $
    descomposicionesTriangulares 30 `shouldBe`
      [(1,1,28),(3,6,21),(10,10,10)]
  it "e5" $
    descomposicionesTriangulares 61 `shouldBe`
      [(1,15,45),(3,3,55),(6,10,45),(10,15,36)]
  it "e6" $
    descomposicionesTriangulares 52 `shouldBe`
      [(1,6,45),(1,15,36),(3,21,28),(6,10,36),(10,21,21)]
  it "e7" $
    descomposicionesTriangulares 82 `shouldBe`
      [(1,3,78),(1,15,66),(1,36,45),(6,10,66),(6,21,55),(10,36,36)]

spec :: Spec
spec = do
  describe "def. 1" $ specG descomposicionesTriangulares1
  describe "def. 2" $ specG descomposicionesTriangulares2
  describe "def. 3" $ specG descomposicionesTriangulares3
  describe "def. 4" $ specG descomposicionesTriangulares4
  describe "def. 5" $ specG descomposicionesTriangulares5

-- La verificación es
--    λ> verifica
--    28 examples, 0 failures

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

-- La propiedad es
prop_descomposicionesTriangulares_equiv ::  Positive Int -> Bool
prop_descomposicionesTriangulares_equiv (Positive n) =
  all (== descomposicionesTriangulares1 n)
      [descomposicionesTriangulares2 n,
       descomposicionesTriangulares3 n,
       descomposicionesTriangulares4 n,
       descomposicionesTriangulares5 n]
```

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

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

-- La comparación es
--   λ> last (descomposicionesTriangulares1 (2*10^4))
--   (5671,6328,8001)
--   (3.34 secs, 1,469,517,168 bytes)
--   λ> last (descomposicionesTriangulares2 (2*10^4))
--   (5671,6328,8001)
--   (1.29 secs, 461,433,928 bytes)
--   λ> last (descomposicionesTriangulares3 (2*10^4))
--   (5671,6328,8001)
--   (0.08 secs, 6,574,056 bytes)
--
--   λ> last (descomposicionesTriangulares3 (5*10^5))
--   (140185,148240,211575)
--   (2.12 secs, 151,137,280 bytes)
--   λ> last (descomposicionesTriangulares4 (5*10^5))
--   (140185,148240,211575)
--   (2.30 secs, 103,280,216 bytes)
--   λ> last (descomposicionesTriangulares5 (5*10^5))
--   (140185,148240,211575)
--   (0.30 secs, 103,508,368 bytes)
```

Ejercicio 35

Número de inversiones

```haskell
-- ----------------------------------------------------------------------
-- Se dice que en una sucesión de números x(1), x(2), ..., x(n) hay una
-- inversión cuando existe un par de números x(i) > x(j), siendo i < j.
-- Por ejemplo, en la permutación 2, 1, 4, 3 hay dos inversiones
-- (2 antes que 1 y 4 antes que 3) y en la permutación 4, 3, 1, 2 hay
-- cinco inversiones (4 antes 3, 4 antes 1, 4 antes 2, 3 antes 1,
-- 3 antes 2).
--
-- Definir la función
--    numeroInversiones :: Ord a => [a] -> Int
-- tal que (numeroInversiones xs) es el número de inversiones de xs. Por
-- ejemplo,
--    numeroInversiones [2,1,4,3]  ==  2
--    numeroInversiones [4,3,1,2]  ==  5
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Numero_de_inversiones where

import Test.QuickCheck (quickCheck)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Data.Array ((!), listArray)

-- 1ª solución
-- ===========

numeroInversiones1 :: Ord a => [a] -> Int
numeroInversiones1 = length . indicesInversiones

-- (indicesInversiones xs) es la lista de los índices de las inversiones
-- de xs. Por ejemplo,
```

```haskell
--    indicesInversiones [2,1,4,3]  ==  [(0,1),(2,3)]
--    indicesInversiones [4,3,1,2]  ==  [(0,1),(0,2),(0,3),(1,2),(1,3)]
indicesInversiones :: Ord a => [a] -> [(Int,Int)]
indicesInversiones xs = [(i,j) | i <- [0..n-2],
                                 j <- [i+1..n-1],
                                 xs!!i > xs!!j]
  where n = length xs

-- 2ª solución
-- ===========

numeroInversiones2 :: Ord a => [a] -> Int
numeroInversiones2 = length . indicesInversiones2

indicesInversiones2 :: Ord a => [a] -> [(Int,Int)]
indicesInversiones2 xs = [(i,j) | i <- [0..n-2],
                                  j <- [i+1..n-1],
                                  v!i > v!j]
  where n = length xs
        v = listArray (0,n-1) xs

-- 3ª solución
-- ===========

numeroInversiones3 :: Ord a => [a] -> Int
numeroInversiones3 = length . inversiones

-- (inversiones xs) es la lista de las inversiones  de xs. Por ejemplo,
--    Inversiones [2,1,4,3]  ==  [(2,1),(4,3)]
--    Inversiones [4,3,1,2]  ==  [(4,3),(4,1),(4,2),(3,1),(3,2)]
inversiones :: Ord a => [a] -> [(a,a)]
inversiones []     = []
inversiones (x:xs) = [(x,y) | y <- xs, y < x] ++ inversiones xs

-- 4ª solución
-- ===========

numeroInversiones4 :: Ord a => [a] -> Int
numeroInversiones4 []     = 0
numeroInversiones4 (x:xs) = length (filter (x>) xs) + numeroInversiones4 xs

-- 5ª solución
-- ===========

numeroInversiones5 :: Ord a => [a] -> Int
```

```haskell
numeroInversiones5 xs = snd (ordenadaConInversiones xs)

ordenadaConInversiones :: Ord a => [a] -> ([a], Int)
ordenadaConInversiones []  = ([], 0)
ordenadaConInversiones [x] = ([x], 0)
ordenadaConInversiones xs  =
  (mezcla, izqInversiones + dchaInversiones + mezclaInversiones)
  where
  (izq, dcha) = splitAt (length xs `div` 2) xs
  (izqOrdenada, izqInversiones)  = ordenadaConInversiones izq
  (dchaOrdenada, dchaInversiones) = ordenadaConInversiones dcha
  (mezcla, mezclaInversiones) = mezclaYcuenta izqOrdenada dchaOrdenada

mezclaYcuenta :: Ord a => [a] -> [a] -> ([a], Int)
mezclaYcuenta [] ys = (ys, 0)
mezclaYcuenta xs [] = (xs, 0)
mezclaYcuenta (x:xs) (y:ys)
  | x <= y = let (zs, n) = mezclaYcuenta xs (y:ys) in
      (x : zs, n)
  | otherwise = let (zs, n) = mezclaYcuenta (x:xs) ys in
      (y : zs, length xs + n + 1)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> Int) -> Spec
specG numeroInversiones = do
  it "e1" $
    numeroInversiones [2,1,4,3]  `shouldBe`  2
  it "e2" $
    numeroInversiones [4,3,1,2]  `shouldBe`  5

spec :: Spec
spec = do
  describe "def. 1" $ specG numeroInversiones1
  describe "def. 2" $ specG numeroInversiones2
  describe "def. 3" $ specG numeroInversiones3
  describe "def. 4" $ specG numeroInversiones4
  describe "def. 5" $ specG numeroInversiones5

-- La verificación es
--    λ> verifica
```

```
--    10 examples, 0 failures

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

-- La propiedad es
prop_numeroInversiones :: [Int] -> Bool
prop_numeroInversiones xs =
  all (== numeroInversiones1 xs)
      [numeroInversiones2 xs,
       numeroInversiones3 xs,
       numeroInversiones4 xs,
       numeroInversiones5 xs]

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

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

-- La comparación es
--    λ> numeroInversiones1 [1200,1199..1]
--    719400
--    (2.15 secs, 260,102,328 bytes)
--    λ> numeroInversiones2 [1200,1199..1]
--    719400
--    (0.36 secs, 294,624,048 bytes)
--    λ> numeroInversiones3 [1200,1199..1]
--    719400
--    (0.21 secs, 150,647,848 bytes)
--    λ> numeroInversiones4 [1200,1199..1]
--    719400
--    (0.06 secs, 41,504,368 bytes)
--    λ> numeroInversiones5 [1200,1199..1]
--    719400
--    (0.06 secs, 6,825,296 bytes)
--    λ> numeroInversiones3 [3000,2999..1]
--    4498500
--    (1.03 secs, 937,320,624 bytes)
--    λ> numeroInversiones4 [3000,2999..1]
--    4498500
--    (0.40 secs, 254,111,416 bytes)
--    λ> numeroInversiones5 [3000,2999..1]
--    4498500
```

```
--    (0.09 secs, 17,593,416 bytes)
```

Ejercicio 36

Separación por posición

```
-- ---------------------------------------------------------------------
-- Definir la función
--    particion :: [a] -> ([a],[a])
-- tal que (particion xs) es el par cuya primera componente son los
-- elementos de xs en posiciones pares y su segunda componente son los
-- restantes elementos. Por ejemplo,
--    particion [3,5,6,2]    ==  ([3,6],[5,2])
--    particion [3,5,6,2,7]  ==  ([3,6,7],[5,2])
--    particion "particion"  ==  ("priin","atco")
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Separacion_por_posicion where

import Data.List (partition)
import Control.Arrow ((***))
import qualified Data.Vector as V ((!), fromList, length)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

particion1 :: [a] -> ([a],[a])
particion1 xs = ([x | (n,x) <- nxs, even n],
                 [x | (n,x) <- nxs, odd n])
  where nxs = enumeracion xs

-- (enumeracion xs) es la enumeración de xs. Por ejemplo,
--    enumeracion [7,9,6,8]  ==  [(0,7),(1,9),(2,6),(3,8)]
enumeracion :: [a] -> [(Int,a)]
```

```haskell
enumeracion = zip [0..]

-- 2ª solución
-- ===========

particion2 :: [a] -> ([a],[a])
particion2 []     = ([],[])
particion2 (x:xs) = (x:zs,ys)
  where (ys,zs) = particion2 xs

-- 3ª solución
-- ===========

particion3 :: [a] -> ([a],[a])
particion3 = foldr f ([],[])
  where f x (ys,zs) = (x:zs,ys)

-- 4ª solución
-- ===========

particion4 :: [a] -> ([a],[a])
particion4 = foldr (\x (ys,zs) -> (x:zs,ys)) ([],[])

-- 5ª solución
-- ===========

particion5 :: [a] -> ([a],[a])
particion5 xs =
  ([xs!!k | k <- [0,2..n]],
   [xs!!k | k <- [1,3..n]])
  where n = length xs - 1

-- 6ª solución
-- ===========

particion6 :: [a] -> ([a],[a])
particion6 xs = (pares xs, impares xs)

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

```haskell
-- (impares xs) es la lista de los elementos de xs en posiciones
-- impares. Por ejemplo,
--    impares [3,5,6,2]  ==  [5,2]
impares :: [a] -> [a]
impares []     = []
impares (_:xs) = pares xs

-- 7ª solución
-- ===========

particion7 :: [a] -> ([a],[a])
particion7 [] = ([],[])
particion7 xs =
  ([v V.! k | k <- [0,2..n-1]],
   [v V.! k | k <- [1,3..n-1]])
  where v = V.fromList xs
        n = V.length v

-- 8ª solución
-- ===========

particion8 :: [a] -> ([a],[a])
particion8 xs =
  (map snd ys, map snd zs)
  where (ys,zs) = partition posicionPar (zip [0..] xs)

posicionPar :: (Int,a) -> Bool
posicionPar = even . fst

-- 9ª solución
-- ===========

particion9 :: [a] -> ([a], [a])
particion9 xs =
  ([x | (x, b) <- zip xs (cycle [True, False]), b]
  ,[x | (x, b) <- zip xs (cycle [True, False]), not b])

-- 10ª solución
-- ============

particion10 :: [a] -> ([a], [a])
particion10 = (map snd *** map snd) . partition (even . fst) . zip [0..]

-- Verificación
-- ============
```

```haskell
verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> ([Int], [Int])) -> Spec
specG particion = do
  it "e1" $
    particion [3,5,6,2]    `shouldBe`  ([3,6],[5,2])
  it "e2" $
    particion [3,5,6,2,7]  `shouldBe`  ([3,6,7],[5,2])

spec :: Spec
spec = do
  describe "def. 1"  $ specG particion1
  describe "def. 2"  $ specG particion2
  describe "def. 3"  $ specG particion3
  describe "def. 4"  $ specG particion4
  describe "def. 5"  $ specG particion5
  describe "def. 6"  $ specG particion6
  describe "def. 7"  $ specG particion7
  describe "def. 8"  $ specG particion8
  describe "def. 9"  $ specG particion9
  describe "def. 10" $ specG particion10

-- La verificación es
--    λ> verifica
--    20 examples, 0 failures

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

-- La propiedad es
prop_particion :: [Int] -> Bool
prop_particion xs =
  all (== particion1 xs)
      [particion2 xs,
       particion3 xs,
       particion4 xs,
       particion5 xs,
       particion6 xs,
       particion7 xs,
       particion8 xs,
       particion9 xs,
       particion10 xs]
```

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

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

-- La comparación es
--    λ> last (snd (particion1 [1..6*10^6]))
--    6000000
--    (2.74 secs, 2,184,516,080 bytes)
--    λ> last (snd (particion2 [1..6*10^6]))
--    6000000
--    (2.02 secs, 1,992,515,880 bytes)
--    λ> last (snd (particion3 [1..6*10^6]))
--    6000000
--    (3.17 secs, 1,767,423,240 bytes)
--    λ> last (snd (particion4 [1..6*10^6]))
--    6000000
--    (3.23 secs, 1,767,423,240 bytes)
--    λ> last (snd (particion5 [1..6*10^6]))
--    6000000
--    (1.62 secs, 1,032,516,192 bytes)
--    λ> last (snd (particion5 [1..6*10^6]))
--    6000000
--    (1.33 secs, 1,032,516,192 bytes)
--    λ> last (snd (particion6 [1..6*10^6]))
--    6000000
--    (1.80 secs, 888,515,960 bytes)
--    λ> last (snd (particion7 [1..6*10^6]))
--    6000000
--    (1.29 secs, 1,166,865,672 bytes)
--    λ> last (snd (particion8 [1..6*10^6]))
--    6000000
--    (0.87 secs, 3,384,516,616 bytes)
--    λ> last (snd (particion9 [1..6*10^6]))
--    6000000
--    (1.68 secs, 1,368,602,104 bytes)
--    λ> last (snd (particion10 [1..6*10^6]))
--    6000000
--    (1.83 secs, 3,192,595,776 bytes)
--
--    λ> last (snd (particion5 [1..10^7]))
--    10000000
--    (1.94 secs, 1,720,516,872 bytes)
```

```
--    λ> last (snd (particion7 [1..10^7]))
--    10000000
--    (2.54 secs, 1,989,215,176 bytes)
--    λ> last (snd (particion8 [1..10^7]))
--    10000000
--    (1.33 secs, 5,640,516,960 bytes)
--    λ> last (snd (particion9 [1..10^7]))
--    10000000
--    (2.66 secs, 2,280,602,872 bytes)
--    λ> last (snd (particion10 [1..10^7]))
--    10000000
--    (1.77 secs, 4,888,602,592 bytes)
```

Ejercicio 37

Emparejamiento de árboles

```
-- ---------------------------------------------------------------------
-- Los árboles se pueden representar mediante el siguiente tipo de datos
--    data Arbol a = N a [Arbol a]
--      deriving (Show, Eq)
-- Por ejemplo, los árboles
--      1               3
--     / \             /|\
--    6   3           / | \
--        |          5  4  7
--        5          |     /\
--                   6    2  1
-- se representan por
--    ej1, ej2 :: Arbol Int
--    ej1 = N 1 [N 6 [],N 3 [N 5 []]]
--    ej2 = N 3 [N 5 [N 6 []], N 4 [], N 7 [N 2 [], N 1 []]]
--
-- Definir la función
--    emparejaArboles :: (a -> b -> c) -> Arbol a -> Arbol b -> Arbol c
-- tal que (emparejaArboles f a1 a2) es el árbol obtenido aplicando la
-- función f a los elementos de los árboles a1 y a2 que se encuentran en
-- la misma posición. Por ejemplo,
--    λ> emparejaArboles (+) (N 1 [N 2 [], N 3[]]) (N 1 [N 6 []])
--    N 2 [N 8 []]
--    λ> emparejaArboles (+) ej1 ej2
--    N 4 [N 11 [],N 7 []]
--    λ> emparejaArboles (+) ej1 ej1
--    N 2 [N 12 [],N 6 [N 10 []]]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Emparejamiento_de_arboles where
```

```haskell
import Data.Tree (Tree (..))
import Control.Monad.Zip (mzipWith)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (Arbitrary, Gen,
                        arbitrary, generate, sublistOf, sized, quickCheck)

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

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

-- 1ª solución
-- ===========

emparejaArboles1 :: (a -> b -> c) -> Arbol a -> Arbol b -> Arbol c
emparejaArboles1 f (N x xs) (N y ys) =
  N (f x y) (emparejaListaArboles f xs ys)

emparejaListaArboles :: (a -> b -> c) -> [Arbol a] -> [Arbol b] -> [Arbol c]
emparejaListaArboles _ [] _ = []
emparejaListaArboles _ _ [] = []
emparejaListaArboles f (x:xs) (y:ys) =
  emparejaArboles1 f x y : emparejaListaArboles f xs ys

-- 2ª solución
-- ===========

emparejaArboles2 :: (a -> b -> c) -> Arbol a -> Arbol b -> Arbol c
emparejaArboles2 f (N x xs) (N y ys) =
  N (f x y) (zipWith (emparejaArboles2 f) xs ys)

-- 3ª solución
-- ===========

emparejaArboles3 :: (a -> b -> c) -> Arbol a -> Arbol b -> Arbol c
emparejaArboles3 f x y =
  treeAarbol (mzipWith f (arbolAtree x) (arbolAtree y))

arbolAtree :: Arbol a -> Tree a
arbolAtree (N x xs) = Node x (map arbolAtree xs)

treeAarbol :: Tree a -> Arbol a
```

```haskell
treeAarbol (Node x xs) = N x (map treeAarbol xs)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ((Int -> Int -> Int) -> Arbol Int -> Arbol Int -> Arbol Int) -> Spec
specG emparejaArboles = do
  it "e1" $
    show (emparejaArboles (+) (N 1 [N 2 [], N 3[]]) (N 1 [N 6 []]))
      `shouldBe` "N 2 [N 8 []]"
  it "e2" $
    show (emparejaArboles (+) ej1 ej2)
      `shouldBe` "N 4 [N 11 [],N 7 []]"
  it "e3" $
    show (emparejaArboles (+) ej1 ej1)
      `shouldBe` "N 2 [N 12 [],N 6 [N 10 []]]"

spec :: Spec
spec = do
  describe "def. 1"  $ specG emparejaArboles1
  describe "def. 2"  $ specG emparejaArboles2
  describe "def. 3"  $ specG emparejaArboles3

-- La verificación es
--    λ> verifica
--    9 examples, 0 failures

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

-- (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
--    λ> generate (arbolArbitrario 5 :: Gen (Arbol Int))
--    N (-26) [N 8 [N 6 [N 11 []]],N 7 []]
--    λ> generate (arbolArbitrario 5 :: Gen (Arbol Int))
--    N 1 []
--    λ> generate (arbolArbitrario 5 :: Gen (Arbol Int))
--    N (-19) [N (-11) [],N 25 [],N 19 [N (-27) [],N (-19) [N 17 []]]]
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario n = do
  x  <- arbitrary
  ms <- sublistOf [0 .. n `div` 2]
  as <- mapM arbolArbitrario ms
```

```
  return (N x as)

-- Arbol es una subclase de Arbitraria
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_emparejaArboles :: Arbol Int -> Arbol Int -> Bool
prop_emparejaArboles x y =
  emparejaArboles1 (+) x y == emparejaArboles2 (+) x y &&
  emparejaArboles1 (*) x y == emparejaArboles2 (*) x y &&
  emparejaArboles1 (+) x y == emparejaArboles3 (+) x y &&
  emparejaArboles1 (*) x y == emparejaArboles3 (*) x y

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

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

-- La comparación es
--    λ> a500 <- generate (arbolArbitrario 500 :: Gen (Arbol Int))
--    λ> emparejaArboles1 (+) a500 a500 == emparejaArboles1 (+) a500 a500
--    True
--    (3.03 secs, 1,981,353,912 bytes)
--    λ> emparejaArboles2 (+) a500 a500 == emparejaArboles1 (+) a500 a500
--    True
--    (2.12 secs, 1,325,826,688 bytes)
--    λ> emparejaArboles3 (+) a500 a500 == emparejaArboles1 (+) a500 a500
--    True
--    (2.57 secs, 1,937,547,296 bytes)
```

Ejercicio 38

Eliminación de las ocurrencias aisladas

```
-- ----------------------------------------------------------------------
-- Definir la función
--    eliminaAisladas :: Eq a => a -> [a] -> [a]
-- tal que (eliminaAisladas x ys) es la lista obtenida eliminando de ys
-- las ocurrencias aisladas de x (es decir, aquellas ocurrencias de x
-- tales que su elemento anterior y posterior son distintos de x). Por
-- ejemplo,
--    eliminaAisladas 'X' ""                  == ""
--    eliminaAisladas 'X' "X"                 == ""
--    eliminaAisladas 'X' "XX"                == "XX"
--    eliminaAisladas 'X' "XXX"               == "XXX"
--    eliminaAisladas 'X' "abcd"              == "abcd"
--    eliminaAisladas 'X' "Xabcd"             == "abcd"
--    eliminaAisladas 'X' "XXabcd"            == "XXabcd"
--    eliminaAisladas 'X' "XXXabcd"           == "XXXabcd"
--    eliminaAisladas 'X' "abcdX"             == "abcd"
--    eliminaAisladas 'X' "abcdXX"            == "abcdXX"
--    eliminaAisladas 'X' "abcdXXX"           == "abcdXXX"
--    eliminaAisladas 'X' "abXcd"             == "abcd"
--    eliminaAisladas 'X' "abXXcd"            == "abXXcd"
--    eliminaAisladas 'X' "abXXXcd"           == "abXXXcd"
--    eliminaAisladas 'X' "XabXcdX"           == "abcd"
--    eliminaAisladas 'X' "XXabXXcdXX"        == "XXabXXcdXX"
--    eliminaAisladas 'X' "XXXabXXXcdXXX"     == "XXXabXXXcdXXX"
--    eliminaAisladas 'X' "XabXXcdXeXXXfXx"   == "abXXcdeXXXfx"
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```haskell
module Elimina_aisladas where

import Data.List (group)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

eliminaAisladas1 :: Eq a => a -> [a] -> [a]
eliminaAisladas1 _ [] = []
eliminaAisladas1 x [y]
  | x == y    = []
  | otherwise = [y]
eliminaAisladas1 x (y1:y2:ys)
  | y1 /= x   = y1 : eliminaAisladas1 x (y2:ys)
  | y2 /= x   = y2 : eliminaAisladas1 x ys
  | otherwise = takeWhile (==x) (y1:y2:ys) ++
                eliminaAisladas1 x (dropWhile (==x) ys)

-- 2ª solución
-- ===========

eliminaAisladas2 :: Eq a => a -> [a] -> [a]
eliminaAisladas2 _ [] = []
eliminaAisladas2 x ys
  | cs == [x] = as ++ eliminaAisladas2 x ds
  | otherwise = as ++ cs ++ eliminaAisladas2 x ds
  where (as,bs) = span (/=x) ys
        (cs,ds) = span (==x) bs

-- 3ª solución
-- ===========

eliminaAisladas3 :: Eq a => a -> [a] -> [a]
eliminaAisladas3 x ys =
  concat [zs | zs <- group ys, zs /= [x]]

-- 4ª solución
-- ===========

eliminaAisladas4 :: Eq a => a -> [a] -> [a]
eliminaAisladas4 x =
  concat . filter (/= [x]) . group
```

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

verifica :: IO ()
verifica = hspec spec

specG :: (Char -> String -> String) -> Spec
specG eliminaAisladas = do
  it "e1" $
    eliminaAisladas 'X' ""                 `shouldBe` ""
  it "e2" $
    eliminaAisladas 'X' "X"                `shouldBe` ""
  it "e3" $
    eliminaAisladas 'X' "XX"               `shouldBe` "XX"
  it "e4" $
    eliminaAisladas 'X' "XXX"              `shouldBe` "XXX"
  it "e5" $
    eliminaAisladas 'X' "abcd"             `shouldBe` "abcd"
  it "e6" $
    eliminaAisladas 'X' "Xabcd"            `shouldBe` "abcd"
  it "e7" $
    eliminaAisladas 'X' "XXabcd"           `shouldBe` "XXabcd"
  it "e8" $
    eliminaAisladas 'X' "XXXabcd"          `shouldBe` "XXXabcd"
  it "e9" $
    eliminaAisladas 'X' "abcdX"            `shouldBe` "abcd"
  it "e10" $
    eliminaAisladas 'X' "abcdXX"           `shouldBe` "abcdXX"
  it "e11" $
    eliminaAisladas 'X' "abcdXXX"          `shouldBe` "abcdXXX"
  it "e12" $
    eliminaAisladas 'X' "abXcd"            `shouldBe` "abcd"
  it "e13" $
    eliminaAisladas 'X' "abXXcd"           `shouldBe` "abXXcd"
  it "e14" $
    eliminaAisladas 'X' "abXXXcd"          `shouldBe` "abXXXcd"
  it "e15" $
    eliminaAisladas 'X' "XabXcdX"          `shouldBe` "abcd"
  it "e16" $
    eliminaAisladas 'X' "XXabXXcdXX"       `shouldBe` "XXabXXcdXX"
  it "e17" $
    eliminaAisladas 'X' "XXXabXXXcdXXX"    `shouldBe` "XXXabXXXcdXXX"
  it "e18" $
    eliminaAisladas 'X' "XabXXcdXeXXXfXx" `shouldBe` "abXXcdeXXXfx"
```

```haskell
spec :: Spec
spec = do
  describe "def. 1"  $ specG eliminaAisladas1
  describe "def. 2"  $ specG eliminaAisladas2
  describe "def. 3"  $ specG eliminaAisladas3
  describe "def. 4"  $ specG eliminaAisladas4

-- La verificación es
--    λ> verifica
--    72 examples, 0 failures

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

-- La propiedad es
prop_eliminaAisladas :: Int -> [Int] -> Bool
prop_eliminaAisladas x ys =
  all (== eliminaAisladas1 x ys)
      [eliminaAisladas2 x ys,
       eliminaAisladas3 x ys,
       eliminaAisladas4 x ys]

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

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

-- La comparación es
--    λ> length (eliminaAisladas1 'a' (take (5*10^6) (cycle "abca")))
--    4999998
--    (3.86 secs, 2,030,515,400 bytes)
--    λ> length (eliminaAisladas2 'a' (take (5*10^6) (cycle "abca")))
--    4999998
--    (3.41 secs, 2,210,516,832 bytes)
--    λ> length (eliminaAisladas3 'a' (take (5*10^6) (cycle "abca")))
--    4999998
--    (2.11 secs, 2,280,516,448 bytes)
--    λ> length (eliminaAisladas4 'a' (take (5*10^6) (cycle "abca")))
--    4999998
--    (0.92 secs, 1,920,516,704 bytes)
```

Ejercicio 39

Ordenada cíclicamente

```haskell
-- ----------------------------------------------------------------------
-- Se dice que una sucesión x(1), ..., x(n) está ordenada cíclicamente
-- si existe un índice i tal que la sucesión
--    x(i), x(i+1), ..., x(n), x(1), ..., x(i-1)
-- está ordenada creciente de forma estricta.
--
-- Definir la función
--    ordenadaCiclicamente :: Ord a => [a] -> Maybe Int
-- tal que (ordenadaCiclicamente xs) es el índice a partir del cual está
-- ordenada, si la lista está ordenado cíclicamente y Nothing en caso
-- contrario. Por ejemplo,
--    ordenadaCiclicamente [1,2,3,4]      ==  Just 0
--    ordenadaCiclicamente [5,8,1,3]      ==  Just 2
--    ordenadaCiclicamente [4,6,7,5,1,3]  ==  Nothing
--    ordenadaCiclicamente [1,0,3,2]      ==  Nothing
--    ordenadaCiclicamente [1,2,0]        ==  Just 2
--    ordenadaCiclicamente "cdeab"        ==  Just 3
--
-- Nota: Se supone que el argumento es una lista no vacía sin elementos
-- repetidos.
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Ordenada_ciclicamente where

import Data.List       (nub, sort)
import Data.Maybe      (isJust, listToMaybe)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (Arbitrary, Gen, NonEmptyList (NonEmpty), Property,
                        arbitrary, chooseInt, collect, quickCheck)
```

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

ordenadaCiclicamente1 :: Ord a => [a] -> Maybe Int
ordenadaCiclicamente1 xs = aux 0 xs
  where n = length xs
        aux i zs
          | i == n      = Nothing
          | ordenada zs = Just i
          | otherwise   = aux (i+1) (siguienteCiclo zs)

-- (ordenada xs) se verifica si la lista xs está ordenada
-- crecientemente. Por ejemplo,
--   ordenada "acd"   ==  True
--   ordenada "acdb"  ==  False
ordenada :: Ord a => [a] -> Bool
ordenada []     = True
ordenada (x:xs) = all (x <) xs && ordenada xs

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

-- 2ª solución
-- ===========

ordenadaCiclicamente2 :: Ord a => [a] -> Maybe Int
ordenadaCiclicamente2 xs =
  listToMaybe [n | n <- [0..length xs-1],
                   ordenada (drop n xs ++ take n xs)]

-- 3ª solución
-- ===========

ordenadaCiclicamente3 :: Ord a => [a] -> Maybe Int
ordenadaCiclicamente3 xs
  | ordenada (bs ++ as) = Just k
  | otherwise           = Nothing
  where (_,k)   = minimum (zip xs [0..])
        (as,bs) = splitAt k xs

-- Verificación
```

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

verifica :: IO ()
verifica = hspec spec

specG :: ([Int] -> Maybe Int) -> Spec
specG ordenadaCiclicamente = do
  it "e1" $
    ordenadaCiclicamente [1,2,3,4]      `shouldBe`  Just 0
  it "e2" $
    ordenadaCiclicamente [5,8,1,3]      `shouldBe`  Just 2
  it "e3" $
    ordenadaCiclicamente [4,6,7,5,1,3]  `shouldBe`  Nothing
  it "e4" $
    ordenadaCiclicamente [1,0,3,2]      `shouldBe`  Nothing
  it "e5" $
    ordenadaCiclicamente [1,2,0]        `shouldBe`  Just 2

spec :: Spec
spec = do
  describe "def. 1"  $ specG ordenadaCiclicamente1
  describe "def. 2"  $ specG ordenadaCiclicamente2
  describe "def. 3"  $ specG ordenadaCiclicamente3

-- La verificación es
--    λ> verifica
--    15 examples, 0 failures

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

-- La propiedad es
prop_ordenadaCiclicamente1 :: NonEmptyList Int -> Bool
prop_ordenadaCiclicamente1 (NonEmpty xs) =
  ordenadaCiclicamente1 xs == ordenadaCiclicamente2 xs

-- La comprobación es
--    λ> quickCheck prop_ordenadaCiclicamente1
--    +++ OK, passed 100 tests.

-- La propiedad para analizar los casos de prueba
prop_ordenadaCiclicamente2 :: NonEmptyList Int -> Property
prop_ordenadaCiclicamente2 (NonEmpty xs) =
  collect (isJust (ordenadaCiclicamente1 xs)) $
  ordenadaCiclicamente1 xs == ordenadaCiclicamente2 xs
```

```haskell
-- El análisis es
--    λ> quickCheck prop_ordenadaCiclicamente2
--    +++ OK, passed 100 tests:
--    89% False
--    11% True

-- Tipo para generar listas
newtype Lista = L [Int]
  deriving Show

-- Generador de listas.
listaArbitraria :: Gen Lista
listaArbitraria = do
  x <- arbitrary
  xs <- arbitrary
  let ys = x : xs
  k <- chooseInt (0, length ys)
  let (as,bs) = splitAt k (sort (nub ys))
  return (L (bs ++ as))

-- Lista es una subclase de Arbitrary.
instance Arbitrary Lista where
  arbitrary = listaArbitraria

-- La propiedad para analizar los casos de prueba
prop_ordenadaCiclicamente3 :: Lista -> Property
prop_ordenadaCiclicamente3 (L xs) =
  collect (isJust (ordenadaCiclicamente1 xs)) $
  ordenadaCiclicamente1 xs == ordenadaCiclicamente2 xs

-- El análisis es
--    λ> quickCheck prop_ordenadaCiclicamente3
--    +++ OK, passed 100 tests (100% True).

-- Tipo para generar
newtype Lista2 = L2 [Int]
  deriving Show

-- Generador de listas
listaArbitraria2 :: Gen Lista2
listaArbitraria2 = do
  x' <- arbitrary
  xs <- arbitrary
  let ys = x' : xs
```

```
  k <- chooseInt (0, length ys)
  let (as,bs) = splitAt k (sort (nub ys))
  n <- chooseInt (0,1)
  return (if even n
          then L2 (bs ++ as)
          else L2 ys)

-- Lista es una subclase de Arbitrary.
instance Arbitrary Lista2 where
  arbitrary = listaArbitraria2

-- La propiedad para analizar los casos de prueba
prop_ordenadaCiclicamente4 :: Lista2 -> Property
prop_ordenadaCiclicamente4 (L2 xs) =
  collect (isJust (ordenadaCiclicamente1 xs)) $
  ordenadaCiclicamente1 xs == ordenadaCiclicamente2 xs

-- El análisis es
--    λ> quickCheck prop_ordenadaCiclicamente4
--    +++ OK, passed 100 tests:
--    51% True
--    49% False

-- La propiedad es
prop_ordenadaCiclicamente :: Lista2 -> Bool
prop_ordenadaCiclicamente (L2 xs) =
  all (== ordenadaCiclicamente1 xs)
      [ordenadaCiclicamente2 xs,
       ordenadaCiclicamente3 xs]

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

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

-- La comparación es
--    λ> ordenadaCiclicamente1 ([100..4000] ++ [1..99])
--    Just 3901
--    (3.27 secs, 2,138,864,568 bytes)
--    λ> ordenadaCiclicamente2 ([100..4000] ++ [1..99])
--    Just 3901
--    (2.44 secs, 1,430,040,008 bytes)
--    λ> ordenadaCiclicamente3 ([100..4000] ++ [1..99])
```

```
--    Just 3901
--    (1.18 secs, 515,549,200 bytes)
```

Ejercicio 40

Órbita prima

```haskell
-- ----------------------------------------------------------------------
-- La órbita prima de un número n es la sucesión construida de la
-- siguiente forma:
-- + si n es compuesto su órbita no tiene elementos
-- + si n es primo, entonces n está en su órbita; además, sumamos n y
--   sus dígitos, si el resultado es un número primo repetimos el
--   proceso hasta obtener un número compuesto.
--
-- Por ejemplo, con el 11 podemos repetir el proceso dos veces
--    13 = 11+1+1
--    17 = 13+1+3
--    25 = 17+1+7
-- Así, la órbita prima de 11 es 11, 13, 17.
--
-- Definir la función
--    orbita :: Integer -> [Integer]
-- tal que (orbita n) es la órbita prima de n. Por ejemplo,
--    orbita 11 == [11,13,17]
--    orbita 59 == [59,73,83]
--
-- Calcular el menor número cuya órbita prima tiene más de 3 elementos.
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Orbita_prima where

import Data.Numbers.Primes (isPrime)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
```

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

orbita1 :: Integer -> [Integer]
orbita1 n | not (esPrimo n) = []
          | otherwise       = n : orbita1 (n + sum (digitos n))

-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo 7   ==  True
--    esPrimo 15  ==  False
esPrimo :: Integer -> Bool
esPrimo n = [x | x <- [1..n], n `rem` x == 0] == [1,n]

-- (digitos n) es la lista de los dígitos de n. Por ejemplo,
--    digitos 15  ==  [1,5]
digitos :: Integer -> [Integer]
digitos n = [read [x]| x <- show n]

-- 2ª solución
-- ===========

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

-- 3ª solución
-- ===========

orbita3 :: Integer -> [Integer]
orbita3 n = takeWhile isPrime (iterate f n)
  where f x = x + sum (digitos x)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Integer -> [Integer]) -> Spec
specG orbita = do
  it "e1" $
    orbita 11 `shouldBe` [11,13,17]
  it "e2" $
    orbita 59 `shouldBe` [59,73,83]

spec :: Spec
```

```
spec = do
  describe "def. 1"  $ specG orbita1
  describe "def. 2"  $ specG orbita2
  describe "def. 3"  $ specG orbita3

-- La verificación es
--    λ> verifica
--    6 examples, 0 failures

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

-- La propiedad es
prop_orbita :: Integer -> Bool
prop_orbita n =
  all (== orbita1 n)
      [orbita2 n,
       orbita3 n]

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

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

-- La comparación es
--    λ> orbita1 516493
--    [516493,516521,516541,516563,516589,516623]
--    (1.12 secs, 620,631,504 bytes)
--    λ> orbita2 516493
--    [516493,516521,516541,516563,516589,516623]
--    (1.08 secs, 620,631,112 bytes)
--    λ> orbita3 516493
--    [516493,516521,516541,516563,516589,516623]
--    (0.01 secs, 2,340,960 bytes)

-- Cálculo
-- =======

-- El cálculo es
--    λ> head [x | x <- [1,3..], length (orbita3 x) > 3]
--    277
--
--    λ> orbita3 277
```

```
--    [277,293,307,317]
```

Ejercicio 41

Divisores de un número con final dado

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    divisoresConFinal :: Integer -> Integer -> [Integer]
-- tal que (divisoresConFinal n m) es la lista de los divisores de n
-- cuyos dígitos finales coincide con m. Por ejemplo,
--    divisoresConFinal 84 4    ==  [4,14,84]
--    divisoresConFinal 720 20  ==  [20,120,720]
-- ---------------------------------------------------------------------

module Divisores_con_final where

import Data.List (group, inits, isSuffixOf, nub, sort, subsequences)
import Data.Numbers.Primes (primeFactors)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

divisoresConFinal1 :: Integer -> Integer -> [Integer]
divisoresConFinal1 n m =
  [x | x <- divisores1 n, final1 x m]

-- (divisores n) es el conjunto de divisores de n. Por ejemplo,
--   divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- (final x y) se verifica si el final de x es igual a y. Por ejemplo,
```

```haskell
--    final 325 5   ==  True
--    final 325 25  ==  True
--    final 325 35  ==  False
final1 :: Integer -> Integer -> Bool
final1 x y = take n xs == ys
  where xs = reverse (show x)
        ys = reverse (show y)
        n  = length ys

-- 2ª solución
-- ===========

divisoresConFinal2 :: Integer -> Integer -> [Integer]
divisoresConFinal2 n m =
  [x | x <- divisores2 n, final2 x m]

divisores2 :: Integer -> [Integer]
divisores2 n = filter ((== 0) . mod n) [1..n]

final2 :: Integer -> Integer -> Bool
final2 x y = show y `isSuffixOf` show x

-- 3ª solución
-- ===========

divisoresConFinal3 :: Integer -> Integer -> [Integer]
divisoresConFinal3 n m =
  [x | x <- divisores3 n, final2 x m]

divisores3 :: Integer -> [Integer]
divisores3 =
  nub . sort . map product . subsequences . primeFactors

-- 4ª solución
-- ===========

divisoresConFinal4 :: Integer -> Integer -> [Integer]
divisoresConFinal4 n m =
  [x | x <- divisores4 n, final2 x m]

divisores4 :: Integer -> [Integer]
divisores4 = 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]

-- 5ª solución
-- ===========

divisoresConFinal5 :: Integer -> Integer -> [Integer]
divisoresConFinal5 n m =
  [x | x <- divisores5 n, final2 x m]

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

-- 6ª solución
-- ===========

divisoresConFinal6 :: Integer -> Integer -> [Integer]
divisoresConFinal6 n m =
  [x | x <- divisores6 n, final2 x m]

divisores6 :: Integer -> [Integer]
divisores6 = sort
           . map (product . concat)
           . mapM inits
           . group
           . primeFactors

-- Verificación
-- ============

verifica :: IO ()
```

```
verifica = hspec spec

specG :: (Integer -> Integer -> [Integer]) -> Spec
specG divisoresConFinal = do
  it "e1" $
    divisoresConFinal 84 4    `shouldBe`  [4,14,84]
  it "e2" $
    divisoresConFinal 720 20  `shouldBe`  [20,120,720]

spec :: Spec
spec = do
  describe "def. 1" $ specG divisoresConFinal1
  describe "def. 2" $ specG divisoresConFinal2
  describe "def. 3" $ specG divisoresConFinal3
  describe "def. 4" $ specG divisoresConFinal4
  describe "def. 5" $ specG divisoresConFinal5
  describe "def. 6" $ specG divisoresConFinal6

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- La propiedad es
prop_divisoresConFinal :: Positive Integer -> Positive Integer -> Bool
prop_divisoresConFinal (Positive n) (Positive m) =
  all (== divisoresConFinal1 n m)
      [ divisoresConFinal2 n m,
        divisoresConFinal3 n m,
        divisoresConFinal4 n m,
        divisoresConFinal5 n m,
        divisoresConFinal6 n m ]

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

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

-- La comparación es
--    λ> divisoresConFinal1 (product [1..11]) 6800
--    [16800,226800,316800,39916800]
```

```
--    (13.89 secs, 7,984,560,800 bytes)
--    λ> divisoresConFinal2 (product [1..11]) 6800
--    [16800,226800,316800,39916800]
--    (4.84 secs, 4,790,920,688 bytes)
--    λ> divisoresConFinal3 (product [1..11]) 6800
--    [16800,226800,316800,39916800]
--    (0.07 secs, 87,137,992 bytes)
--    λ> divisoresConFinal4 (product [1..11]) 6800
--    [16800,226800,316800,39916800]
--    (0.02 secs, 2,324,528 bytes)
--    λ> divisoresConFinal5 (product [1..11]) 6800
--    [16800,226800,316800,39916800]
--    (0.00 secs, 1,801,872 bytes)
--    λ> divisoresConFinal6 (product [1..11]) 6800
--    [16800,226800,316800,39916800]
--    (0.01 secs, 1,801,536 bytes)
--
--    λ> divisoresConFinal4 (product [1..25]) 985984000000
--    [2985984000000,95096985984000000,155112100433309859840000000]
--    (1.77 secs, 2,142,500,832 bytes)
--    λ> divisoresConFinal5 (product [1..25]) 985984000000
--    [2985984000000,95096985984000000,155112100433309859840000000]
--    (1.15 secs, 1,603,330,352 bytes)
--    λ> divisoresConFinal6 (product [1..25]) 985984000000
--    [2985984000000,95096985984000000,155112100433309859840000000]
--    (1.19 secs, 1,603,329,840 bytes)
```

Ejercicio 42

Descomposiciones de x como sumas de n sumandos de una lista

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    sumas :: (Num a, Ord a) => Int -> [a] -> a -> [[a]]
-- tal que (sumas n ns x) es la lista de las descomposiciones de x como
-- sumas de n sumandos de la lista ns. Por ejemplo,
--    sumas 2 [1,2] 3     == [[1,2]]
--    sumas 2 [-1] (-2)   == [[-1,-1]]
--    sumas 2 [-1,3,-1] 2 == [[-1,3]]
--    sumas 2 [1,2] 4     == [[2,2]]
--    sumas 2 [1,2] 5     == []
--    sumas 3 [1,2] 5     == [[1,2,2]]
--    sumas 3 [1,2] 6     == [[2,2,2]]
--    sumas 2 [1,2,5] 6   == [[1,5]]
--    sumas 2 [1,2,3,5] 4 == [[1,3],[2,2]]
--    sumas 2 [1..5] 6    == [[1,5],[2,4],[3,3]]
--    sumas 3 [1..5] 7    == [[1,1,5],[1,2,4],[1,3,3],[2,2,3]]
--    sumas 3 [1..200] 4  == [[1,1,2]]
-- ---------------------------------------------------------------------

module Descomposiciones_con_n_sumandos where

import Data.List (nub, sort)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- =============
```

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

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

-- 2ª solución
-- =============

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

-- 3ª solución
-- =============

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

-- Verificación
```

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

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> [Int] -> Int -> [[Int]]) -> Spec
specG sumas = do
  it "e1" $
    sumas' 2 [1,2] 3     `shouldBe` [[1,2]]
  it "e2" $
    sumas' 2 [-1] (-2)   `shouldBe` [[-1,-1]]
  it "e3" $
    sumas' 2 [-1,3,-1] 2 `shouldBe` [[-1,3]]
  it "e4" $
    sumas' 2 [1,2] 4     `shouldBe` [[2,2]]
  it "e5" $
    sumas' 2 [1,2] 5     `shouldBe` []
  it "e6" $
    sumas' 3 [1,2] 5     `shouldBe` [[1,2,2]]
  it "e7" $
    sumas' 3 [1,2] 6     `shouldBe` [[2,2,2]]
  it "e8" $
    sumas' 2 [1,2,5] 6   `shouldBe` [[1,5]]
  it "e9" $
    sumas' 2 [1,2,3,5] 4 `shouldBe` [[1,3],[2,2]]
  it "e10" $
    sumas' 2 [1..5] 6    `shouldBe` [[1,5],[2,4],[3,3]]
  it "e11" $
    sumas' 3 [1..5] 7    `shouldBe` [[1,1,5],[1,2,4],[1,3,3],[2,2,3]]
  where sumas' n ys x = sort (map sort (sumas n ys x))

spec :: Spec
spec = do
  describe "def. 1" $ specG sumas1
  describe "def. 2" $ specG sumas2
  describe "def. 3" $ specG sumas3

-- La verificación es
--    λ> verifica
--    33 examples, 0 failures

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

prop_equiv_sumas :: Positive Int -> [Int] -> Int -> Bool
```

```haskell
prop_equiv_sumas (Positive n) ns x =
  all (== normal (sumas1 n ns x))
      [ normal (sumas2 n ns x)
      , normal (sumas3 n ns x)]
  where normal = sort . map sort

-- La verificación es
--    λ> quickCheckWith (stdArgs {maxSize=7}) prop_equiv_sumas
--    +++ OK, passed 100 tests.

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

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

Ejercicio 43

Selección hasta el primero que falla inclusive

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    seleccionConFallo :: (a -> Bool) -> [a] -> [a]
-- tal que (seleccionConFallo p xs) es la lista de los elementos de xs
-- que cumplen el predicado p hasta el primero que no lo cumple
-- inclusive. Por ejemplo,
--    seleccionConFallo (<5) [3,2,5,7,1,0]  ==  [3,2,5]
--    seleccionConFallo odd [1..4]          ==  [1,2]
--    seleccionConFallo odd [1,3,5]         ==  [1,3,5]
--    seleccionConFallo (<5) [10..20]       ==  [10]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Seleccion_con_fallo where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck.HigherOrder (quickCheck')

-- 1ª solución
-- ===========

seleccionConFallo1 :: (a -> Bool) -> [a] -> [a]
seleccionConFallo1 _ []                 = []
seleccionConFallo1 p (x:xs) | p x       = x : seleccionConFallo1 p xs
                            | otherwise = [x]

-- 2ª solución
-- ===========
```

```haskell
seleccionConFallo2 :: (a -> Bool) -> [a] -> [a]
seleccionConFallo2 p xs = ys ++ take 1 zs
  where (ys,zs) = span p xs

-- 3ª solución
-- ===========

seleccionConFallo3 :: (a -> Bool) -> [a] -> [a]
seleccionConFallo3 = ((uncurry (++) . fmap (take 1)) .) . span

-- Ejemplo de cálculo:
--    seleccionConFallo (<5) [3,2,5,7,1,0]
--    = (((uncurry (++) . fmap (take 1)) .) . span) (<5) [3,2,5,7,1,0]
--    = (uncurry (++) . fmap (take 1)) ([3,2],[5,7,1,0])
--    = uncurry (++) ([3,2],[5])
--    = [3,2,5]

-- Nota: (fmap f (x,y)) es equivalente a (x,f y).

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ((Int -> Bool) -> [Int] -> [Int]) -> Spec
specG seleccionConFallo = do
  it "e1" $
    seleccionConFallo (<5) [3,2,5,7,1,0]  `shouldBe`  [3,2,5]
  it "e2" $
    seleccionConFallo odd [1..4]          `shouldBe`  [1,2]
  it "e3" $
    seleccionConFallo odd [1,3,5]         `shouldBe`  [1,3,5]
  it "e4" $
    seleccionConFallo (<5) [10..20]       `shouldBe`  [10]

spec :: Spec
spec = do
  describe "def. 1" $ specG seleccionConFallo1
  describe "def. 2" $ specG seleccionConFallo2
  describe "def. 3" $ specG seleccionConFallo3

-- La verificación es
--    λ> verifica
```

```
--    12 examples, 0 failures

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

-- La propiedad es
prop_seleccionConFallo :: (Int -> Bool) -> [Int] -> Bool
prop_seleccionConFallo p xs =
  all (== seleccionConFallo1 p xs)
      [seleccionConFallo2 p xs,
       seleccionConFallo3 p xs]

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

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

-- La comparación es
--    λ> length (seleccionConFallo1 (<5*10^6) [1..])
--    5000000
--    (1.92 secs, 1,240,601,360 bytes)
--    λ> length (seleccionConFallo2 (<5*10^6) [1..])
--    5000000
--    (1.36 secs, 1,280,601,504 bytes)
--    λ> length (seleccionConFallo3 (<5*10^6) [1..])
--    5000000
--    (1.17 secs, 1,280,601,544 bytes)
```

Ejercicio 44

Buscaminas

```
-- ---------------------------------------------------------------------
-- El buscaminas es un juego cuyo objetivo es despejar un campo de minas
-- sin detonar ninguna.
--
-- El campo de minas se representa mediante un cuadrado con NxN
-- casillas. Algunas casillas tienen un número, este número indica las
-- minas que hay en todas las casillas vecinas. Cada casilla tiene como
-- máximo 8 vecinas. Por ejemplo, el campo 4x4 de la izquierda
-- contiene dos minas, cada una representada por el número 9, y a la
-- derecha se muestra el campo obtenido anotando las minas vecinas de
-- cada casilla
--    9 0 0 0       9 1 0 0
--    0 0 0 0       2 2 1 0
--    0 9 0 0       1 9 1 0
--    0 0 0 0       1 1 1 0
-- de la misma forma, la anotación del siguiente a la izquierda es el de
-- la derecha
--    9 9 0 0 0     9 9 1 0 0
--    0 0 0 0 0     3 3 2 0 0
--    0 9 0 0 0     1 9 1 0 0
--
-- Utilizando la librería Data.Matrix, los campos de minas se
-- representan mediante matrices:
--    type Campo = Matrix Int
-- Por ejemplo, los anteriores campos de la izquierda se definen por
--    ejCampo1, ejCampo2 :: Campo
--    ejCampo1 = fromLists [[9,0,0,0],
--                          [0,0,0,0],
--                          [0,9,0,0],
--                          [0,0,0,0]]
--    ejCampo2 = fromLists [[9,9,0,0,0],
--                          [0,0,0,0,0],
```

```
--                          [0,9,0,0,0]]
--
-- Definir la función
--    buscaminas :: Campo -> Campo
-- tal que (buscaminas c) es el campo obtenido anotando las minas
-- vecinas de cada casilla. Por ejemplo,
--    λ> buscaminas1 ejCampo1
--    ┌         ┐
--    │ 9 1 0 0 │
--    │ 2 2 1 0 │
--    │ 1 9 1 0 │
--    │ 1 1 1 0 │
--    └         ┘
--    λ> buscaminas1 ejCampo2
--    ┌           ┐
--    │ 9 9 1 0 0 │
--    │ 3 3 2 0 0 │
--    │ 1 9 1 0 0 │
--    └           ┘
--
-- Notas.
-- 1. El manual de la librería Data.Matrix se encuentra en
--    https://hackage.haskell.org/package/matrix-0.3.6.1/docs/Data-Matrix.html
-- 2. Las funciones de dicha librería útiles para este ejercicio son
--    fromLists, matrix, nrows y ncols.
-- ---------------------------------------------------------------------

module Buscaminas where

import Data.List (foldl')
import Data.Matrix (Matrix, (!), fromLists, matrix, mapPos, nrows, ncols, toLists)
import Control.Monad (replicateM)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

type Campo   = Matrix Int
type Casilla = (Int,Int)

ejCampo1, ejCampo2 :: Campo
ejCampo1 = fromLists [[9,0,0,0],
                      [0,0,0,0],
                      [0,9,0,0],
                      [0,0,0,0]]
ejCampo2 = fromLists [[9,9,0,0,0],
                      [0,0,0,0,0],
```

```
                     [0,9,0,0,0]]

-- 1ª solución
-- ===========

buscaminas1 :: Campo -> Campo
buscaminas1 c = matrix m n (\(i,j) -> minas c (i,j))
  where m = nrows c
        n = ncols c

-- (minas c (i,j)) es el número de minas en las casillas vecinas de la
-- (i,j) en el campo de mina c y es 9 si en (i,j) hay una mina. Por
-- ejemplo,
--    minas ejCampo (1,1)  ==  9
--    minas ejCampo (1,2)  ==  1
--    minas ejCampo (1,3)  ==  0
--    minas ejCampo (2,1)  ==  2
minas :: Campo -> Casilla -> Int
minas c (i,j)
  | c!(i,j) == 9 = 9
  | otherwise    = length (filter (==9) [c!(x,y) | (x,y) <- vecinas m n (i,j)])
  where m = nrows c
        n = ncols c

-- (vecinas m n (i,j)) es la lista de las casillas vecinas de la (i,j) en
-- un campo de dimensiones mxn. Por ejemplo,
--    vecinas 4 (1,1)  ==  [(1,2),(2,1),(2,2)]
--    vecinas 4 (1,2)  ==  [(1,1),(1,3),(2,1),(2,2),(2,3)]
--    vecinas 4 (2,3)  ==  [(1,2),(1,3),(1,4),(2,2),(2,4),(3,2),(3,3),(3,4)]
vecinas :: Int -> Int -> Casilla -> [Casilla]
vecinas m n (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                             b <- [max 1 (j-1)..min n (j+1)],
                             (a,b) /= (i,j)]

-- 2ª solución
-- ===========

buscaminas2 :: Campo -> Campo
buscaminas2 c = matrix m n (\(i,j) -> minas' (i,j))
  where m = nrows c
        n = ncols c
        minas' :: Casilla -> Int
        minas' (i,j)
          | c!(i,j) == 9 = 9
          | otherwise    =
```

```
            length (filter (==9) [c!(x,y) | (x,y) <- vecinas' (i,j)])
      vecinas' :: Casilla -> [Casilla]
      vecinas' (i,j) = [(a,b) | a <- [max 1 (i-1)..min m (i+1)],
                                b <- [max 1 (j-1)..min n (j+1)],
                                (a,b) /= (i,j)]

-- 3ª solución
-- ===========

buscaminas3 :: Campo -> Campo
buscaminas3 c = matrix m n (\(i,j) -> minas2 c (i,j))
  where m = nrows c
        n = ncols c

minas2 :: Campo -> Casilla -> Int
minas2 c (i,j)
  | c!(i,j) == 9 = 9
  | otherwise    = foldl' (\acc v -> if v == 9 then acc+1 else acc)
                          0
                          [c!(x,y) | (x,y) <- vecinas m n (i,j)]
  where m = nrows c
        n = ncols c

-- 4ª solución
-- ===========

buscaminas4 :: Campo -> Campo
buscaminas4 campo = mapPos f campo
  where
    f (i,j) val
        | val == 9 = 9
        | otherwise = contarAlrededor (i,j)
    contarAlrededor (i,j) = length
        [ () | di <- [-1..1], dj <- [-1..1], (di,dj) /= (0,0)
             , let ni = i+di, let nj = j+dj
             , inRange ni nj
             , campo ! (ni,nj) == 9 ]
    inRange i j = i >= 1 && i <= nrows campo
                && j >= 1 && j <= ncols campo

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec
```

```haskell
specG :: (Campo -> Campo)  -> Spec
specG buscaminas = do
  it "e1" $
    toLists (buscaminas ejCampo1) `shouldBe`
    [[9,1,0,0],[2,2,1,0],[1,9,1,0],[1,1,1,0]]
  it "e2" $
    toLists (buscaminas ejCampo2) `shouldBe`
    [[9,9,1,0,0],[3,3,2,0,0],[1,9,1,0,0]]

spec :: Spec
spec = do
  describe "def. 1" $ specG buscaminas1
  describe "def. 2" $ specG buscaminas2
  describe "def. 3" $ specG buscaminas3
  describe "def. 4" $ specG buscaminas4

-- La verificación es
--    λ> verifica
--    4 examples, 0 failures

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

newtype Campo2 = C Campo

instance Show Campo2 where
  show (C p) = show p

-- Generador aleatorio de una casilla (0 = vacío, 9 = mina). Por
-- ejemplo,
--    λ> generate genCasilla
--    9
--    λ> generate genCasilla
--    0
genCasilla :: Gen Int
genCasilla = elements [0,9]

-- Generador de campos. Por ejemplo,
--    λ> generate (genCampo 5)
--    ┌         ┐
--    │ 0 9 9 0 │
--    │ 9 0 0 9 │
--    │ 0 9 0 0 │
--    └         ┘
```

```haskell
genCampo :: Int -> Gen Campo2
genCampo a = do
  let cota = max 1 a
  m <- choose (1,cota)
  n <- choose (1,cota)
  rows <- replicateM m (replicateM n genCasilla)
  return (C (fromLists rows))

instance Arbitrary Campo2 where
  arbitrary = sized genCampo

-- La propiedad es
prop_buscaminas :: Campo2 -> Bool
prop_buscaminas (C p) =
  all (== buscaminas1 p)
      [buscaminas2 p,
       buscaminas3 p,
       buscaminas4 p]

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

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

-- La comparación es
--    λ> C p <- generate (genCampo 20000)
--    λ> length (buscaminas1 p)
--    9414947
--    (3.41 secs, 3,164,606,704 bytes)
--    λ> length (buscaminas2 p)
--    9414947
--    (1.21 secs, 678,499,992 bytes)
--    λ> length (buscaminas3 p)
--    9414947
--    (0.50 secs, 678,499,952 bytes)
--    λ> length (buscaminas4 p)
--    9414947
--    (0.86 secs, 678,499,992 bytes)
```

Ejercicio 45

Mayor sucesión del problema 3n+1

```
-- ---------------------------------------------------------------------
-- La sucesión 3n+1 generada por un número entero positivo x es la
-- sucesión generada por el siguiente algoritmo: Se empieza con el
-- número x. Si x es par, se divide entre 2. Si x es impar, se
-- multiplica por 3 y se le suma 1. El  proceso se repite con el número
-- obtenido hasta que se alcanza el valor 1. Por ejemplo, la sucesión de
-- números generadas cuando se empieza en 22 es
--    22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
-- Se ha conjeturado (aunque no demostrado) que este algoritmo siempre
-- alcanza el 1 empezando en cualquier entero positivo.
--
-- Definir la función
--    mayorLongitud :: Integer -> Integer -> Integer
-- tal que (mayorLongitud i j) es el máximo de las longitudes de las
-- sucesiones 3n+1 para todos los números comprendidos entre i y j,
-- ambos inclusives. Por ejemplo,
--    mayorLongitud   1   10  ==  20
--    mayorLongitud 100  200  ==  125
--    mayorLongitud 201  210  ==  89
--    mayorLongitud 900 1000  ==  174
-- ---------------------------------------------------------------------

module Mayor_sucesion_3n_mas_1 where

import Data.List (genericLength)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
```

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

mayorLongitud1 :: Integer -> Integer -> Integer
mayorLongitud1 i j = maximum [genericLength (sucesion k) | k <- [i..j]]

-- (sucesion n) es la sucesión 3n+1 generada por n. Por ejemplo,
--    sucesion 22  ==  [22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
sucesion :: Integer -> [Integer]
sucesion 1 = [1]
sucesion n | even n    = n : sucesion (n `div` 2)
           | otherwise = n : sucesion (3*n+1)

-- 2ª solución
-- ===========

mayorLongitud2 :: Integer -> Integer -> Integer
mayorLongitud2 i j = maximum [longitud k | k <- [i..j]]

-- (longitud n) es la longitud de la sucesión 3n+1 generada por n. Por
-- ejemplo,
--    longitud 22  ==  16
longitud :: Integer -> Integer
longitud 1 = 1
longitud n | even n    = 1 + longitud (n `div` 2)
           | otherwise = 1 + longitud (3*n+1)

-- 3ª solución (con iterate)
-- =========================

mayorLongitud3 :: Integer -> Integer -> Integer
mayorLongitud3 i j = maximum [genericLength (sucesion2 k) | k <- [i..j]]

-- (sucesion2 n) es la sucesión 3n+1 generada por n. Por ejemplo,
--    sucesion2 22  ==  [22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
sucesion2 :: Integer -> [Integer]
sucesion2 n = takeWhile (/=1) (iterate f n) ++ [1]
    where f x | even x    = x `div` 2
              | otherwise = 3*x+1

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec
```

```
specG :: (Integer -> Integer -> Integer) -> Spec
specG mayorLongitud = do
  it "e1" $
    mayorLongitud   1   10  `shouldBe`  20
  it "e2" $
    mayorLongitud 100  200  `shouldBe`  125
  it "e3" $
    mayorLongitud 201  210  `shouldBe`  89
  it "e4" $
    mayorLongitud 900 1000  `shouldBe`  174

spec :: Spec
spec = do
  describe "def. 1" $ specG mayorLongitud1
  describe "def. 2" $ specG mayorLongitud2
  describe "def. 3" $ specG mayorLongitud3

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- La propiedad es
prop_mayorLongitud :: Positive Integer -> Positive Integer -> Bool
prop_mayorLongitud (Positive i) (Positive j) =
  all (== mayorLongitud1 i (i+j))
      [mayorLongitud2 i (i+j),
       mayorLongitud3 i (i+j)]

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

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

-- La comparación es
--    λ> mayorLongitud1 1 40000
--    324
--    (2.61 secs, 1,714,320,680 bytes)
--    λ> mayorLongitud2 1 40000
--    324
--    (2.64 secs, 1,457,194,704 bytes)
```

```
--    λ> mayorLongitud3 1 40000
--    324
--    (3.77 secs, 2,660,488,832 bytes)
```

Ejercicio 46

Filtro booleano

```
-- ---------------------------------------------------------------------
-- Definir la función
--    filtroBooleano :: [Bool] -> [a] -> [Maybe a]
-- tal que (filtroBooleano xs ys) es la lista de los elementos de ys
-- tales que el elemento de xs en la misma posición es verdadero. Por
-- ejemplo,
--    λ> filtroBooleano [True,False,True] "Sevilla"
--    [Just 'S',Nothing,Just 'v']
--    λ> filtroBooleano (repeat True) "abc"
--    [Just 'a',Just 'b',Just 'c']
--    λ> take 3 (filtroBooleano (repeat True) [1..])
--    [Just 1,Just 2,Just 3]
--    λ> take 3 (filtroBooleano (repeat False) [1..])
--    [Nothing,Nothing,Nothing]
--    λ> take 3 (filtroBooleano (cycle [True,False]) [1..])
--    [Just 1,Nothing,Just 3]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Filtro_booleano where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

filtroBooleano1 :: [Bool] -> [a] -> [Maybe a]
filtroBooleano1 xs ys = [f x y | (x,y) <- zip xs ys]
    where f True y = Just y
          f _    _ = Nothing
```

```haskell
-- 2ª solución
-- ===========

filtroBooleano2 :: [Bool] -> [a] -> [Maybe a]
filtroBooleano2 = zipWith f
    where f True y = Just y
          f _    _ = Nothing

-- 3ª solución
-- ===========

filtroBooleano3 :: [Bool] -> [a] -> [Maybe a]
filtroBooleano3 = zipWith (\x y -> if x then Just y else Nothing)

-- 4ª solución
-- ===========

filtroBooleano4 :: [Bool] -> [a] -> [Maybe a]
filtroBooleano4 (x:xs) (y:ys) | x         = Just y : filtroBooleano4 xs ys
                              | otherwise = Nothing : filtroBooleano4 xs ys
filtroBooleano4 _ _                       = []

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Bool] -> [Int] -> [Maybe Int]) -> Spec
specG filtroBooleano = do
  it "e1" $
    take 3 (filtroBooleano (repeat True) [1..])
    `shouldBe` [Just 1,Just 2,Just 3]
  it "e2" $
    take 3 (filtroBooleano (repeat False) [1..])
    `shouldBe` [Nothing,Nothing,Nothing]
  it "e3" $
    take 3 (filtroBooleano (cycle [True,False]) [1..])
    `shouldBe` [Just 1,Nothing,Just 3]

spec :: Spec
spec = do
  describe "def. 1" $ specG filtroBooleano1
  describe "def. 2" $ specG filtroBooleano2
```

```
  describe "def. 3" $ specG filtroBooleano3
  describe "def. 4" $ specG filtroBooleano4

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- La propiedad es
prop_filtroBooleano :: [Bool] -> [Int] -> Bool
prop_filtroBooleano xs ys =
  all (== filtroBooleano1 xs ys)
      [filtroBooleano2 xs ys,
       filtroBooleano3 xs ys,
       filtroBooleano4 xs ys]

-- La verificación es
--    λ> quickCheck prop_filtroBooleano
--    +++ OK, passed 100 tests.

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

-- La comparación es
--    λ> length (take 10000000 (filtroBooleano1 (cycle [True,False]) [1..]))
--    10000000
--    (2.15 secs, 3,360,601,632 bytes)
--    λ> length (take 10000000 (filtroBooleano2 (cycle [True,False]) [1..]))
--    10000000
--    (0.38 secs, 2,320,601,496 bytes)
--    λ> length (take 10000000 (filtroBooleano3 (cycle [True,False]) [1..]))
--    10000000
--    (0.38 secs, 2,320,601,520 bytes)
--    λ> length (take 10000000 (filtroBooleano4 (cycle [True,False]) [1..]))
--    10000000
--    (3.20 secs, 2,800,602,664 bytes)
```

Ejercicio 47

Entero positivo de la cadena

```
-- ----------------------------------------------------------------------
-- Definir la función
--    enteroPositivo :: String -> Maybe Integer
-- tal que (enteroPositivo cs) es justo el contenido de la cadena cs, si
-- dicho contenido es un entero positivo, y Nothing en caso contrario.
-- Por ejemplo,
--    enteroPositivo "235"    ==  Just 235
--    enteroPositivo "-235"   ==  Nothing
--    enteroPositivo "23.5"   ==  Nothing
--    enteroPositivo "235 "   ==  Nothing
--    enteroPositivo "cinco"  ==  Nothing
--    enteroPositivo ""       ==  Nothing
-- ----------------------------------------------------------------------

module Entero_positivo_de_la_cadena where

import Data.Maybe (listToMaybe)
import Numeric (readDec)
import Data.Char (isDigit)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

enteroPositivo1 :: String -> Maybe Integer
enteroPositivo1 ""                  = Nothing
enteroPositivo1 cs | todosDigitos1 cs = Just (read cs)
                   | otherwise        = Nothing

-- (todosDigitos cs) se verifica si todos los elementos de cs son
-- dígitos. Por ejemplo,
```

```
--    todosDigitos "235"    ==  True
--    todosDigitos "-235"   ==  False
--    todosDigitos "23.5"   ==  False
--    todosDigitos "235 "   ==  False
--    todosDigitos "cinco"  ==  False
todosDigitos1 :: String -> Bool
todosDigitos1 cs =
  and [esDigito1 c | c <- cs]

-- (esDigito c) se verifica si el carácter c es un dígito. Por ejemplo,
--    esDigito '5'  ==  True
--    esDigito 'a'  ==  False
esDigito1 :: Char -> Bool
esDigito1 c = c `elem` "0123456789"

-- 2ª solución
-- ===========

enteroPositivo2 :: String -> Maybe Integer
enteroPositivo2 ""                  = Nothing
enteroPositivo2 cs | todosDigitos2 cs = Just (read cs)
                   | otherwise        = Nothing

todosDigitos2 :: String -> Bool
todosDigitos2 cs =
  and [esDigito2 c | c <- cs]

esDigito2 :: Char -> Bool
esDigito2 c = c `elem` ['0'..'9']

-- 3ª solución
-- ===========

enteroPositivo3 :: String -> Maybe Integer
enteroPositivo3 ""                  = Nothing
enteroPositivo3 cs | todosDigitos3 cs = Just (read cs)
                   | otherwise        = Nothing

todosDigitos3 :: String -> Bool
todosDigitos3 cs =
  and [esDigito3 c | c <- cs]

esDigito3 :: Char -> Bool
esDigito3 = (`elem` ['0'..'9'])
```

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

enteroPositivo4 :: String -> Maybe Integer
enteroPositivo4 ""                  = Nothing
enteroPositivo4 cs | todosDigitos4 cs = Just (read cs)
                   | otherwise        = Nothing

todosDigitos4 :: String -> Bool
todosDigitos4 cs =
  and [esDigito4 c | c <- cs]

esDigito4 :: Char -> Bool
esDigito4 c = '0' <= c && c <= '9'

-- 5ª solución
-- ===========

enteroPositivo5 :: String -> Maybe Integer
enteroPositivo5 ""                  = Nothing
enteroPositivo5 cs | todosDigitos5 cs = Just (read cs)
                   | otherwise        = Nothing

todosDigitos5 :: String -> Bool
todosDigitos5 cs =
  and [isDigit c | c <- cs]

-- 6ª solución
-- ===========

enteroPositivo6 :: String -> Maybe Integer
enteroPositivo6 ""                  = Nothing
enteroPositivo6 cs | todosDigitos6 cs = Just (read cs)
                   | otherwise        = Nothing

todosDigitos6 :: String -> Bool
todosDigitos6 []     = True
todosDigitos6 (c:cs) = isDigit c && todosDigitos6 cs

-- 7ª solución
-- ===========

enteroPositivo7 :: String -> Maybe Integer
enteroPositivo7 ""                  = Nothing
enteroPositivo7 cs | todosDigitos7 cs = Just (read cs)
```

```
                  | otherwise        = Nothing

todosDigitos7 :: String -> Bool
todosDigitos7 = foldr ((&&) . isDigit) True

-- 8ª solución
-- ===========

enteroPositivo8 :: String -> Maybe Integer
enteroPositivo8 ""                   = Nothing
enteroPositivo8 cs | todosDigitos8 cs = Just (read cs)
                   | otherwise        = Nothing

todosDigitos8 :: String -> Bool
todosDigitos8 = all isDigit

-- 9ª solución
-- ===========

enteroPositivo9 :: String -> Maybe Integer
enteroPositivo9 cs
  | null xs   = Nothing
  | otherwise = Just (head xs)
  where xs = [x | (x,y) <- readDec cs, null y]

-- Nota. En la solución anterior se ha usado la función readDec de la
-- librería Numeric. El valor de (readDec cs) es la lista de los pares
-- (x,y) tales que x es el entero positivo al principio de cs e y es el
-- resto. Por ejemplo,
--    readDec "235"    ==  [(235,"")]
--    readDec "-235"   ==  []
--    readDec "23.5"   ==  [(23,".5")]
--    readDec "235 "   ==  [(235," ")]
--    readDec "cinco"  ==  []
--    readDec ""       ==  []

-- 10ª solución
-- ============

enteroPositivo10 :: String -> Maybe Integer
enteroPositivo10 =
    fmap fst . listToMaybe . filter (null . snd) . readDec

-- Nota. En la solución anterior se ha usado la función listToMaybe
-- (de la librería Data.Maybe) tal que (listToMaybe xs) es Nothing si xs
```

```
-- es la lista vacía o (Just x) donde x es el primer elemento de xs. Por
-- ejemplo,
--    listToMaybe []       ==  Nothing
--    listToMaybe [3,2,5]  ==  Just 3
-- y la función fmap tal que (fmap f x) le aplica la función f a los
-- elementos de x. Por ejemplo,
--    fmap (+2) (Just 3)  ==  Just 5
--    fmap (+2) Nothing   ==  Nothing
--    fmap (+2) [3,4,6]   ==  [5,6,8]
--    fmap (+2) []        ==  []

-- Nota. Ejemplos de cálculo con enteroPositivo3
--    enteroPositivo10 "325"
--    = (fmap fst . listToMaybe . filter (null . snd) . readDec) "325"
--    = (fmap fst . listToMaybe . filter (null . snd)) [(325,"")]
--    = (fmap fst . listToMaybe) [(325,"")]
--    = fmap fst (Just (325,""))
--    = Just 325
--
--    enteroPositivo10 "32.5"
--    = (fmap fst . listToMaybe . filter (null . snd) . readDec) "32.5"
--    = (fmap fst . listToMaybe . filter (null . snd)) [(32,".5")]
--    = (fmap fst . listToMaybe ) []
--    = fmap fst Nothing
--    = Nothing

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (String -> Maybe Integer) -> Spec
specG enteroPositivo = do
  it "e1" $
    enteroPositivo "235"    `shouldBe`  Just 235
  it "e2" $
    enteroPositivo "-235"   `shouldBe`  Nothing
  it "e3" $
    enteroPositivo "23.5"   `shouldBe`  Nothing
  it "e4" $
    enteroPositivo "235 "   `shouldBe`  Nothing
  it "e5" $
    enteroPositivo "cinco"  `shouldBe`  Nothing
  it "e6" $
```

```
    enteroPositivo ""       `shouldBe`  Nothing

spec :: Spec
spec = do
  describe "def. 1"  $ specG enteroPositivo1
  describe "def. 2"  $ specG enteroPositivo2
  describe "def. 3"  $ specG enteroPositivo3
  describe "def. 4"  $ specG enteroPositivo4
  describe "def. 5"  $ specG enteroPositivo5
  describe "def. 6"  $ specG enteroPositivo6
  describe "def. 7"  $ specG enteroPositivo7
  describe "def. 8"  $ specG enteroPositivo8
  describe "def. 9"  $ specG enteroPositivo9
  describe "def. 10" $ specG enteroPositivo10

-- La verificación es
--    λ> verifica
--    60 examples, 0 failures

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

-- Generador de cadenas. Por ejemplo.
--    λ> generate cadenaArbitraria
--    "69883777219"
--    λ> generate cadenaArbitraria
--    "iyodnfsw2m78mhu651bvtt7"
cadenaArbitraria :: Gen String
cadenaArbitraria = frequency [
  (1, listOf (elements ['0'..'9'])),                -- 50% solo dígitos
  (1, listOf (elements (['0'..'9'] ++ ['a'..'z']))) -- 50% alfanuméricos
  ]

-- La propiedad es
prop_enteroPositivo :: Property
prop_enteroPositivo = forAll cadenaArbitraria $ \s ->
  all (== enteroPositivo1 s)
      [enteroPositivo2 s,
       enteroPositivo3 s,
       enteroPositivo4 s,
       enteroPositivo5 s,
       enteroPositivo6 s,
       enteroPositivo7 s,
       enteroPositivo8 s,
       enteroPositivo9 s,
```

```
       enteroPositivo10 s]

-- La verificación es
--    λ> quickCheck prop_enteroPositivo
--    +++ OK, passed 100 tests.

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

-- La comparación es
--    λ> ej = replicate (2*10^6) '5'
--    λ> (length . show) <$> enteroPositivo1 ej
--    Just 2000000
--    (2.19 secs, 2,187,674,312 bytes)
--    λ> (length . show) <$> enteroPositivo2 ej
--    Just 2000000
--    (1.97 secs, 2,235,673,400 bytes)
--    λ> (length . show) <$> enteroPositivo3 ej
--    Just 2000000
--    (1.52 secs, 1,243,673,336 bytes)
--    λ> (length . show) <$> enteroPositivo4 ej
--    Just 2000000
--    (1.87 secs, 1,467,673,352 bytes)
--    λ> (length . show) <$> enteroPositivo5 ej
--    Just 2000000
--    (1.45 secs, 1,243,673,400 bytes)
--    λ> (length . show) <$> enteroPositivo6 ej
--    Just 2000000
--    (1.49 secs, 1,099,673,184 bytes)
--    λ> (length . show) <$> enteroPositivo7 ej
--    Just 2000000
--    (1.04 secs, 1,035,673,176 bytes)
--    λ> (length . show) <$> enteroPositivo8 ej
--    Just 2000000
--    (0.97 secs, 1,019,673,232 bytes)
--    λ> (length . show) <$> enteroPositivo9 ej
--    Just Interrupted.
--    λ> (length . show) <$> enteroPositivo10 ej
--    Just Interrupted.
```

Ejercicio 48

N gramas

```haskell
-- ----------------------------------------------------------------------
-- Un n-grama de una sucesión es una subsucesión contigua de n elementos.
--
-- Definir la función
--    nGramas :: Int -> [a] -> [[a]]
-- tal que (nGramas k xs) es la lista de los n-gramas de xs de longitud
-- k. Por ejemplo,
--    nGramas 0 "abcd"  ==  []
--    nGramas 1 "abcd"  ==  ["a","b","c","d"]
--    nGramas 2 "abcd"  ==  ["ab", "bc", "cd"]
--    nGramas 3 "abcd"  ==  ["abc", "bcd"]
--    nGramas 4 "abcd"  ==  ["abcd"]
--    nGramas 5 "abcd"  ==  []
-- ----------------------------------------------------------------------

module N_gramas where

import Data.List (unfoldr)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

nGramas1 :: Int -> [a] -> [[a]]
nGramas1 k xs
  | k <= 0    = []
  | k > n     = []
  | otherwise = take k xs : nGramas1 k (tail xs)
  where n = length xs

-- 2ª solución
```

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

nGramas2 :: Int -> [a] -> [[a]]
nGramas2 k xs
  | k <= 0    = []
  | k > n     = []
  | otherwise = unfoldr aux xs
  where n = length xs
        aux ys | length ys < k = Nothing
               | otherwise     = Just (take k ys, tail ys)

-- 3ª solución
-- ===========

nGramas3 :: Int -> [a] -> [[a]]
nGramas3 k xs
  | k <= 0    = []
  | otherwise = aux k (length xs) xs
  where
    aux k' n ys
      | k' > n    = []
      | otherwise = take k' ys : aux k' (n-1) (tail ys)

-- 4ª solución
-- ===========

nGramas4 :: Int -> [a] -> [[a]]
nGramas4 k xs
  | k <= 0    = []
  | otherwise = [take k (drop i xs) | i <- [0..length xs - k]]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> String -> [String]) -> Spec
specG nGramas = do
  it "e1" $
    nGramas 0 "abcd"  `shouldBe`  []
  it "e2" $
    nGramas 1 "abcd"  `shouldBe`  ["a","b","c","d"]
  it "e3" $
    nGramas 2 "abcd"  `shouldBe`  ["ab", "bc", "cd"]
```

```
  it "e4" $
    nGramas 3 "abcd"  `shouldBe`  ["abc", "bcd"]
  it "e5" $
    nGramas 4 "abcd"  `shouldBe`  ["abcd"]
  it "e6" $
    nGramas 5 "abcd"  `shouldBe`  []

spec :: Spec
spec = do
  describe "def. 1"  $ specG nGramas1
  describe "def. 2"  $ specG nGramas2
  describe "def. 3"  $ specG nGramas3
  describe "def. 4"  $ specG nGramas4

-- La verificación es
--    λ> verifica
--    24 examples, 0 failures

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

-- La propiedad es
prop_nGramas :: NonNegative Int -> [Int] -> Bool
prop_nGramas (NonNegative k) xs =
  all (== nGramas1 k xs)
      [nGramas2 k xs,
       nGramas3 k xs,
       nGramas4 k xs]

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

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

-- La comparación es
--    λ> length (nGramas1 20000 [1..40000])
--    20001
--    (3.25 secs, 10,839,512 bytes)
--    λ> length (nGramas2 20000 [1..40000])
--    20001
--    (3.25 secs, 10,519,720 bytes)
--    λ> length (nGramas3 20000 [1..40000])
--    20001
```

```
--    (0.04 secs, 11,159,656 bytes)
--    λ> length (nGramas4 20000 [1..40000])
--    20001
--    (0.03 secs, 7,479,488 bytes)
--
--    λ> length (nGramas3 1000000 [1..10000000])
--    9000001
--    (6.87 secs, 4,176,601,176 bytes)
--    λ> length (nGramas4 1000000 [1..10000000])
--    9000001
--    (3.25 secs, 2,520,601,008 bytes)
```

Ejercicio 49

Sopa de letras

```
-- ---------------------------------------------------------------------
-- Las matrices se puede representar mediante tablas cuyos índices son
-- pares de números naturales:
--    type Matriz a = Array (Int,Int) a
--
-- Definir la función
--    enLaSopa :: Eq a => [a] -> Matriz a -> Bool
-- tal que (enLaSopa c p) se verifica si c está en la matriz p en
-- horizontal o en vertical. Por ejemplo, si ej1 es la matriz siguiente:
--    ej1 :: Matriz Char
--    ej1 = listaMatriz [”mjtholueq”,
--                       ”juhoolauh”,
--                       ”dariouhyj”,
--                       ”rngkploaa”]
-- donde la función listaMatriz está definida por
--    listaMatriz :: [[a]] -> Matriz a
--    listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
--      where m = length xss
--            n = length (head xss)
-- entonces,
--    enLaSopa ”dar”  ej1  ==  True   -- En horizontal a la derecha en la 3ª fila
--    enLaSopa ”oir”  ej1  ==  True   -- En horizontal a la izquierda en la 3ª fila
--    enLaSopa ”juan” ej1  ==  True   -- En vertical descendente en la 2ª columna
--    enLaSopa ”kio”  ej1  ==  True   -- En vertical ascendente en la 3ª columna
--    enLaSopa ”Juan” ej1  ==  False
--    enLaSopa ”hola” ej1  ==  False
-- ---------------------------------------------------------------------

module Sopa_de_letras where

import Data.Array (Array, (!), bounds, listArray)
import Data.List (isInfixOf, transpose)
```

```haskell
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

type Matriz a = Array (Int,Int) a

listaMatriz :: [[a]] -> Matriz a
listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
  where m = length xss
        n = length (head xss)

ej1 :: Matriz Char
ej1 = listaMatriz ["mjtholueq",
                   "juhoolauh",
                   "dariouhyj",
                   "rngkploaa"]

-- 1ª solución
-- ===========

enLaSopa1 :: Eq a => [a] -> Matriz a -> Bool
enLaSopa1 c p =
    or [c `isInfixOf` xs |
        xs <- [[p!(i,j) | j <- [1..n]]     | i <- [1..m]] ++
              [[p!(i,j) | j <- [n,n-1..1]] | i <- [1..m]] ++
              [[p!(i,j) | i <- [1..m]]     | j <- [1..n]] ++
              [[p!(i,j) | i <- [m,m-1..1]] | j <- [1..n]]]
    where (_,(m,n)) = bounds p

-- 2ª solución
-- ===========

enLaSopa2 :: Eq a => [a] -> Matriz a -> Bool
enLaSopa2 c p = estaEnHorizontal c p || estaEnVertical c p

-- (numFilas p) es el número de filas de la matriz p. Por ejemplo,
--    numFilas ej1  ==  4
numFilas :: Matriz a -> Int
numFilas = fst . snd . bounds

-- (numColumnas p) es el número de columnas de la matriz p. Por ejemplo,
--    numColumnas ej1  ==  9
numColumnas :: Matriz a -> Int
numColumnas = snd . snd . bounds

-- (fila i p) es la fila i-ésima de la matriz p. Por ejemplo,
```

```
--    fila 2 ej1  ==  "juhoolauh"
fila :: Int -> Matriz a -> [a]
fila i p =
  [p!(i,j) | j <- [1..n]]
  where n = numColumnas p

-- (columna j p) es la columna j-ésima de la matriz p. Por ejemplo,
--    columna 2 ej1  ==  "juan"
columna :: Int -> Matriz a -> [a]
columna j p =
  [p!(i,j) | i <- [1..m]]
  where m = numFilas p

-- (filas p) es la lista de las filas de la matriz p. Por ejemplo,
--    λ> filas ej1
--    ["mjtholueq","juhoolauh","dariouhyj","rngkploaa"]
filas :: Matriz a -> [[a]]
filas p =
  [fila i p | i <- [1..numFilas p]]

-- (columnas p) es la lista de las columnas de la matriz p. Por ejemplo,
--    λ> columnas ej1
--    ["mjdr","juan","thrg","hoik","ooop","llul","uaho","euya","qhja"]
columnas :: Matriz a -> [[a]]
columnas p =
  [columna j p | j <- [1..numColumnas p]]

-- (estaEnHorizontal c p) se verifica si c está en la matriz p en
-- horizontal. Por ejemplo,
--    estaEnHorizontal "dar" ej1  ==  True
--    estaEnHorizontal "oir" ej1  ==  True
--    estaEnHorizontal "juan" ej1 ==  False
estaEnHorizontal :: Eq a => [a] -> Matriz a -> Bool
estaEnHorizontal c p =
  or [c `isInfixOf` xs | xs <- filas p ++ map reverse (filas p)]

-- (estaEnVertical c p) se verifica si c está en la matriz p en
-- vertical. Por ejemplo,
--    estaEnVertical "juan" ej1  ==  True
--    estaEnVertical "kio"  ej1  ==  True
--    estaEnVertical "dar"  ej1  ==  False
estaEnVertical :: Eq a => [a] -> Matriz a -> Bool
estaEnVertical c p =
  or [c `isInfixOf` xs | xs <- columnas p ++ map reverse (columnas p)]
```

```haskell
-- 3ª solución
-- ===========

enLaSopa3 :: Eq a => [a] -> Matriz a -> Bool
enLaSopa3 c p = any (c `isInfixOf`) lineas
  where
    fs = filas p
    cs = transpose fs
    lineas = fs ++ map reverse fs ++ cs ++ map reverse cs

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([Char] -> Matriz Char -> Bool) -> Spec
specG enLaSopa = do
  it "e1" $
    enLaSopa "dar"  ej1  `shouldBe`  True
  it "e2" $
    enLaSopa "oir"  ej1  `shouldBe`  True
  it "e3" $
    enLaSopa "juan" ej1  `shouldBe`  True
  it "e4" $
    enLaSopa "kio"  ej1  `shouldBe`  True
  it "e5" $
    enLaSopa "Juan" ej1  `shouldBe`  False
  it "e6" $
    enLaSopa "hola" ej1  `shouldBe`  False

spec :: Spec
spec = do
  describe "def. 1"  $ specG enLaSopa1
  describe "def. 2"  $ specG enLaSopa2
  describe "def. 3"  $ specG enLaSopa3

-- La verificación es
--    λ> verifica
--    18 examples, 0 failures

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

newtype Matriz2 = M (Array (Int,Int) Int)
```

```
  deriving Show

-- Generador de matrices arbitrarias. Por ejemplo,
--    λ> generate matrizArbitraria
--    M (array ((1,1),(3,4))
--             [((1,1),18),((1,2),6), ((1,3),-23),((1,4),-13),
--              ((2,1),-2),((2,2),22),((2,3),-25),((2,4),-5),
--              ((3,1),2), ((3,2),16),((3,3),-15),((3,4),7)])
matrizArbitraria :: Gen Matriz2
matrizArbitraria = do
  m  <- chooseInt (1,10)
  n  <- chooseInt (1,10)
  xs <- vectorOf (m*n) arbitrary
  return (M (listArray ((1,1),(m,n)) xs))

-- Matriz es una subclase de Arbitrary.
instance Arbitrary Matriz2 where
  arbitrary = matrizArbitraria

-- La propiedad es
prop_enLaSopa :: [Int] -> Matriz2 -> Bool
prop_enLaSopa xs (M p) =
  all (== enLaSopa1 xs p)
      [enLaSopa2 xs p,
       enLaSopa3 xs p]

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

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

-- La comparación es
--    λ> ejemplo = listaMatriz (replicate 1000 (replicate 1000 'a'))
--    λ> enLaSopa1 "b" ejemplo
--    False
--    (1.88 secs, 1,458,976,016 bytes)
--    λ> enLaSopa2 "b" ejemplo
--    False
--    (1.72 secs, 1,491,736,440 bytes)
--    λ> enLaSopa3 "b" ejemplo
--    False
--    (0.70 secs, 553,799,536 bytes)
```

Ejercicio 50

Intercalación de n copias

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    intercala :: Int -> a -> [a] -> [[a]]
-- tal que (intercala n x ys) es la lista de la listas obtenidas
-- intercalando n copias de x en ys, suponiendo que x no pertenece a
-- ys. Por ejemplo,
--    intercala 2 'a' "bc" == ["bcaa","baca","baac","abca","abac","aabc"]
--    intercala 2 'a' "c"  == ["caa","aca","aac"]
--    intercala 1 'a' "c"  == ["ca","ac"]
--    intercala 0 'a' "c"  == ["c"]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Intercala_n_copias where

import Data.List (nub, sort)
import Data.Set (fromList, singleton, toList)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

intercala1 :: Int -> a -> [a] -> [[a]]
intercala1 0 _ xs = [xs]
intercala1 n y [] = [replicate n y]
intercala1 n y (x:xs) =
  concat [[replicate i y ++ x : zs | zs <- intercala1 (n-i) y xs]
          | i <- [0..n]]

-- 2ª solución
```

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

intercala2 :: Int -> a -> [a] -> [[a]]
intercala2 0 _ xs = [xs]
intercala2 n y [] = [replicate n y]
intercala2 n y (x:xs) =
  concatMap (\i -> [replicate i y ++ x : zs | zs <- intercala1 (n-i) y xs]) [0..n]

-- 3ª solución
-- ===========

intercala3 :: Int -> a -> [a] -> [[a]]
intercala3 0 _ ys = [ys]
intercala3 n x ys = aux n ys []
  where
    aux 0 ys' zs = [zs ++ ys']
    aux n' [] zs = [zs ++ replicate n' x]
    aux n' (y:ys') zs =
      aux n' ys' (zs ++ [y]) ++
      aux (n'-1) (y:ys') (zs ++ [x])

-- 4ª solución
-- ===========

intercala4 :: Eq a => Int -> a -> [a] -> [[a]]
intercala4 n x ys = nub (aux n ys)
  where
    aux 0 ys' = [ys']
    aux n' ys' = concat [intercalaUno x zs | zs <- aux (n'-1) ys']

-- (intercalaUno x ys) es la lista de las listas obtenidas intercalando
-- una copia de x en ys. Por ejemplo,
--    intercalaUno 'a' "bc"  == ["abc","bac","bca"]
intercalaUno :: a -> [a] -> [[a]]
intercalaUno x []     = [[x]]
intercalaUno x (y:ys) = (x:y:ys) : [y:zs | zs <- intercalaUno x ys]

-- 5ª solución
-- ===========

intercala5 :: Ord a => Int -> a -> [a] -> [[a]]
intercala5 n x ys = toList (aux n ys)
  where
    aux 0 ys'  = singleton ys'
    aux n' ys' = fromList (concat [intercalaUno x zs | zs <- toList (aux (n'-1) ys')])
```

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

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> Char -> String -> [String]) -> Spec
specG intercala = do
  it "e1" $
    intercala' 2 'a' "bc" `shouldBe` ["aabc", "abac", "abca", "baac", "baca", "bcaa"]
  it "e2" $
    intercala' 2 'a' "c"  `shouldBe` ["aac", "aca", "caa"]
  it "e3" $
    intercala' 1 'a' "c"  `shouldBe` ["ac", "ca"]
  it "e4" $
    intercala' 0 'a' "c"  `shouldBe` ["c"]
  where intercala' n x ys = sort (intercala n x ys)

spec :: Spec
spec = do
  describe "def. 1"  $ specG intercala1
  describe "def. 2"  $ specG intercala2
  describe "def. 3"  $ specG intercala3
  describe "def. 4"  $ specG intercala4
  describe "def. 5"  $ specG intercala5

-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- La propiedad es
prop_intercala :: Int -> Int -> [Int] -> Bool
prop_intercala n x ys =
  sort (intercala1 m x ys') == sort (intercala2 m x ys')
  where m = n `mod` 3
        ys' = filter (/= x) ys

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

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

-- La comparación es
--    λ> length (intercala1 3 'a' ['b'..'z'])
--    3276
--    (0.06 secs, 17,106,368 bytes)
--    λ> length (intercala2 3 'a' ['b'..'z'])
--    3276
--    (0.06 secs, 17,104,248 bytes)
--    λ> length (intercala3 3 'a' ['b'..'z'])
--    3276
--    (0.01 secs, 6,077,080 bytes)
--    λ> length (intercala4 3 'a' ['b'..'z'])
--    3276
--    (1.51 secs, 49,533,208 bytes)
--    λ> length (intercala5 3 'a' ['b'..'z'])
--    3276
--    (0.07 secs, 31,249,936 bytes)
--
--    λ> length (intercala1 5 'a' ['b'..'z'])
--    142506
--    (1.32 secs, 767,853,344 bytes)
--    λ> length (intercala2 5 'a' ['b'..'z'])
--    142506
--    (1.34 secs, 767,852,344 bytes)
--    λ> length (intercala3 5 'a' ['b'..'z'])
--    142506
--    (0.22 secs, 255,971,880 bytes)
--    λ> length (intercala5 5 'a' ['b'..'z'])
--    142506
--    (6.63 secs, 2,684,942,184 bytes)
```

Ejercicio 51

Eliminación de n elementos

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    elimina :: Int -> [a] -> [[a]]
-- tal que (elimina n xs) es la lista de las listas obtenidas eliminando
-- n elementos de xs. Por ejemplo,
--    elimina 0 "abcd"  ==  ["abcd"]
--    elimina 1 "abcd"  ==  ["bcd","acd","abd","abc"]
--    elimina 2 "abcd"  ==  ["cd","bd","bc","ad","ac","ab"]
--    elimina 3 "abcd"  ==  ["d","c","b","a"]
--    elimina 4 "abcd"  ==  [""]
--    elimina 5 "abcd"  ==  []
--    elimina 6 "abcd"  ==  []
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Elimina_n_elementos where

import Data.List (sort, subsequences)
import Math.Combinat.Sets (choose)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

elimina1 :: Int -> [a] -> [[a]]
elimina1 0 xs     = [xs]
elimina1 _ []     = []
elimina1 n (x:xs) = elimina1 (n-1) xs ++ [x:ys | ys <- elimina1 n xs]

-- 2ª solución
```

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

elimina2 :: Int -> [a] -> [[a]]
elimina2 0 xs     = [xs]
elimina2 _ []     = []
elimina2 n (x:xs) = elimina2 (n-1) xs ++ map (x:) (elimina2 n xs)

-- 3ª solución
-- ===========

elimina3 :: Int -> [a] -> [[a]]
elimina3 n xs =
  reverse [ys | ys <- subsequences xs, length ys == k]
  where k = length xs - n

-- 4ª solución
-- ===========

elimina4 :: Int -> [a] -> [[a]]
elimina4 n xs = combinaciones (length xs - n) xs

-- (combinaciones k xs) es la lista de las combinaciones de orden k de
-- los elementos de la lista xs. Por ejemplo,
--    λ> combinaciones 2 "bcde"
--    ["bc","bd","be","cd","ce","de"]
--    λ> combinaciones 3 "bcde"
--    ["bcd","bce","bde","cde"]
--    λ> combinaciones 3 "abcde"
--    ["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
combinaciones :: Int -> [a] -> [[a]]
combinaciones 0 _          = [[]]
combinaciones _ []         = []
combinaciones k (x:xs) =
  [x:ys | ys <- combinaciones (k-1) xs] ++ combinaciones k xs

-- 5ª solución
-- ===========

elimina5 :: Int -> [a] -> [[a]]
elimina5 n xs = combinaciones2 (length xs - n) xs

combinaciones2 :: Int -> [a] -> [[a]]
combinaciones2 0 _      = [[]]
combinaciones2 _ []     = []
combinaciones2 k (x:xs) =
```

```haskell
  map (x:) (combinaciones2 (k-1) xs) ++ combinaciones2 k xs

-- 6ª solución
-- ===========

elimina6 :: Int -> [a] -> [[a]]
elimina6 n xs
  | n < 0 || n > length xs = []
  | otherwise = selecciona (length xs - n) xs
  where
    selecciona 0 _      = [[]]
    selecciona _ []     = []
    selecciona k (y:ys) = map (y:) (selecciona (k-1) ys) ++ selecciona k ys

-- 7ª solución
-- ===========

elimina7 :: Int -> [a] -> [[a]]
elimina7 n xs = choose (length xs - n) xs

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> String -> [String]) -> Spec
specG elimina = do
  it "e1" $
    elimina' 0 "abcd"  `shouldBe` ["abcd"]
  it "e2" $
    elimina' 1 "abcd"  `shouldBe` ["abc","abd","acd","bcd"]
  it "e3" $
    elimina' 2 "abcd"  `shouldBe` ["ab","ac","ad","bc","bd","cd"]
  it "e4" $
    elimina' 3 "abcd"  `shouldBe` ["a","b","c","d"]
  it "e5" $
    elimina' 4 "abcd"  `shouldBe` [""]
  it "e6" $
    elimina' 5 "abcd"  `shouldBe` []
  it "e7" $
    elimina' 6 "abcd"  `shouldBe` []
  where elimina' n xs = sort (elimina n xs)

spec :: Spec
```

```
spec = do
  describe "def. 1"  $ specG elimina1
  describe "def. 2"  $ specG elimina2
  describe "def. 3"  $ specG elimina3
  describe "def. 4"  $ specG elimina4
  describe "def. 5"  $ specG elimina5
  describe "def. 6"  $ specG elimina6
  describe "def. 7"  $ specG elimina7

-- La verificación es
--    λ> verifica
--    49 examples, 0 failures

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

-- La propiedad es
prop_elimina :: Int -> [Int] -> Bool
prop_elimina n xs =
  all (`igual` elimina1 n' xs')
      [elimina2 n' xs',
       elimina3 n' xs',
       elimina4 n' xs',
       elimina5 n' xs',
       elimina6 n' xs',
       elimina7 n' xs']
  where igual as bs = sort as == sort bs
        n' = n `mod` 3
        xs' = take 10 xs

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

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

-- La comparación es
--    λ> length (elimina1 3 [1..21])
--    1330
--    (0.01 secs, 3,682,200 bytes)
--    λ> length (elimina2 3 [1..21])
--    1330
--    (0.02 secs, 3,082,664 bytes)
--    λ> length (elimina3 3 [1..21])
```

```
--    1330
--    (0.79 secs, 436,934,736 bytes)
--    λ> length (elimina4 3 [1..21])
--    1330
--    (1.69 secs, 893,960,152 bytes)
--    λ> length (elimina5 3 [1..21])
--    1330
--    (1.62 secs, 859,643,368 bytes)
--    λ> length (elimina6 3 [1..21])
--    1330
--    (2.72 secs, 1,329,353,632 bytes)
--    λ> length (elimina7 3 [1..21])
--    1330
--    (0.07 secs, 119,942,472 bytes)
--
--    λ> length (elimina1 3 [1..27])
--    2925
--    (0.01 secs, 8,903,640 bytes)
--    λ> length (elimina2 3 [1..27])
--    2925
--    (0.03 secs, 7,165,992 bytes)
--    λ> length (elimina7 3 [1..27])
--    2925
--    (2.16 secs, 7,522,387,728 bytes)
--
--    λ> length (elimina1 3 [1..150])
--    551300
--    (10.12 secs, 7,643,122,544 bytes)
--    λ> length (elimina2 3 [1..150])
--    551300
--    (4.31 secs, 5,689,134,120 bytes)
```

Ejercicio 52

Límite de sucesiones

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    limite :: (Double -> Double) -> Double -> Double
-- tal que (limite f a) es el valor de f en el primer término x tal que,
-- para todo y entre x+1 y x+100, el valor absoluto de la diferencia
-- entre f(y) y f(x) es menor que a. Por ejemplo,
--    limite (\n -> (2*n+1)/(n+5)) 0.001  ==  1.9900110987791344
--    limite (\n -> (1+1/n)**n) 0.001     ==  2.714072874546881
-- ---------------------------------------------------------------------

module Limites_de_sucesiones where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

limite1 :: (Double -> Double) -> Double -> Double
limite1 f a = buscaX 1
  where
    buscaX x
      | cumpleCondicion x = f x
      | otherwise         = buscaX (x + 1)
    cumpleCondicion x = all (\y -> abs (f y - f x) < a) [x+1 .. x+100]

-- 2ª solución
-- ===========

limite2 :: (Double -> Double) -> Double -> Double
limite2 f a =
  head [f x | x <- [1..],
```

```
              all (\y -> abs (f y - f x) < a) [x+1 .. x+100]]

-- 3ª solución
-- ===========

limite3 :: (Double -> Double) -> Double -> Double
limite3 f a =
  head [f x | x <- [1..],
              maximum [abs (f y - f x) | y <- [x+1..x+100]] < a]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ((Double -> Double) -> Double -> Double) -> Spec
specG limite = do
  it "e1" $
    limite (\n -> (2*n+1)/(n+5)) 0.001  `shouldBe`  1.9900110987791344
  it "e2" $
    limite (\n -> (1+1/n)**n) 0.001     `shouldBe`  2.714072874546881

spec :: Spec
spec = do
  describe "def. 1"  $ specG limite1
  describe "def. 2"  $ specG limite2
  describe "def. 3"  $ specG limite3

-- La verificación es
--    λ> verifica
--    2 examples, 0 failures

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

-- Definimos un tipo para representar funciones simples de forma que
-- (F a b c d) representa λn -> (a*n + b)/(c*n + d)
data Funcion = F Int Int Int Int
  deriving Show

-- Para generar funciones arbitrarias
instance Arbitrary Funcion where
  arbitrary = do
    a <- choose (1, 10)
```

```
    b <- choose (0, 10)
    c <- choose (1, 10)
    d <- choose (1, 10)
    return (F a b c d)

-- (aFuncion (F a b c d)) es la función λn -> (a*n + b)/(c*n + d)
aFuncion :: Funcion -> Double -> Double
aFuncion (F a b c d) n =
  fromIntegral (a * round n + b) / fromIntegral (c * round n + d)

-- La propiedad es
prop_limite :: Funcion -> Positive Double -> Bool
prop_limite func (Positive a) =
  let f = aFuncion func
      a' = min a 0.1
      l1 = limite1 f a'
      l2 = limite2 f a'
      l3 = limite3 f a'
  in abs (l1 - l2) < 1e-10 && abs (l2 - l3) < 1e-10

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

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

-- La comparación es
--    λ> limite1 (\n -> (2*n+1)/(n+5)) 0.00001
--    1.9990463070891173
--    (0.52 secs, 298,641,448 bytes)
--    λ> limite2 (\n -> (2*n+1)/(n+5)) 0.00001
--    1.9990463070891173
--    (0.52 secs, 298,264,360 bytes)
--    λ> limite3 (\n -> (2*n+1)/(n+5)) 0.00001
--    1.9990463070891173
--    (1.51 secs, 859,004,120 bytes)
```

Ejercicio 53

Empiezan con mayúscula

```
-- ---------------------------------------------------------------------
-- Definir, por composición, la función
--    conMayuscula :: String -> Int
-- tal que (conMayuscula cs) es el número de palabras de cs que empiezan
-- con mayúscula. Por ejemplo.
--    conMayuscula "Juan vive en Sevilla o en Huelva"  ==  3
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Empiezan_con_mayuscula where

import Data.List (foldl')
import Data.Char (isUpper)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

conMayuscula1 :: String -> Int
conMayuscula1 cs =
  length [p | p <- words cs, isUpper (head p)]

-- 2ª solución
-- ===========

conMayuscula2 :: String -> Int
conMayuscula2 cs = length (filter empiezaConMayuscula (words cs))
  where
    empiezaConMayuscula p = isUpper (head p)
```

```haskell
-- 3ª solución
-- ===========

conMayuscula3 :: String -> Int
conMayuscula3 cs = aux (words cs)
  where
    aux [] = 0
    aux (p:ps)
      | isUpper (head p) = 1 + aux ps
      | otherwise        = aux ps

-- 4ª solución
-- ===========

conMayuscula4 :: String -> Int
conMayuscula4 cs = foldr aux 0 (words cs)
  where
    aux p n
      | isUpper (head p) = 1 + n
      | otherwise        = n

-- 5ª solución
-- ===========

conMayuscula5 :: String -> Int
conMayuscula5 =
  foldr (\p n -> fromEnum (isUpper (head p)) + n) 0 . words

-- 6ª solución
-- ===========

conMayuscula6 :: String -> Int
conMayuscula6 =
  foldr ((+) . fromEnum . isUpper . head) 0 . words

-- 7ª solución
-- ===========

conMayuscula7 :: String -> Int
conMayuscula7 cs = foldl aux 0 (words cs)
  where
    aux n p
      | isUpper (head p) = n + 1
      | otherwise        = n
```

```haskell
-- 8ª solución
-- ===========

conMayuscula8 :: String -> Int
conMayuscula8 cs = foldl' aux 0 (words cs)
  where
    aux n p
      | isUpper (head p) = n + 1
      | otherwise        = n

-- 9ª solución
-- ===========

conMayuscula9 :: String -> Int
conMayuscula9 =
  foldl' (\n p -> n + fromEnum (isUpper (head p))) 0 . words

-- 10ª solución
-- ============

conMayuscula10 :: String -> Int
conMayuscula10 =
  foldl' (flip ((+) . fromEnum . isUpper . head)) 0 . words

-- 11ª solución
-- ===========

conMayuscula11 :: String -> Int
conMayuscula11 =
  length . filter (isUpper . head) . words

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (String -> Int) -> Spec
specG conMayuscula = do
  it "e1" $
    conMayuscula "Juan vive en Sevilla o en Huelva"  `shouldBe`  3

spec :: Spec
spec = do
  describe "def. 1"  $ specG conMayuscula1
```

```
  describe "def. 2"  $ specG conMayuscula2
  describe "def. 3"  $ specG conMayuscula3
  describe "def. 4"  $ specG conMayuscula4
  describe "def. 5"  $ specG conMayuscula5
  describe "def. 6"  $ specG conMayuscula6
  describe "def. 7"  $ specG conMayuscula7
  describe "def. 8"  $ specG conMayuscula8
  describe "def. 9"  $ specG conMayuscula9
  describe "def. 10"  $ specG conMayuscula10
  describe "def. 11"  $ specG conMayuscula11

-- La verificación es
--    λ> verifica
--    2 examples, 0 failures

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

-- La propiedad es
prop_conMayuscula :: String -> Bool
prop_conMayuscula cs =
  all (== conMayuscula1 cs)
      [conMayuscula2 cs,
       conMayuscula3 cs,
       conMayuscula4 cs,
       conMayuscula5 cs,
       conMayuscula6 cs,
       conMayuscula7 cs,
       conMayuscula8 cs,
       conMayuscula9 cs,
       conMayuscula10 cs,
       conMayuscula11 cs]

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

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

-- (ejemplo n) es la cadena obtenida repitiendo n veces la cadena
-- "Hoy es Lunes". Por ejemplo
--    λ> ejemplo 3
--    "Hoy es Lunes Hoy es Lunes Hoy es Lunes"
ejemplo :: Int -> String
```

```
ejemplo n =
  unwords (concat (replicate n ["Hoy", "es", "Lunes"]))

-- La comparación es
--    λ> conMayuscula1 (ejemplo 1000000)
--    2000000
--    (1.62 secs, 3,080,601,344 bytes)
--    λ> conMayuscula2 (ejemplo 1000000)
--    2000000
--    (1.12 secs, 2,952,601,456 bytes)
--    λ> conMayuscula3 (ejemplo 1000000)
--    2000000
--    (2.39 secs, 3,489,575,816 bytes)
--    λ> conMayuscula4 (ejemplo 1000000)
--    2000000
--    (2.13 secs, 3,464,592,760 bytes)
--    λ> conMayuscula5 (ejemplo 1000000)
--    2000000
--    (2.14 secs, 3,301,526,888 bytes)
--    λ> conMayuscula6 (ejemplo 1000000)
--    2000000
--    (1.08 secs, 3,324,347,536 bytes)
--    λ> conMayuscula7 (ejemplo 1000000)
--    2000000
--    (3.71 secs, 3,488,396,128 bytes)
--    λ> conMayuscula8 (ejemplo 1000000)
--    2000000
--    (1.77 secs, 3,296,601,480 bytes)
--    λ> conMayuscula9 (ejemplo 1000000)
--    2000000
--    (1.76 secs, 3,104,601,400 bytes)
--    λ> conMayuscula10 (ejemplo 1000000)
--    2000000
--    (0.72 secs, 3,128,601,768 bytes)
--    λ> conMayuscula11 (ejemplo 1000000)
--    2000000
--    (0.61 secs, 2,904,601,664 bytes)
```

Ejercicio 54

Renombramiento de un árbol

```
-- ----------------------------------------------------------------------
-- Los árboles binarios se pueden representar mediante el tipo Arbol
-- definido por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq, Foldable, Functor)
-- Por ejemplo, el árbol
--          "C"
--          / \
--         /   \
--        /     \
--      "B"     "A"
--      / \     / \
--    "A" "B" "B" "C"
-- se puede definir por
--    ej1 :: Arbol String
--    ej1 = N "C" (N "B" (H "A") (H "B")) (N "A" (H "B") (H "C"))
--
-- Definir la función
--    renombraArbol :: Arbol t -> Arbol Int
-- tal que (renombraArbol a) es el árbol obtenido sustituyendo el valor
-- de los nodos y hojas por números tales que tengan el mismo valor si y
-- sólo si coincide su contenido. Por ejemplo,
--    λ> renombraArbol ej1
--    N 2 (N 1 (H 0) (H 1)) (N 0 (H 1) (H 2))
-- Gráficamente,
--           2
--          / \
--         /   \
--        /     \
--       1       0
--      / \     / \
```

```
--    0   1   1   2
--
-- Nótese que los elementos del árbol pueden ser de cualquier tipo. Por
-- ejemplo,
--    λ> renombraArbol (N 9 (N 4 (H 8) (H 4)) (N 8 (H 4) (H 9)))
--    N 2 (N 0 (H 1) (H 0)) (N 1 (H 0) (H 2))
--    λ> renombraArbol (N True (N False (H True) (H False)) (H True))
--    N 1 (N 0 (H 1) (H 0)) (H 1)
--    λ> renombraArbol (N False (N False (H True) (H False)) (H True))
--    N 0 (N 0 (H 1) (H 0)) (H 1)
--    λ> renombraArbol (H False)
--    H 0
--    λ> renombraArbol (H True)
--    H 0
-- ----------------------------------------------------------------------

{-# LANGUAGE DeriveFoldable, DeriveFunctor #-}
{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Renombra_arbol where

import Data.Map.Strict (Map, (!), fromList)
import Data.List (nub, sort, elemIndex)
import Data.Maybe (fromJust)
import Data.Foldable (toList)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

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

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

-- 1ª solución
-- ===========

renombraArbol1 :: Ord t => Arbol t -> Arbol Int
renombraArbol1 a = aux a
  where ys            = valores a
        aux (H x)     = H (posicion x ys)
        aux (N x i d) = N (posicion x ys) (aux i) (aux d)

-- (valores a) es la lista de los valores en los nodos y las hojas del
```

```haskell
-- árbol a. Por ejemplo,
--    valores ej1  ==  ["A","B","C"]
valores :: Ord a => Arbol a -> [a]
valores a = sort (nub (aux a))
    where aux (H x)     = [x]
          aux (N x i d) = x : (aux i ++ aux d)

-- (posicion x ys) es la posición de x en ys. Por ejemplo.
--    posicion 7 [5,3,7,8]  ==  2
posicion :: Ord a => a -> [a] -> Int
posicion x ys =
  head [n | (y,n) <- zip ys [0..], y == x]

-- 2ª solución
-- ===========

renombraArbol2 :: Ord t => Arbol t -> Arbol Int
renombraArbol2 a = aux a
  where ys = valores a
        aux (H x)     = H (posicion2 x ys)
        aux (N x i d) = N (posicion2 x ys) (aux i) (aux d)

posicion2 :: Ord a => a -> [a] -> Int
posicion2 x ys =
  fromJust (elemIndex x ys)

-- 3ª solución
-- ===========

renombraArbol3 :: Ord t => Arbol t -> Arbol Int
renombraArbol3 a = aux a
  where
    ys = sort (nub (toList a))
    aux (H x)     = H (fromJust (elemIndex x ys))
    aux (N x i d) = N (fromJust (elemIndex x ys)) (aux i) (aux d)

-- 4ª solución
-- ===========

renombraArbol4 :: Ord t => Arbol t -> Arbol Int
renombraArbol4 a = fmap convertir a
  where
    indice = zip (sort (nub (toList a))) [0..]
    convertir x = fromJust (lookup x indice)
```

```
-- 5ª solución
-- ===========

-- (dicValores a) es el diccionario de los valores en los nodos y las
-- hojas del árbol a. Por ejemplo,
--    λ> dicValores ej1
--    fromList [("A",0),("B",1),("C",2)]
dicValores :: Ord a => Arbol a -> Map a Int
dicValores a =
  fromList $ zip (valores a) [0..]

renombraArbol5 :: Ord t => Arbol t -> Arbol Int
renombraArbol5 a =
  repl a
  where
    dic = dicValores a
    repl (H x)     = H (dic ! x)
    repl (N x i d) = N (dic ! x) (repl i) (repl d)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Arbol String -> Arbol Int) -> Spec
specG renombraArbol = do
  it "e1" $
    show (renombraArbol ej1) `shouldBe`
    "N 2 (N 1 (H 0) (H 1)) (N 0 (H 1) (H 2))"

spec :: Spec
spec = do
  describe "def. 1" $ specG renombraArbol1
  describe "def. 2" $ specG renombraArbol2
  describe "def. 3" $ specG renombraArbol3
  describe "def. 4" $ specG renombraArbol4
  describe "def. 5" $ specG renombraArbol5

-- La verificación es
--    λ> verifica
--    5 examples, 0 failures

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

```haskell
-- (genArbol n) genera un árbol aleatorio de orden n. Por ejemplo,
--    λ> generate (genArbol 3 :: Gen (Arbol Int))
--    N (-13) (H 1) (N 30 (H (-10)) (H (-1)))
--    λ> generate (genArbol 3 :: Gen (Arbol Int))
--    N (-3) (H (-29)) (N (-17) (H 8) (H 28))
genArbol :: Arbitrary a => Int -> Gen (Arbol a)
genArbol 0 = H <$> arbitrary
genArbol n = frequency
  [ (1, H <$> arbitrary)
  , (3, N <$> arbitrary <*> sub <*> sub)]
  where
    sub = genArbol (n `div` 2)

-- Arbol es subclase de Arbitraria
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized genArbol

-- La propiedad es
prop_renombraArbol :: Arbol Int -> Bool
prop_renombraArbol a =
  all (== renombraArbol1 a)
      [renombraArbol2 a,
       renombraArbol3 a,
       renombraArbol4 a,
       renombraArbol5 a]

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

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

-- (arbol n) es el árbol completo de profundidad n. Por ejemplo,
--    λ> arbol 2
--    N 12 (N 11 (H 0) (H 0)) (N 11 (H 0) (H 0))
--    λ> renombraArbol1 (arbol 2)
--    N 2 (N 1 (H 0) (H 0)) (N 1 (H 0) (H 0))
arbol :: Int -> Arbol Int
arbol 0 = H 0
arbol n = N (n+10) (arbol (n-1)) (arbol (n-1))

-- La comparación es
--    λ> length (renombraArbol1 (arbol 20))
```

```
--    2097151
--    (2.13 secs, 1,191,782,032 bytes)
--    λ> length (renombraArbol2 (arbol 20))
--    2097151
--    (2.26 secs, 1,191,782,056 bytes)
--    λ> length (renombraArbol3 (arbol 20))
--    2097151
--    (2.16 secs, 1,225,336,576 bytes)
--    λ> length (renombraArbol4 (arbol 20))
--    2097151
--    (1.96 secs, 1,032,398,632 bytes)
--    λ> length (renombraArbol5 (arbol 20))
--    2097151
--    (2.00 secs, 1,191,782,056 bytes)
```

Ejercicio 55

Divide si todos son múltiplos

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    divideSiTodosMultiplos :: Integral a => a -> [a] -> Maybe [a]
-- tal que (divideSiTodosMultiplos x ys) es justo la lista de los
-- cocientes de los elementos de ys entre x si todos son múltiplos de x
-- y Nothing en caso contrario (donde x es distinto de cero). Por ejemplo,
--    divideSiTodosMultiplos 2 [6,10,4]  ==  Just [3,5,2]
--    divideSiTodosMultiplos 2 [6,10,5]  ==  Nothing
--
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Divide_si_todos_multiplos where

import Data.Maybe (isNothing, fromJust)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

divideSiTodosMultiplos1 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos1 x ys
  | todosMultiplos x ys = Just [y `div` x | y <- ys]
  | otherwise           = Nothing

-- (todosMultiplos x ys) se verifica si todos los elementos de ys son
-- múltiplos de x. Por ejemplo,
--    todosMultiplos 2 [6,10,4]  ==  True
--    todosMultiplos 2 [6,10,5]  ==  False
todosMultiplos :: Integral a => a -> [a] -> Bool
```

```haskell
todosMultiplos x ys =
  and [y `mod` x == 0 | y <- ys]

-- 2ª solución
-- ===========

divideSiTodosMultiplos2 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos2 x ys
  | todosMultiplos2 x ys = Just [y `div` x | y <- ys]
  | otherwise            = Nothing

todosMultiplos2 :: Integral a => a -> [a] -> Bool
todosMultiplos2 x =
  all (\y -> y `mod` x == 0)

-- 3ª solución
-- ===========

divideSiTodosMultiplos3 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos3 0 _  = Nothing
divideSiTodosMultiplos3 _ [] = Just []
divideSiTodosMultiplos3 x (y:ys)
  | y `mod` x /= 0  = Nothing
  | isNothing aux   = Nothing
  | otherwise       = Just ((y `div` x) : fromJust aux)
  where aux = divideSiTodosMultiplos3 x ys

-- 4ª solución
-- ===========

divideSiTodosMultiplos4 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos4 0 _  = Nothing
divideSiTodosMultiplos4 _ [] = Just []
divideSiTodosMultiplos4 x (y:ys)
  | y `mod` x /= 0        = Nothing
  | otherwise = case divideSiTodosMultiplos4 x ys of
                  Nothing   -> Nothing
                  Just qs   -> Just (y `div` x : qs)

-- 5ª solución
-- ===========

divideSiTodosMultiplos5 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos5 x ys =
  sequence (map (x `divide`) ys)
```

```haskell
-- (divide x y) es justo el cociente de x entre y, si x es divisible por
-- y y Nothing, en caso contrario. Por ejemplo,
divide :: Integral a => a -> a -> Maybe a
divide x y
  | y `mod` x == 0 = Just (y `div` x)
  | otherwise      = Nothing

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

-- 6ª solución
-- ===========

divideSiTodosMultiplos6 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos6 x =
  sequence . map (x `divide`)

-- 7ª solución
-- ===========

divideSiTodosMultiplos7 :: Integral a => a -> [a] -> Maybe [a]
divideSiTodosMultiplos7 x =
  mapM (x `divide`)

-- Nota. En la solución anterior se usa la función mapM ya que
--    mapM f
-- es equivalente a
--    sequence . map f
-- Por ejemplo,
--    λ> mapM (\n -> if even n then Just (2*n) else Nothing) [4,6,10]
--    Just [8,12,20]
--    λ> mapM (\n -> if even n then Just (2*n) else Nothing) [4,6,11]
--    Nothing

-- Verificación
-- ============
```

```haskell
verifica :: IO ()
verifica = hspec spec

specG :: (Int -> [Int] -> Maybe [Int]) -> Spec
specG divideSiTodosMultiplos = do
  it "e1" $
    divideSiTodosMultiplos 2 [6,10,4]  `shouldBe`  Just [3,5,2]
  it "e2" $
    divideSiTodosMultiplos 2 [6,10,5]  `shouldBe`  Nothing

spec :: Spec
spec = do
  describe "def. 1" $ specG divideSiTodosMultiplos1
  describe "def. 2" $ specG divideSiTodosMultiplos2
  describe "def. 3" $ specG divideSiTodosMultiplos3
  describe "def. 4" $ specG divideSiTodosMultiplos4
  describe "def. 5" $ specG divideSiTodosMultiplos5
  describe "def. 6" $ specG divideSiTodosMultiplos6
  describe "def. 7" $ specG divideSiTodosMultiplos7

-- La verificación es
--    λ> verifica
--    21 examples, 0 failures

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

-- La propiedad es
prop_divideSiTodosMultiplos :: NonZero Int -> [Int] -> Bool
prop_divideSiTodosMultiplos x' ys =
  all (== divideSiTodosMultiplos1 x ys)
      [divideSiTodosMultiplos2 x ys,
       divideSiTodosMultiplos3 x ys,
       divideSiTodosMultiplos4 x ys,
       divideSiTodosMultiplos5 x ys,
       divideSiTodosMultiplos6 x ys]
  where x = getNonZero x'

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

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

```
-- La comparación es
--    λ> last <$> (divideSiTodosMultiplos1 2 [2,4..10^6])
--    Just 500000
--    (0.40 secs, 284,605,936 bytes)
--    λ> last <$> (divideSiTodosMultiplos2 2 [2,4..10^6])
--    Just 500000
--    (0.32 secs, 212,605,792 bytes)
--    λ> last <$> (divideSiTodosMultiplos3 2 [2,4..10^6])
--    Just 500000
--    (1.30 secs, 595,295,824 bytes)
--    λ> last <$> (divideSiTodosMultiplos4 2 [2,4..10^6])
--    Just 500000
--    (0.98 secs, 462,908,920 bytes)
--    λ> last <$> (divideSiTodosMultiplos5 2 [2,4..10^6])
--    Just 500000
--    (0.73 secs, 405,609,672 bytes)
--    λ> last <$> (divideSiTodosMultiplos6 2 [2,4..10^6])
--    Just 500000
--    (0.81 secs, 405,609,800 bytes)
--    λ> last <$> (divideSiTodosMultiplos7 2 [2,4..10^6])
--    Just 500000
--    (0.72 secs, 377,609,624 bytes)
```

Ejercicio 56

Ventana deslizante

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    ventanas :: Int -> Int -> [a] -> [[a]]
-- tal que (ventanas x y zs) es la lista de ventanas de zs de tamaño x
-- y deslizamiento y; es decir listas de x elementos consecutivos de zs
-- (salvo, posiblemente, la última que puede ser menor) tales que la
-- diferencia de posiciones entre los primeros elementos de ventanas
-- consecutivas es y. Por ejemplo,
--    ventanas 3 2 [5,1,9,2] == [[5,1,9],[9,2]]
--    ventanas 3 3 [5,1,9,2] == [[5,1,9],[2]]
--    ventanas 3 4 [5,1,9,2] == [[5,1,9]]
--    ventanas 4 1 [1..7]    == [[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7]]
--    ventanas 4 2 [1..7]    == [[1,2,3,4],[3,4,5,6],[5,6,7]]
--    ventanas 4 3 [1..7]    == [[1,2,3,4],[4,5,6,7]]
--    ventanas 4 4 [1..7]    == [[1,2,3,4],[5,6,7]]
--    ventanas 4 5 [1..7]    == [[1,2,3,4],[6,7]]
--    ventanas 4 6 [1..7]    == [[1,2,3,4],[7]]
--    ventanas 4 7 [1..7]    == [[1,2,3,4]]
--    ventanas 4 8 [1..7]    == [[1,2,3,4]]
--    ventanas 3 2 "abcdef"  == ["abc","cde","ef"]
--    ventanas 3 3 "abcdef"  == ["abc","def"]
--    ventanas 3 4 "abcdef"  == ["abc","ef"]
--    ventanas 3 5 "abcdef"  == ["abc","f"]
--    ventanas 3 6 "abcdef"  == ["abc"]
--    ventanas 3 7 "abcdef"  == ["abc"]
--    ventanas 1 5 "abcdef"  == ["a","f"]
-- ----------------------------------------------------------------------

module Ventana_deslizante where

import Data.List (unfoldr)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
```

```haskell
import Test.QuickCheck

-- 1º solución
-- ===========

ventanas1 :: Int -> Int -> [a] -> [[a]]
ventanas1 _ _ [] = []
ventanas1 x y zs
  | length zs <= x = [zs]
  | otherwise      = take x zs : ventanas1 x y (drop y zs)

-- 2ª solución
-- ===========

ventanas2 :: Int -> Int -> [a] -> [[a]]
ventanas2 x y zs = aux (length zs) zs
  where
    aux _ [] = []
    aux n zs'
      | n <= x    = [zs']
      | otherwise = take x zs' : aux (n - y) (drop y zs')

-- 3ª solución
-- ===========

ventanas3 :: Int -> Int -> [a] -> [[a]]
ventanas3 x y = unfoldr aux
  where aux [] = Nothing
        aux xs = Just (ys,zs)
          where (ys,us)        = splitAt x xs
                zs | null us   = []
                   | otherwise = drop y xs

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Int -> Int -> [Int] -> [[Int]]) -> Spec
specG ventanas = do
  it "e1" $
    ventanas 4 1 [1..7]    `shouldBe` [[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7]]
  it "e2" $
    ventanas 4 2 [1..7]    `shouldBe` [[1,2,3,4],[3,4,5,6],[5,6,7]]
```

```
  it "e3" $
    ventanas 4 3 [1..7]    `shouldBe` [[1,2,3,4],[4,5,6,7]]
  it "e4" $
    ventanas 4 4 [1..7]    `shouldBe` [[1,2,3,4],[5,6,7]]
  it "e5" $
    ventanas 4 5 [1..7]    `shouldBe` [[1,2,3,4],[6,7]]
  it "e6" $
    ventanas 4 6 [1..7]    `shouldBe` [[1,2,3,4],[7]]
  it "e7" $
    ventanas 4 7 [1..7]    `shouldBe` [[1,2,3,4]]
  it "e8" $
    ventanas 4 8 [1..7]    `shouldBe` [[1,2,3,4]]
  it "e9" $
    ventanas 3 2 [5,1,9,2] `shouldBe` [[5,1,9],[9,2]]
  it "e10" $
    ventanas 3 3 [5,1,9,2] `shouldBe` [[5,1,9],[2]]
  it "e11" $
    ventanas 3 4 [5,1,9,2] `shouldBe` [[5,1,9]]

spec :: Spec
spec = do
  describe "def. 1" $ specG ventanas1
  describe "def. 2" $ specG ventanas2
  describe "def. 3" $ specG ventanas3

-- La verificación es
--    λ> verifica
--    21 examples, 0 failures

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

-- La propiedad es
prop_ventanas :: Positive Int -> Positive Int -> [Int] -> Bool
prop_ventanas (Positive x) (Positive y) zs =
  all (== ventanas1 x y zs)
      [ventanas2 x y zs,
       ventanas3 x y zs]

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

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

```
-- La comparación es
--    λ> length (ventanas1 4 6 [1..10^5])
--    16667
--    (3.81 secs, 14,199,776 bytes)
--    λ> length (ventanas2 4 6 [1..10^5])
--    16667
--    (0.05 secs, 14,466,488 bytes)
--    λ> length (ventanas3 4 6 [1..10^5])
--    16667
--    (0.03 secs, 22,333,600 bytes)
--
--    λ> length (ventanas2 4 6 [1..10^7])
--    1666667
--    (1.87 secs, 1,387,268,112 bytes)
--    λ> length (ventanas3 4 6 [1..10^7])
--    1666667
--    (1.18 secs, 2,173,935,176 bytes)
```

Ejercicio 57

Representación de Zeckendorf

```
-- ----------------------------------------------------------------
-- Los primeros números de Fibonacci son
--    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
-- tales que los dos primeros son iguales a 1 y los siguientes se
-- obtienen sumando los dos anteriores.
--
-- El [teorema de Zeckendorf](https://bit.ly/3k5NNt1) establece que todo
-- entero positivo n se puede representar, de manera única, como la suma
-- de números de Fibonacci no consecutivos decrecientes. Dicha suma se
-- llama la representación de Zeckendorf de n. Por ejemplo, la
-- representación de Zeckendorf de 100 es
--    100 = 89 + 8 + 3
-- Hay otras formas de representar 100 como sumas de números de
-- Fibonacci; por ejemplo,
--    100 = 89 +  8 + 2 + 1
--    100 = 55 + 34 + 8 + 3
-- pero no son representaciones de Zeckendorf porque 1 y 2 son números
-- de Fibonacci consecutivos, al igual que 34 y 55.
--
-- Definir la función
--    zeckendorf :: Integer -> [Integer]
-- tal que (zeckendorf n) es la representación de Zeckendorf de n. Por
-- ejemplo,
--    zeckendorf 100 == [89,8,3]
--    zeckendorf 200 == [144,55,1]
--    zeckendorf 300 == [233,55,8,3,1]
--    length (zeckendorf (10^50000)) == 66097
-- ----------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```haskell
module Representacion_de_Zeckendorf where

import Data.List (subsequences)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (Positive (Positive), quickCheck)

-- 1ª solución
-- ===========

zeckendorf1 :: Integer -> [Integer]
zeckendorf1 = head . zeckendorf1Aux

zeckendorf1Aux :: Integer -> [[Integer]]
zeckendorf1Aux n =
  [xs | xs <- subsequences (reverse (takeWhile (<= n) (tail fibs))),
        sum xs == n,
        sinFibonacciConsecutivos xs]

-- fibs es la la sucesión de los números de Fibonacci. Por ejemplo,
--    take 14 fibs  == [1,1,2,3,5,8,13,21,34,55,89,144,233,377]
fibs :: [Integer]
fibs = 1 : scanl (+) 1 fibs

-- (sinFibonacciConsecutivos xs) se verifica si en la sucesión
-- decreciente de número de Fibonacci xs no hay dos consecutivos. Por
-- ejemplo,
--    sinFibonacciConsecutivos [89, 8, 3]      ==  True
--    sinFibonacciConsecutivos [55, 34, 8, 3]  ==  False
sinFibonacciConsecutivos :: [Integer] -> Bool
sinFibonacciConsecutivos xs =
  and [x /= siguienteFibonacci y | (x,y) <- zip xs (tail xs)]

-- (siguienteFibonacci n) es el menor número de Fibonacci mayor que
-- n. Por ejemplo,
--    siguienteFibonacci 34  ==  55
siguienteFibonacci :: Integer -> Integer
siguienteFibonacci n =
  head (dropWhile (<= n) fibs)

-- 2ª solución
-- ===========

zeckendorf2 :: Integer -> [Integer]
zeckendorf2 = head . zeckendorf2Aux
```

```haskell
zeckendorf2Aux :: Integer -> [[Integer]]
zeckendorf2Aux n = map reverse (aux n (tail fibs))
  where aux 0 _ = [[]]
        aux m (x:y:zs)
            | x <= m     = [x:xs | xs <- aux (m-x) zs] ++ aux m (y:zs)
            | otherwise  = []

-- 3ª solución
-- ===========

zeckendorf3 :: Integer -> [Integer]
zeckendorf3 0 = []
zeckendorf3 n = x : zeckendorf3 (n - x)
  where x = last (takeWhile (<= n) fibs)

-- 4ª solución
-- ===========

zeckendorf4 :: Integer -> [Integer]
zeckendorf4 n = aux n (reverse (takeWhile (<= n) fibs))
  where aux 0 _      = []
        aux m (x:xs) = x : aux (m-x) (dropWhile (>m-x) xs)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Integer -> [Integer]) -> Spec
specG zeckendorf = do
  it "e1" $
    zeckendorf 100 == [89,8,3]
  it "e2" $
    zeckendorf 200 == [144,55,1]
  it "e3" $
    zeckendorf 300 == [233,55,8,3,1]

spec :: Spec
spec = do
  describe "def. 1" $ specG zeckendorf1
  describe "def. 2" $ specG zeckendorf2
  describe "def. 3" $ specG zeckendorf3
  describe "def. 4" $ specG zeckendorf4
```

```
-- La verificación es
--    λ> verifica
--    12 examples, 0 failures

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

-- La propiedad es
prop_zeckendorf_equiv :: Positive Integer -> Bool
prop_zeckendorf_equiv (Positive n) =
  all (== zeckendorf1 n)
      [zeckendorf2 n,
       zeckendorf3 n,
       zeckendorf4 n]

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

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

-- La comparación es
--    λ> zeckendorf1 (7*10^4)
--    [46368,17711,4181,1597,89,34,13,5,2]
--    (1.49 secs, 2,380,707,744 bytes)
--    λ> zeckendorf2 (7*10^4)
--    [46368,17711,4181,1597,89,34,13,5,2]
--    (0.07 secs, 21,532,008 bytes)
--
--    λ> zeckendorf2 (10^6)
--    [832040,121393,46368,144,55]
--    (1.40 secs, 762,413,432 bytes)
--    λ> zeckendorf3 (10^6)
--    [832040,121393,46368,144,55]
--    (0.01 secs, 542,488 bytes)
--    λ> zeckendorf4 (10^6)
--    [832040,121393,46368,144,55]
--    (0.01 secs, 536,424 bytes)
--
--    λ> length (zeckendorf3 (10^3000))
--    3947
--    (3.02 secs, 1,611,966,408 bytes)
--    λ> length (zeckendorf4 (10^2000))
--    2611
```

```
--    (0.02 secs, 10,434,336 bytes)
--
--    λ> length (zeckendorf4 (10^50000))
--    66097
--    (2.84 secs, 3,976,483,760 bytes)

-- ---------------------------------------------------------------------
-- § Referencias                                                       --
-- ---------------------------------------------------------------------

-- Este ejercicio se basa en el problema
-- [Zeckendorf representation](http://bit.ly/VB4yz6)
-- de [Programming Praxis](http://programmingpraxis.com).
--
-- La representación de Zeckendorf se describe en el artículo de la
-- Wikipedia [Zeckendorf's theorem](http://bit.ly/VB2pU3).
```

Ejercicio 58

Elemento más cercano que cumple una propiedad

```
-- ---------------------------------------------------------------------
-- El código Morse es un sistema de representación de letras y números
-- mediante señales emitidas de forma intermitente.
--
-- A los signos (letras mayúsculas o dígitos) se le asigna un código
-- como se muestra a continuación
--    |---+-------|---+-------|---+-------|---+-------|
--    | A | .-    | J | .---  | S | ...   | 1 | ..--- |
--    | B | -...  | K | -.-   | T | -     | 2 | ...-- |
--    | C | -.-.  | L | .-..  | U | ..-   | 3 | ....- |
--    | D | -..   | M | --    | V | ...-  | 4 | ..... |
--    | E | .     | N | -.    | W | .--   | 5 | -.... |
--    | F | ..-.  | O | ---   | X | -..-  | 6 | --... |
--    | G | --.   | P | .--.  | Y | -.--  | 7 | ---.. |
--    | H | ....  | Q | --.-  | Z | --..  | 8 | ----. |
--    | I | ..    | R | .-.   | 0 | .---- | 9 | ----- |
--    |---+-------|---+-------|---+-------|---+-------|
--
-- El código Morse de las palabras se obtiene a partir del de sus
-- caracteres insertando un espacio entre cada uno. Por ejemplo, el
-- código de "todo" es "- --- -.. ---"
--
-- El código Morse de las frase se obtiene a partir del de sus
-- palabras insertando un espacio entre cada uno. Por ejemplo, el
-- código de "todo o nada" es "- --- -.. ---  ---  -. .- -.. .-"
--
-- Definir las funciones
--    fraseAmorse :: String -> String
--    morseAfrase :: String -> String
```

```
-- tales que
-- + (fraseAmorse cs) es la traducción de la frase cs a Morse. Por
--   ejemplo,
--      λ> fraseAmorse "En todo la medida"
--      ". -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-"
-- + (morseAfrase cs) es la frase cuya traducción a Morse es cs. Por
--   ejemplo,
--      λ> morseAfrase ". -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-"
--      "EN TODO LA MEDIDA"
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Codigo_Morse where

import Data.Char (toUpper, isAlphaNum)
import Data.List (intercalate)
import Data.List.Split (splitOn)
import Data.Maybe (fromJust)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

-- caracteres es la lista ordenada de las caracteres (letras mayúsculas
-- y dígitos) que se usan en los mensajes Morse.
caracteres :: [Char]
caracteres = ['A'..'Z'] ++ ['0'..'9']

-- morse es la lista de los códigos Morse correspondientes a la lista
-- de caracteres.
morse :: [String]
morse = [".-","-...","-.-.","-..",".","..-.","--.","....","..",".---",
         "-.-",".-..","--","-.","---",".--.","--.-",".-.","...","-",
         "..-","...-",".--","-..-","-.--","--..",".----","..---","...--",
         "....-",".....","-....","--...","---..","----.","-----"]

-- (correspondiente xs ys x) es el elemento de ys en la misma posición
-- que x en xs. Por ejemplo,
--    correspondiente [1..10] [2,4..20] 3  ==  6
correspondiente :: Ord a => [a] -> [b] -> a -> b
correspondiente xs ys x =
  head [y | (z,y) <- zip xs ys, z == x]
```

```haskell
-- (caracterAmorse x) es el código Morse correspondiente al carácter
-- x. Por ejemplo,
--    caracterAmorse 'A'  ==  ".-"
--    caracterAmorse 'B'  ==  "-..."
--    caracterAmorse '1'  ==  "..---"
--    caracterAmorse 'a'  ==  ".-"
caracterAmorse :: Char -> String
caracterAmorse =
  correspondiente caracteres morse . toUpper

-- (morseAcaracter x) es el carácter cuyo código Morse es x. Por
-- ejemplo,
--    morseAcaracter ".-"     ==  'A'
--    morseAcaracter "-..."   ==  'B'
--    morseAcaracter "..---"  ==  '1'
morseAcaracter :: String -> Char
morseAcaracter =
  correspondiente morse caracteres

-- (palabraAmorse cs) es el código Morse correspondiente a la palabra
-- cs. Por ejemplo,
--    palabraAmorse "En"  ==  ". -."
palabraAmorse :: [Char] -> String
palabraAmorse = unwords . map caracterAmorse

-- (morseApalabra cs) es la palabra cuyo traducción a Morse es cs. Por
-- ejemplo,
--    morseApalabra ". -."  ==  "EN"
morseApalabra :: String -> [Char]
morseApalabra = map morseAcaracter . words

-- (fraseAmorse cs) es la traducción de la frase cs a Morse. Por ejemplo,
--    λ> fraseAmorse "En todo la medida"
--    ". -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-"
fraseAmorse1 :: String -> String
fraseAmorse1 = intercalate "  " . map palabraAmorse . words

-- Ejemplo de cálculo
--    fraseAmorse "En todo la medida"
--    = (intercalate "  " . map palabraAmorse . words)
--      "En todo la medida"
--    = (intercalate "  " . map palabraAmorse)
--      ["En","todo","la","medida"]
--    = intercalate "  " [". -.","- --- -.. ---",".-.. .-","-- . -.. .. -.. .-"]
--    = ". -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-"
```

```haskell
-- (morseAfrase cs) es la frase cuya traducción a Morse es cs. Por
-- ejemplo,
--    λ> morseAfrase ”. -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-”
--    ”EN TODO LA MEDIDA”
morseAfrase1 :: String -> String
morseAfrase1 = unwords . map morseApalabra . splitOn ”  ”

-- Ejemplo de cálculo
--    morseAfrase ”. -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-”
--    = (unwords . map morseApalabra)
--      ”. -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-”
--    = (unwords . map morseApalabra)
--      [”. -.”,”- --- -.. ---”,”.-.. .-”,”-- . -.. .. -.. .-”]
--    = unwords [”EN”,”TODO”,”LA”,”MEDIDA”]
--    = ”EN TODO LA MEDIDA”

-- 2ª solución
-- ===========

-- Diccionario de Morse
diccionarioMorse :: [(Char, String)]
diccionarioMorse =
  zip (['A'..'Z'] ++ ['0'..'9'])
      [”.-”,”-...”,”-.-.”,”-..”,”.”,”..-.”,”--.”,”....”,”..”,”.---”,
       ”-.-”,”.-..”,”--”,”-.”,”---”,”.--.”,”--.-”,”.-.”,”...”,”-”,
       ”..-”,”...-”,”.--”,”-..-”,”-.--”,”--..”,”.----”,”..---”,”...--”,
       ”....-”,”.....”,”-....”,”--...”,”---..”,”----.”,”-----”]

fraseAmorse2 :: String -> String
fraseAmorse2 =
  intercalate ”  ”
  . map (intercalate ” ”
  . map caracterAmorse2)
  . words
  . map toUpper

caracterAmorse2 :: Char -> String
caracterAmorse2 c =
  fromJust (lookup c diccionarioMorse)

morseAfrase2 :: String -> String
morseAfrase2 =
  unwords
  . map (map morseAcaracter2 . words)
```

```
  . splitOn "  "

morseAcaracter2 :: String -> Char
morseAcaracter2 m =
  fromJust (lookup m (map (\(a,b) -> (b,a)) diccionarioMorse))

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG1 :: (String -> String) -> Spec
specG1 fraseAmorse = do
  it "e1" $
    fraseAmorse "En todo la medida" `shouldBe`
    ". -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-"

specG2 :: (String -> String) -> Spec
specG2 morseAfrase = do
  it "e1" $
    morseAfrase ". -.  - --- -.. ---  .-.. .-  -- . -.. .. -.. .-"
    `shouldBe` "EN TODO LA MEDIDA"

spec :: Spec
spec = do
  describe "def. 1" $ specG1 fraseAmorse1
  describe "def. 2" $ specG1 fraseAmorse2
  describe "def. 1" $ specG2 morseAfrase1
  describe "def. 2" $ specG2 morseAfrase2

-- La verificación es
--    λ> verifica
--    21 examples, 0 failures

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

-- La propiedad es
prop_equivalencia :: Property
prop_equivalencia =
  forAll (listOf (elements (['A'..'Z'] ++ ['0'..'9'] ++ " "))) $ \xs ->
  let ys = fraseAmorse1 xs in
  fraseAmorse2 xs == ys && morseAfrase1 ys == morseAfrase2 ys
```

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

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

-- La comparación es
--    λ> length (fraseAmorse1 (take (3*10^6) (cycle "ABC ")))
--    10499998
--    (2.03 secs, 3,318,602,592 bytes)
--    λ> length (fraseAmorse2 (take (3*10^6) (cycle "ABC ")))
--    10499998
--    (0.93 secs, 2,694,602,584 bytes)
--
--    λ> length (morseAfrase1 ejemplo)
--    2999999
--    (4.04 secs, 6,606,601,512 bytes)
--    λ> length (morseAfrase2 ejemplo)
--    2999999
--    (0.79 secs, 3,288,600,112 bytes)
```

Ejercicio 59

Producto cartesiano de una familia de conjuntos

```
-- ----------------------------------------------------------------------
-- Definir la función
--    producto :: [[a]] -> [[a]]
-- tal que (producto xss) es el producto cartesiano de los conjuntos xss.
-- Por ejemplo,
--    λ> producto [[2,5],[6,4]]
--    [[2,6],[2,4],[5,6],[5,4]]
--    λ> producto [[1,3],[2,5],[6,4]]
--    [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
--    λ> producto [[1,3,5],[2,4]]
--    [[1,2],[1,4],[3,2],[3,4],[5,2],[5,4]]
--    λ> producto []
--    [[]]
--
-- Comprobar con QuickCheck que para toda lista de listas de números
-- enteros, xss, se verifica que el número de elementos de (producto
-- xss) es igual al producto de los números de elementos de cada una de
-- las listas de xss.
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Producto_cartesiano where

import Control.Monad (liftM2)
import Control.Applicative (liftA2)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)
```

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

producto1 :: [[a]] -> [[a]]
producto1 []       = [[]]
producto1 (xs:xss) = [x:ys | x <- xs, ys <- producto1 xss]

-- 2ª solución
-- ===========

producto2 :: [[a]] -> [[a]]
producto2 []       = [[]]
producto2 (xs:xss) = [x:ys | x <- xs, ys <- ps]
  where ps = producto2 xss

-- 3ª solución
-- ===========

producto3 :: [[a]] -> [[a]]
producto3 []       = [[]]
producto3 (xs:xss) = inserta3 xs (producto3 xss)

-- (inserta xs xss) inserta cada elemento de xs en los elementos de
-- xss. Por ejemplo,
--    λ> inserta [1,2] [[3,4],[5,6]]
--    [[1,3,4],[1,5,6],[2,3,4],[2,5,6]]
inserta3 :: [a] -> [[a]] -> [[a]]
inserta3 [] _       = []
inserta3 (x:xs) yss = [x:ys | ys <- yss] ++ inserta3 xs yss

-- 4ª solución
-- ===========

producto4 :: [[a]] -> [[a]]
producto4 = foldr inserta4 [[]]

inserta4 :: [a] -> [[a]] -> [[a]]
inserta4 []     _   = []
inserta4 (x:xs) yss = map (x:) yss ++ inserta4 xs yss

-- 5ª solución
-- ===========

producto5 :: [[a]] -> [[a]]
producto5 = foldr inserta5 [[]]
```

```haskell
inserta5 :: [a] -> [[a]] -> [[a]]
inserta5 xs yss = [x:ys | x <- xs, ys <- yss]

-- 6ª solución
-- ===========

producto6 :: [[a]] -> [[a]]
producto6 = foldr inserta6 [[]]

inserta6 :: [a] -> [[a]] -> [[a]]
inserta6 xs yss = concatMap (\x -> map (x:) yss) xs

-- 7ª solución
-- ===========

producto7 :: [[a]] -> [[a]]
producto7 = foldr inserta7 [[]]

inserta7 :: [a] -> [[a]] -> [[a]]
inserta7 xs yss = xs >>= (\x -> map (x:) yss)

-- 8ª solución
-- ===========

producto8 :: [[a]] -> [[a]]
producto8 = foldr inserta8 [[]]

inserta8 :: [a] -> [[a]] -> [[a]]
inserta8 xs yss = (:) <$> xs <*> yss

-- 9ª solución
-- ===========

producto9 :: [[a]] -> [[a]]
producto9 = foldr inserta9 [[]]

inserta9 :: [a] -> [[a]] -> [[a]]
inserta9 = liftA2 (:)

-- 10ª solución
-- ============

producto10 :: [[a]] -> [[a]]
producto10 = foldr (liftM2 (:)) [[]]
```

```haskell
-- 11ª solución
-- ============

producto11 :: [[a]] -> [[a]]
producto11 = sequence

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([[Int]] -> [[Int]]) -> Spec
specG producto = do
  it "e1" $
    producto [[1,3],[2,5]]
    `shouldBe` [[1,2],[1,5],[3,2],[3,5]]
  it "e2" $
    producto [[1,3],[2,5],[6,4]]
    `shouldBe` [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
  it "e3" $
    producto [[1,3,5],[2,4]]
    `shouldBe` [[1,2],[1,4],[3,2],[3,4],[5,2],[5,4]]

spec :: Spec
spec = do
  describe "def. 1" $ specG producto1
  describe "def. 2" $ specG producto2
  describe "def. 3" $ specG producto3
  describe "def. 4" $ specG producto4
  describe "def. 5" $ specG producto5
  describe "def. 6" $ specG producto6
  describe "def. 7" $ specG producto7
  describe "def. 8" $ specG producto8
  describe "def. 9" $ specG producto9
  describe "def. 10" $ specG producto10
  describe "def. 11" $ specG producto11

-- La verificación es
--    λ> verifica
--    33 examples, 0 failures

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

```haskell
-- La propiedad es
prop_producto :: [[Int]] -> Bool
prop_producto xss =
  all (== producto1 xss)
      [ producto2 xss
      , producto3 xss
      , producto4 xss
      , producto5 xss
      , producto6 xss
      , producto7 xss
      , producto8 xss
      , producto9 xss
      , producto10 xss
      , producto11 xss
      ]

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

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

-- La comparación es
--    λ> length (producto1 (replicate 7 [0..9]))
--    10000000
--    (10.04 secs, 10,507,268,856 bytes)
--    λ> length (producto2 (replicate 7 [0..9]))
--    10000000
--    (1.71 secs, 1,333,943,632 bytes)
--    λ> length (producto3 (replicate 7 [0..9]))
--    10000000
--    (2.94 secs, 1,956,176,072 bytes)
--    λ> length (producto4 (replicate 7 [0..9]))
--    10000000
--    (1.06 secs, 1,600,616,296 bytes)
--    λ> length (producto5 (replicate 7 [0..9]))
--    10000000
--    (1.77 secs, 1,333,943,248 bytes)
--    λ> length (producto6 (replicate 7 [0..9]))
--    10000000
--    (1.06 secs, 1,600,608,064 bytes)
--    λ> length (producto7 (replicate 7 [0..9]))
--    10000000
```

```
--    (0.34 secs, 1,600,607,784 bytes)
--    λ> length (producto8 (replicate 7 [0..9]))
--    10000000
--    (1.03 secs, 978,390,888 bytes)
--    λ> length (producto9 (replicate 7 [0..9]))
--    10000000
--    (1.20 secs, 1,067,273,920 bytes)
--    λ> length (producto10 (replicate 7 [0..9]))
--    10000000
--    (0.58 secs, 2,311,718,360 bytes)
--    λ> length (producto11 (replicate 7 [0..9]))
--    10000000
--    (1.22 secs, 1,067,273,840 bytes)
--
--    λ> length (producto7 (replicate 7 [1..14]))
--    105413504
--    (3.71 secs, 16,347,812,624 bytes)
--    λ> length (producto10 (replicate 7 [1..14]))
--    105413504
--    (5.12 secs, 23,613,234,792 bytes)
--    λ> length (producto11 (replicate 7 [1..14]))
--    105413504
--    (17.83 secs, 10,898,744,528 bytes)

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

-- La propiedad es
prop_longitud :: [[Int]] -> Bool
prop_longitud xss =
  length (producto7 xss) == product (map length xss)

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

Ejercicio 60

Todas tienen par

```haskell
-- ----------------------------------------------------------------------
-- Definir la función
--    todasTienenPar :: [[Int]] -> Bool
-- tal que tal que (todasTienenPar xss) se verifica si cada elemento de
-- la lista de listas xss contiene algún número par. Por ejemplo,
--    todasTienenPar [[1,2],[3,4,5],[8]]  ==  True
--    todasTienenPar [[1,2],[3,5]]        ==  False
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Todas_tienen_par where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

todasTienenPar1 :: [[Int]] -> Bool
todasTienenPar1 xss =
  and [or [even x | x <- xs] | xs <- xss]

-- 2ª solución
-- ===========

todasTienenPar2 :: [[Int]] -> Bool
todasTienenPar2 []       = True
todasTienenPar2 (xs:xss) = tienePar xs && todasTienenPar2 xss

-- (tienePar xs) se verifica si xs contiene algún número par.
tienePar  :: [Int] -> Bool
```

```haskell
tienePar []     = False
tienePar (x:xs) = even x || tienePar xs

-- 3ª solución
-- ===========

todasTienenPar3 :: [[Int]] -> Bool
todasTienenPar3 = foldr ((&&) . tienePar3) True

-- (tienePar3 xs) se verifica si xs contiene algún número par.
tienePar3  :: [Int] -> Bool
tienePar3 = foldr ((||) . even) False

-- 4ª solución
-- ===========

todasTienenPar4 :: [[Int]] -> Bool
todasTienenPar4 = all (any even)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: ([[Int]] -> Bool) -> Spec
specG todasTienenPar = do
  it "e1" $
    todasTienenPar [[1,2],[3,4,5],[8]]  `shouldBe` True
  it "e2" $
    todasTienenPar [[1,2],[3,5]]        `shouldBe` False

spec :: Spec
spec = do
  describe "def. 1" $ specG todasTienenPar1
  describe "def. 2" $ specG todasTienenPar2
  describe "def. 3" $ specG todasTienenPar3
  describe "def. 4" $ specG todasTienenPar4

-- La verificación es
--    λ> verifica
--    8 examples, 0 failures

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

```haskell
-- La propiedad es
prop_todasTienenPar :: [[Int]] -> Bool
prop_todasTienenPar xss =
  all (== todasTienenPar1 xss)
      [ todasTienenPar2 xss
      , todasTienenPar3 xss
      , todasTienenPar4 xss]

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

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

-- La comparación es
--    λ> todasTienenPar1 (replicate 4000 ([1,3..4000] ++ [0]))
--    True
--    (2.02 secs, 2,691,974,232 bytes)
--    λ> todasTienenPar2 (replicate 4000 ([1,3..4000] ++ [0]))
--    True
--    (2.63 secs, 2,114,663,016 bytes)
--    λ> todasTienenPar3 (replicate 4000 ([1,3..4000] ++ [0]))
--    True
--    (0.65 secs, 1,858,502,376 bytes)
--    λ> todasTienenPar4 (replicate 4000 ([1,3..4000] ++ [0]))
--    True
--    (0.47 secs, 1,794,470,264 bytes)
```

Ejercicio 61

Sucesiones pucelanas

```
-- ---------------------------------------------------------------------
-- En la Olimpiada de Matemática del 2010 se planteó el siguiente
-- problema:
--   Una sucesión pucelana es una sucesión creciente de 16 números
--   impares positivos consecutivos, cuya suma es un cubo perfecto.
--   ¿Cuántas sucesiones pucelanas tienen solamente números de tres
--   cifras?
-- Para resolverlo se propone el siguiente ejercicio.
--
-- Definir la función
--    pucelanasConNcifras :: Integer -> [[Integer]]
-- tal que (pucelanasConNcifras n) es la lista de las sucesiones
-- pucelanas que tienen solamente números de n cifras. Por ejemplo,
--    λ> pucelanasConNcifras 2
--    [[17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47]]
--
-- Calcular cuántas sucesiones pucelanas tienen solamente números de
-- tres cifras.
-- ---------------------------------------------------------------------

module Sucesiones_pucelanas where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)

-- 1ª solución
-- ===========

pucelanasConNcifras1 :: Integer -> [[Integer]]
pucelanasConNcifras1 n =
  [[x,x+2..x+30] | x <- [10^(n-1)+1..10^n-31],
                   esCubo (sum [x,x+2..x+30])]
```

```haskell
-- (esCubo n) se verifica si n es un cubo. Por ejemplo,
--    esCubo 27  ==  True
--    esCubo 28  ==  False
esCubo :: Integer -> Bool
esCubo x = y^3 == x
  where y = ceiling (fromIntegral x ** (1/3))

-- 2ª solución
-- ===========

pucelanasConNcifras2 :: Integer -> [[Integer]]
pucelanasConNcifras2 n =
  [[x,x+2..x+30] | x <- [10^(n-1)+1..10^n-31],
                   esCubo (16 * fromIntegral x + 240)]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Integer -> [[Integer]]) -> Spec
specG pucelanasConNcifras = do
  it "e1" $
    pucelanasConNcifras 2 `shouldBe`
    [[17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47]]

spec :: Spec
spec = do
  describe "def. 1" $ specG pucelanasConNcifras1
  describe "def. 2" $ specG pucelanasConNcifras2

-- La verificación es
--    λ> verifica
--    3 examples, 0 failures

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

-- La propiedad es
prop_pucelanasConNcifras :: Integer -> Bool
prop_pucelanasConNcifras m =
  and [pucelanasConNcifras1 n == pucelanasConNcifras2 n
      | n <- [1..m]]
```

```
-- La comprobación es
--    λ> prop_pucelanasConNcifras 6
--    True

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

-- La comparación es
--    λ> head (head (pucelanasConNcifras1 10))
--    1000187985
--    (0.51 secs, 590,126,640 bytes)
--    λ> head (head (pucelanasConNcifras2 10))
--    1000187985
--    (0.39 secs, 286,342,880 bytes)

-- Cálculo
-- =======

-- El cálculo es
--    λ> length (pucelanasConNcifras1 3)
--    3
```

Ejercicio 62

Producto de matrices como listas de listas

```haskell
-- ----------------------------------------------------------------------
-- Las matrices pueden representarse mediante una lista de listas donde
-- cada una de las lista representa una fila  de la matriz. Por ejemplo,
-- la matriz
--    |1 0 -2|
--    |0 3 -1|
-- puede representarse por [[1,0,-2],[0,3,-1]].
--
-- Definir la función
--    producto :: Num a => [[a]] -> [[a]] -> [[a]]
-- tal que (producto p q) es el producto de las matrices p y q. Por
-- ejemplo,
--    λ> producto [[1,0,-2],[0,3,-1]] [[0,3],[-2,-1],[0,4]]
--    [[0,-5],[-6,-7]]
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Producto_de_matrices_como_listas_de_listas where

import Data.List (transpose)
import Data.Matrix (fromLists, toLists)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

producto1 :: Num a => [[a]] -> [[a]] -> [[a]]
```

```haskell
producto1 p q =
  [[sum [x*y | (x,y) <- zip fila col] | col <- cols] | fila <- p]
  where
    cols = transpose q

-- 2ª solución
-- ===========

producto2 :: Num a => [[a]] -> [[a]] -> [[a]]
producto2 p q =
  [[sum (zipWith (*) fila col) | col <- cols] | fila <- p]
  where
    cols = transpose q

-- 3ª solución
-- ===========

producto3 :: Num a => [[a]] -> [[a]] -> [[a]]
producto3 [] _ = []
producto3 (fila:filas) q = map (productoEscalar fila) cols : producto3 filas q
  where
    cols = transpose q

productoEscalar :: Num a => [a] -> [a] -> a
productoEscalar (x:xs) (y:ys) = x * y + productoEscalar xs ys
productoEscalar _      _      = 0

-- 4ª solución
-- ===========

producto4 :: Num a => [[a]] -> [[a]] -> [[a]]
producto4 p q = map (\fila -> map (productoEscalar fila) (transpose q)) p

-- 5ª solución
-- ===========

producto5 :: Num a => [[a]] -> [[a]] -> [[a]]
producto5 p q =
  toLists (fromLists p * fromLists q)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec
```

```
specG :: ([[Integer]] -> [[Integer]] -> [[Integer]]) -> Spec
specG producto = do
  it "e1" $
    producto [[1,0,-2],[0,3,-1]] [[0,3],[-2,-1],[0,4]]
    `shouldBe` [[0,-5],[-6,-7]]

spec :: Spec
spec = do
  describe "def. 1" $ specG producto1
  describe "def. 2" $ specG producto2
  describe "def. 3" $ specG producto3
  describe "def. 4" $ specG producto4
  describe "def. 5" $ specG producto5

-- La verificación es
--    λ> verifica
--    5 examples, 0 failures

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

-- Generador de matrices compatibles
genMatrices :: (Num a, Arbitrary a) => Gen ( [[a]], [[a]] )
genMatrices = do
  i <- choose (1, 5)
  j <- choose (1, 5)
  k <- choose (1, 5)
  p <- vectorOf i (vectorOf j arbitrary)
  q <- vectorOf j (vectorOf k arbitrary)
  return (p, q)

-- La propiedad es
prop_producto :: Property
prop_producto =
  forAll (genMatrices :: Gen ([[Integer]], [[Integer]])) $ \(p, q) ->
  all (== producto1 p q)
      [producto2 p q,
       producto3 p q,
       producto4 p q,
       producto5 p q]

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

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

-- La comparación es
--    λ> ej = replicate 2000 [1..2000]
--    λ> last (last (producto1 ej ej))
--    4002000000
--    (0.23 secs, 774,699,016 bytes)
--    λ> last (last (producto2 ej ej))
--    4002000000
--    (0.20 secs, 774,490,928 bytes)
--    λ> last (last (producto3 ej ej))
--    4002000000
--    (0.20 secs, 774,909,856 bytes)
--    λ> last (last (producto4 ej ej))
--    4002000000
--    (0.20 secs, 774,588,984 bytes)
--    λ> last (last (producto5 ej ej))
--    4002000000
--    (5.18 secs, 4,431,551,416 bytes)
```

Ejercicio 63

Inserción en árboles binarios de búsqueda

```
-- ----------------------------------------------------------------------
-- Un árbol binario de búsqueda (ABB) es un árbol binario tal que el de
-- cada nodo es mayor que los valores de su subárbol izquierdo y es
-- menor que los valores de su subárbol derecho y, además, ambos
-- subárboles son árboles binarios de búsqueda. Por ejemplo, al
-- almacenar los valores de [8,4,2,6,3] en un ABB se puede obtener el
-- siguiente ABB:
--
--       3
--      / \
--     /   \
--    2     6
--         / \
--        4   8
--
-- Los ABB se pueden representar como tipo de dato algebraico:
--    data ABB = V
--      | N Int ABB ABB
--      deriving (Eq, Show)
-- Por ejemplo, la definición del ABB anteriore es
--    ej :: ABB
--    ej = N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
--
-- Definir la función
--    inserta :: Int -> ABB -> ABB
-- tal que (inserta v a) es el árbol obtenido añadiendo el valor v al
-- ABB a, si no es uno de sus valores. Por ejemplo,
--    λ>  inserta 5 ej
--    N 3 (N 2 V V) (N 6 (N 4 V (N 5 V V)) (N 8 V V))
```

```
--    λ>  inserta 1 ej
--    N 3 (N 2 (N 1 V V) V) (N 6 (N 4 V V) (N 8 V V))
--    λ>  inserta 2 ej
--    N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
--
-- Comprobar con QuickCheck que al insertar un valor en un ABB se
-- obtiene otro ABB.
-- ----------------------------------------------------------------------

{-# LANGUAGE FlexibleInstances #-}

module Insercion_en_arboles_binarios_de_busqueda where

import Test.Hspec (Spec, hspec, it, shouldBe)
import Test.QuickCheck

data ABB a = V
           | N a (ABB a) (ABB a)
  deriving (Show, Eq)

ej :: ABB Int
ej = N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))

inserta :: Ord a => a -> ABB a -> ABB a
inserta v' V = N v' V V
inserta v' (N v i d)
  | v' == v   = N v i d
  | v' < v    = N v (inserta v' i) d
  | otherwise = N v i (inserta v' d)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    inserta 5 ej `shouldBe` N 3 (N 2 V V) (N 6 (N 4 V (N 5 V V)) (N 8 V V))
  it "e2" $
    inserta 1 ej `shouldBe` N 3 (N 2 (N 1 V V) V) (N 6 (N 4 V V) (N 8 V V))
  it "e3" $
    inserta 2 ej `shouldBe` N 3 (N 2 V V) (N 6 (N 4 V V) (N 8 V V))
```

```haskell
-- La verificación es
--    λ> verifica
--    3 examples, 0 failures

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

-- (elementos a) es la lista de los valores de los nodos del ABB a en el
-- recorrido inorden. Por ejemplo,
--    elementos ej == [2,3,4,6,8]
elementos :: ABB a -> [a]
elementos V         = []
elementos (N v i d) = elementos i ++ [v] ++ elementos d

-- (menorTodos v a) se verifica si v es menor que todos los elementos
-- del ABB a. Por ejemplo,
--    menorTodos 1 ej == True
--    menorTodos 2 ej == False
menorTodos :: Ord a => a -> ABB a -> Bool
menorTodos _ V = True
menorTodos v a = v < minimum (elementos a)

-- (mayorTodos v a) se verifica si v es mayor que todos los elementos
-- del ABB a. Por ejemplo,
--    mayorTodos 9 ej == True
--    mayorTodos 8 ej == False
mayorTodos :: Ord a => a -> ABB a -> Bool
mayorTodos _ V = True
mayorTodos v a  = v > maximum (elementos a)

-- (esABB a) se verifica si a es un ABB correcto. Por ejemplo,
--    esABB ej == True
esABB :: Ord a => ABB a -> Bool
esABB V         = True
esABB (N v i d) = mayorTodos v i &&
                  menorTodos v d &&
                  esABB i &&
                  esABB d

-- vacio es el árbol binario de búsqueda vacío.
vacio :: ABB a
vacio = V

-- genABB es un generador de árboles binarios de búsqueda. Por ejemplo,
--    λ> generate genABB
```

```
--    N (-23) (N (-29) V V) (N (-3) V (N 6 V V))
--    λ> generate genABB
--    N (-24) V V
genABB :: Gen (ABB Int)
genABB = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

instance Arbitrary (ABB Int) where
  arbitrary = genABB

-- La propiedad es
prop_inserta :: Int -> ABB Int -> Bool
prop_inserta v a =
  esABB (inserta v a)

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

Ejercicio 64

Matriz permutación

```haskell
-- ---------------------------------------------------------------------
-- Una matriz permutación es una matriz cuadrada con todos sus elementos
-- iguales a 0, excepto uno cualquiera por cada fila y columna, el cual
-- debe ser igual a 1.
--
-- En este ejercicio se usará el tipo de las matrices definido por
--    type Matriz a = Array (Int,Int) a
-- y los siguientes ejemplos de matrices
--    q1, q2, q3,q4 :: Matriz Int
--    q1 = array ((1,1),(2,2)) [((1,1),1),((1,2),0),((2,1),0),((2,2),1)]
--    q2 = array ((1,1),(2,2)) [((1,1),0),((1,2),1),((2,1),0),((2,2),1)]
--    q3 = array ((1,1),(2,2)) [((1,1),3),((1,2),0),((2,1),0),((2,2),1)]
--    q4 = array ((1,1),(2,2)) [((1,1),1),((1,2),3),((2,1),0),((2,2),1)]
--
-- Definir la función
--    esMatrizPermutacion :: Num a => Matriz a -> Bool
-- tal que (esMatrizPermutacion p) se verifica si p es una matriz
-- permutación. Por ejemplo.
--    esMatrizPermutacion q1  ==  True
--    esMatrizPermutacion q2  ==  False
--    esMatrizPermutacion q3  ==  False
-- ---------------------------------------------------------------------

module Matriz_permutacion where

import Data.Array (Array, (!), array, bounds, listArray)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

type Matriz a = Array (Int,Int) a

q1, q2, q3, q4 :: Matriz Int
```

```haskell
q1 = array ((1,1),(2,2)) [((1,1),1),((1,2),0),((2,1),0),((2,2),1)]
q2 = array ((1,1),(2,2)) [((1,1),0),((1,2),1),((2,1),0),((2,2),1)]
q3 = array ((1,1),(2,2)) [((1,1),3),((1,2),0),((2,1),0),((2,2),1)]
q4 = array ((1,1),(2,2)) [((1,1),1),((1,2),3),((2,1),0),((2,2),1)]

-- 1ª solución
-- ===========

esMatrizPermutacion1 :: (Num a, Eq a) => Matriz a -> Bool
esMatrizPermutacion1 p =
  all esListaUnitaria (filas p) &&
  all esListaUnitaria (columnas p)

-- (filas p) es la lista de las filas de la matriz p. Por ejemplo,
--    filas q1 == [[1,0],[0,1]]
--    filas q2 == [[0,1],[0,1]]
--    filas q3 == [[3,0],[0,1]]
--    filas q4 == [[1,3],[0,1]]
filas :: (Num a, Eq a) => Matriz a -> [[a]]
filas p =
  [[p!(i,j) | j <- [1..n]] | i <- [1..n]]
  where (_,(n,_)) = bounds p

-- (columnas p) es la lista de las columnas de la matriz p. Por ejemplo,
--    columnas q1 == [[1,0],[0,1]]
--    columnas q2 == [[0,0],[1,1]]
--    columnas q3 == [[3,0],[0,1]]
--    columnas q4 == [[1,0],[3,1]]
columnas :: (Num a, Eq a) => Matriz a -> [[a]]
columnas p =
  [[p!(i,j) | i <- [1..n]] | j <- [1..n]]
  where (_,(n,_)) = bounds p

-- (esListaUnitaria xs) se verifica si xs tiene un 1 y los restantes
-- elementos son 0. Por ejemplo,
--    esListaUnitaria [0,1,0,0]  ==  True
--    esListaUnitaria [0,1,0,1]  ==  False
--    esListaUnitaria [0,2,0,0]  ==  False
esListaUnitaria :: (Num a, Eq a) => [a] -> Bool
esListaUnitaria xs =
  [x | x <- xs, x /= 0] == [1]

-- 2ª solución
-- ===========
```

```haskell
esMatrizPermutacion2 :: (Num a, Eq a) => Matriz a -> Bool
esMatrizPermutacion2 p =
  all esListaUnitaria (filas p ++ columnas p)

-- 3ª solución
-- ===========

esMatrizPermutacion3 :: (Num a, Eq a) => Matriz a -> Bool
esMatrizPermutacion3 p =
    and [esListaUnitaria [p!(i,j) | i <- [1..n]] | j <- [1..n]] &&
    and [esListaUnitaria [p!(i,j) | j <- [1..n]] | i <- [1..n]]
    where (_,(n,_)) = bounds p

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Matriz Int -> Bool) -> Spec
specG esMatrizPermutacion = do
  it "e1" $
    esMatrizPermutacion q1 `shouldBe` True
  it "e2" $
    esMatrizPermutacion q2 `shouldBe` False
  it "e3" $
    esMatrizPermutacion q3 `shouldBe` False

spec :: Spec
spec = do
  describe "def. 1" $ specG esMatrizPermutacion1
  describe "def. 2" $ specG esMatrizPermutacion2
  describe "def. 3" $ specG esMatrizPermutacion3

-- La verificación es
--    λ> verifica
--    9 examples, 0 failures

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

newtype Matriz2 = M (Array (Int,Int) Int)
  deriving Show

-- (matrizArbitraria n) es un  generador de matrices cuadradas
```

```
-- arbitrarias de orden nxn. Por ejemplo,
--    λ> generate (matrizArbitraria 3)
--    M (array ((1,1),(3,3)) [((1,1),-8), ((1,2),3),  ((1,3),-21),
--                            ((2,1),-17),((2,2),-24),((2,3),30),
--                            ((3,1),-29),((3,2),17), ((3,3),-28)])
matrizArbitraria :: Int -> Gen Matriz2
matrizArbitraria n = do
  xs <- vectorOf (n*n) arbitrary
  return (M (listArray ((1,1),(n,n)) xs))

-- (permutacionAfila xs i) es la lista de los elementos de la fila
-- i-ésima de la matriz permutación correspondiente a la permutación xs
-- de los números peimeros números. Por ejemplo,
--    permutacionAfila [3,1,2] 1 == [0,1,0]
--    permutacionAfila [3,1,2] 2 == [0,0,1]
--    permutacionAfila [3,1,2] 3 == [1,0,0]
permutacionAfila :: [Int] -> Int -> [Int]
permutacionAfila xs i =
  map f xs
  where f x | x == i    = 1
            | otherwise = 0

-- (permutacionAmatriz xs) es la matriz permutación correspondiente a la
-- permutación xs de los números peimeros números. Por ejemplo,
--    λ> permutacionAmatriz [3,1,2]
--    array ((1,1),(3,3)) [((1,1),0),((1,2),1),((1,3),0),
--                         ((2,1),0),((2,2),0),((2,3),1),
--                         ((3,1),1),((3,2),0),((3,3),0)]
permutacionAmatriz :: [Int] -> Matriz Int
permutacionAmatriz xs =
  listArray ((1,1),(n,n)) (concat [permutacionAfila xs i | i <- [1..n]])
  where n = length xs

-- (permutacionArbitraria n) es un generador de permutaciones de los
-- números [1..n]. Por ejemplo,
--    λ> generate (permutacionArbitraria 5)
--    [3,5,2,4,1]
permutacionArbitraria :: Int -> Gen [Int]
permutacionArbitraria n =
  shuffle [1..n]

-- (matrizPermutacionArbitraria n) es un  generador de matrices
-- permutació arbitrarias de orden nxn. Por ejemplo,
--    λ> generate (matrizPermutacionArbitraria 3)
--    M (array ((1,1),(3,3)) [((1,1),1),((1,2),0),((1,3),0),
```

```
--                            ((2,1),0),((2,2),0),((2,3),1),
--                            ((3,1),0),((3,2),1),((3,3),0)])
matrizPermutacionArbitraria :: Int -> Gen Matriz2
matrizPermutacionArbitraria n = do
  xs <- permutacionArbitraria n
  return (M (permutacionAmatriz xs))

-- Matriz es una subclase de Arbitrary.
instance Arbitrary Matriz2 where
  arbitrary = sized $ \n -> do
    frequency
      [ (3, matrizPermutacionArbitraria n)  -- 75% matrices permutación
      , (1, matrizArbitraria n)             -- 25% matrices aleatorias
      ]

-- La propiedad es
prop_esMatrizPermutacion :: Matriz2 -> Bool
prop_esMatrizPermutacion (M p) =
  all (== esMatrizPermutacion1 p)
      [esMatrizPermutacion2 p,
       esMatrizPermutacion2 p]

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

-- La propiedad para que indique el porcentaje de matrices permutación
-- generadas.
prop_esMatrizPermutacion2 :: Matriz2 -> Property
prop_esMatrizPermutacion2 (M p) =
  collect r $ esMatrizPermutacion2 p == r && esMatrizPermutacion3 p == r
  where r = esMatrizPermutacion1 p

-- La comprobación es
--    λ> quickCheck prop_esMatrizPermutacion2
--    +++ OK, passed 100 tests:
--    76% True
--    24% False

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

-- La comparación es
--    λ> (M ej) <- generate (matrizPermutacionArbitraria 1000)
--    λ> esMatrizPermutacion1 ej
```

```
--    True
--    (1.72 secs, 1,251,562,208 bytes)
--    λ> esMatrizPermutacion2 ej
--    True
--    (1.19 secs, 978,158,784 bytes)
--    λ> esMatrizPermutacion3 ej
--    True
--    (1.16 secs, 978,454,728 bytes)
```

Ejercicio 65

Números con todos sus dígitos primos

```haskell
-- ---------------------------------------------------------------------
-- Definir la lista
--    numerosConDigitosPrimos :: [Integer]
-- cuyos elementos son los números con todos sus dígitos primos. Por
-- ejemplo,
--    λ> take 22 numerosConDigitosPrimos
--    [2,3,5,7,22,23,25,27,32,33,35,37,52,53,55,57,72,73,75,77,222,223]
--    λ> numerosConDigitosPrimos !! (10^7)
--    322732232572
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Numeros_con_digitos_primos where

import Data.Char (intToDigit)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck (NonNegative (NonNegative), quickCheck)

-- 1ª solución
-- ===========

numerosConDigitosPrimos1 :: [Integer]
numerosConDigitosPrimos1 = [n | n <- [2..], digitosPrimos n]

-- (digitosPrimos n) se verifica si todos los dígitos de n son
-- primos. Por ejemplo,
--    digitosPrimos 352  ==  True
--    digitosPrimos 362  ==  False
```

```haskell
digitosPrimos :: Integer -> Bool
digitosPrimos n = subconjunto (digitos n) [2,3,5,7]

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

-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys. Por
-- ejemplo,
--    subconjunto [3,2,5,2] [2,7,3,5]  ==  True
--    subconjunto [3,2,5,2] [2,7,2,5]  ==  False
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [x `elem` ys | x <- xs]

-- 2ª solución
-- ===========

numerosConDigitosPrimos2 :: [Integer]
numerosConDigitosPrimos2 =
  filter (all (`elem` "2357") . show) [2..]

-- 3ª solución
-- ===========

--    λ> take 60 numerosConDigitosPrimos2
--    [  2,  3,  5,  7,
--      22, 23, 25, 27,
--      32, 33, 35, 37,
--      52, 53, 55, 57,
--      72, 73, 75, 77,
--     222,223,225,227,
--     232,233,235,237,
--     252,253,255,257,
--     272,273,275,277,
--     322,323,325,327,
--     332,333,335,337,
--     352,353,355,357,
--     372,373,375,377,
--     522,523,525,527,
--     532,533,535,537]

numerosConDigitosPrimos3 :: [Integer]
numerosConDigitosPrimos3 =
  [2,3,5,7] ++ [10*n+d | n <- numerosConDigitosPrimos3, d <- [2,3,5,7]]
```

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

--    λ> take 60 numerosConDigitosPrimos2
--    [ 2, 3, 5, 7,
--     22,23,25,27,
--     32,33,35,37,
--     52,53,55,57,
--     72,73,75,77,
--     222,223,225,227, 232,233,235,237, 252,253,255,257, 272,273,275,277,
--     322,323,325,327, 332,333,335,337, 352,353,355,357, 372,373,375,377,
--     522,523,525,527, 532,533,535,537]

numerosConDigitosPrimos4 :: [Integer]
numerosConDigitosPrimos4 = concat (iterate siguiente [2,3,5,7])

-- (siguiente xs) es la lista obtenida añadiendo delante de cada
-- elemento de xs los dígitos 2, 3, 5 y 7. Por ejemplo,
--    λ> siguiente [5,6,8]
--    [25,26,28,
--     35,36,38,
--     55,56,58,
--     75,76,78]
siguiente :: [Integer] -> [Integer]
siguiente xs = concat [map (pega d) xs | d <- [2,3,5,7]]

-- (pega d n) es el número obtenido añadiendo el dígito d delante del
-- número n. Por ejemplo,
--    pega 3 35  ==  335
pega :: Int -> Integer -> Integer
pega d n = read (intToDigit d : show n)

-- 5ª solución
-- ===========

numerosConDigitosPrimos5 :: [Integer]
numerosConDigitosPrimos5 = concat (iterate siguiente2 [2,3,5,7])
  where
    siguiente2 xs = [10 * n + d | n <- xs, d <- [2,3,5,7]]

-- 6ª solución
-- ===========

numerosConDigitosPrimos6 :: [Integer]
```

```haskell
numerosConDigitosPrimos6 =
  concatMap numerosConDigitosPrimosAux [1..]

-- (numerosConDigitosPrimosAux n) son los números formados con n digitos
-- primos. Por ejemplo,
--    λ> numerosConDigitosPrimosAux 2
--    [22,23,25,27,32,33,35,37,52,53,55,57,72,73,75,77]
numerosConDigitosPrimosAux :: Int -> [Integer]
numerosConDigitosPrimosAux n =
  map read (mapM (const "2357") [1..n])

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: [Integer] -> Spec
specG numerosConDigitosPrimos = do
  it "e1" $
    take 22 numerosConDigitosPrimos `shouldBe`
    [2,3,5,7,22,23,25,27,32,33,35,37,52,53,55,57,72,73,75,77,222,223]

spec :: Spec
spec = do
  describe "def. 1" $ specG numerosConDigitosPrimos1
  describe "def. 2" $ specG numerosConDigitosPrimos2
  describe "def. 3" $ specG numerosConDigitosPrimos3
  describe "def. 4" $ specG numerosConDigitosPrimos4
  describe "def. 5" $ specG numerosConDigitosPrimos5
  describe "def. 6" $ specG numerosConDigitosPrimos6

-- La verificación es
--    λ> verifica
--    6 examples, 0 failures

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

-- La propiedad es
prop_numerosConDigitosPrimos_equiv :: NonNegative Int -> Bool
prop_numerosConDigitosPrimos_equiv (NonNegative n) =
  all (== numerosConDigitosPrimos1 !! n)
      [ numerosConDigitosPrimos2 !! n
      , numerosConDigitosPrimos3 !! n
```

```
      , numerosConDigitosPrimos4 !! n
      , numerosConDigitosPrimos5 !! n
      , numerosConDigitosPrimos6 !! n
      ]

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

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

-- La comparación es
--    λ> numerosConDigitosPrimos1 !! 5000
--    752732
--    (2.45 secs, 6,066,926,272 bytes)
--    λ> numerosConDigitosPrimos2 !! 5000
--    752732
--    (0.34 secs, 387,603,456 bytes)
--    λ> numerosConDigitosPrimos3 !! 5000
--    752732
--    (0.01 secs, 1,437,624 bytes)
--    λ> numerosConDigitosPrimos4 !! 5000
--    752732
--    (0.00 secs, 1,556,104 bytes)
--    λ> numerosConDigitosPrimos5 !! 5000
--    752732
--    (0.02 secs, 1,964,952 bytes)
--    λ> numerosConDigitosPrimos6 !! 5000
--    752732
--    (0.01 secs, 1,981,704 bytes)
--
--    λ> numerosConDigitosPrimos3 !! (10^7)
--    322732232572
--    (2.53 secs, 1,860,588,784 bytes)
--    λ> numerosConDigitosPrimos4 !! (10^7)
--    322732232572
--    (3.61 secs, 2,000,679,432 bytes)
--    λ> numerosConDigitosPrimos5 !! (10^7)
--    322732232572
--    (3.51 secs, 2,780,609,160 bytes)
--    λ> numerosConDigitosPrimos6 !! (10^7)
--    322732232572
--    (2.19 secs, 3,116,479,176 bytes)
```

Ejercicio 66

Cadenas de ceros y unos

```
-- ------------------------------------------------------------------
-- Definir la constante
--    cadenasCerosUnos :: [String]
-- tal que cadenasCerosUnos es la lista de cadenas de ceros y unos,
-- ordenada lexicográficamente. Por ejemplo,
--    λ> take 15 cadenasCerosUnos1
--    ["","0","1","00","01","10","11","000","001","010","011","100",
--     "101","110","111"]
-- ------------------------------------------------------------------

module Cadenas0y1 where

import Control.Monad (replicateM)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
-- ===========

cadenasCerosUnos1 :: [String]
cadenasCerosUnos1 =
  concatMap cadenasLongitud1 [0..]

-- (cadenasLongitud1 n) es la lista de cadenas de loongitud n de ceros y
-- unos, ordenada lexicográficamente. Por ejemplo,
--    λ> cadenasLongitud1 3
--    ["000","001","010","011","100","101","110","111"]
cadenasLongitud1 :: Int -> [String]
cadenasLongitud1 0 = [""]
cadenasLongitud1 n = [ c : s | c <- ['0', '1'], s <- cadenasLongitud1 (n - 1) ]

-- 2ª solución
```

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

cadenasCerosUnos2 :: [String]
cadenasCerosUnos2 =
  concatMap cadenasLongitud2 [0..]

cadenasLongitud2 :: Int -> [String]
cadenasLongitud2 n =
  sequence (replicate n ['0', '1'])

-- 3ª solución
-- ===========

cadenasCerosUnos3 :: [String]
cadenasCerosUnos3 =
  concatMap cadenasLongitud3 [0..]

cadenasLongitud3 :: Int -> [String]
cadenasLongitud3 n =
  replicateM n ['0', '1']

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: [String] -> Spec
specG cadenasCerosUnos = do
  it "e1" $
    take 15 cadenasCerosUnos `shouldBe`
    ["","0","1","00","01","10","11","000","001","010","011","100","101","110","111"]

spec :: Spec
spec = do
  describe "def. 1" $ specG cadenasCerosUnos1
  describe "def. 2" $ specG cadenasCerosUnos2
  describe "def. 3" $ specG cadenasCerosUnos3

-- La verificación es
--    λ> verifica
--    2 examples, 0 failures

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

```haskell
-- La propiedad es
prop_cadenasCerosUnos :: NonNegative Int -> Bool
prop_cadenasCerosUnos (NonNegative n) =
  all (== cadenasCerosUnos1 !! n)
      [ cadenasCerosUnos2 !! n
      , cadenasCerosUnos3 !! n
      ]

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

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

-- La comparación es
--    λ> cadenasCerosUnos1 !! 1000000
--    "11101000001001000001"
--    (5.85 secs, 2,655,311,384 bytes)
--    λ> cadenasCerosUnos2 !! 1000000
--    "11101000001001000001"
--    (0.10 secs, 253,292,456 bytes)
--    λ> cadenasCerosUnos3 !! 1000000
--    "11101000001001000001"
--    (0.09 secs, 253,282,368 bytes)
```

Ejercicio 67

Clausura de un conjunto respecto de una función

```haskell
-- ----------------------------------------------------------------------
-- Un conjunto A está cerrado respecto de una función  f si para todo
-- elemento x de A se tiene que f(x) pertenece a A. La clausura de un
-- conjunto B respecto de una función f es el menor conjunto A que
-- contiene a B y es cerrado respecto de f. Por ejemplo, la clausura de
-- {0,1,2} respecto del opuesto es {-1,-2,0,1,2}.
--
-- Definir la función
--    clausura :: Ord a => (a -> a) -> [a] -> [a]
-- tal que (clausura f xs) es la clausura ordenada de xs respecto de
-- f. Por ejemplo,
--    clausura (\x -> -x) [0,1,2]         ==  [-2,-1,0,1,2]
--    clausura (\x -> (x+1) `mod` 5) [0]  ==  [0,1,2,3,4]
--    length (clausura (\x -> (x+1) `mod` (10^6)) [0]) == 1000000
-- ----------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Clausura where

import Data.List ((\\), nub, sort, union)
import qualified Data.Set as S (Set, difference, fromList, map, null, toList, union)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck.HigherOrder (quickCheck')

-- 1ª solución
-- ===========

clausura1 :: Ord a => (a -> a) -> [a] -> [a]
```

```haskell
clausura1 f xs
  | esCerrado f xs = sort xs
  | otherwise      = clausura1 f (expansion f xs)

-- (esCerrado f xs) se verifica si al aplicar f a cualquier elemento de
-- xs se obtiene un elemento de xs. Por ejemplo,
--    λ> esCerrado (\x -> -x) [0,1,2]
--    False
--    λ> esCerrado (\x -> -x) [0,1,2,-2,-1]
--    True
esCerrado :: Ord a => (a -> a) -> [a] -> Bool
esCerrado f xs = all (`elem` xs) (map f xs)

-- (expansion f xs) es la lista (sin repeticiones) obtenidas añadiéndole
-- a xs el resultado de aplicar f a sus elementos. Por ejemplo,
--    expansion (\x -> -x) [0,1,2]  ==  [0,1,2,-1,-2]
expansion :: Ord a => (a -> a) -> [a] -> [a]
expansion f xs = xs `union` map f xs

-- 2ª solución
-- ===========

clausura2 :: Ord a => (a -> a) -> [a] -> [a]
clausura2 f xs = sort (until (esCerrado f) (expansion f) xs)

-- 3ª solución
-- ===========

clausura3 :: Ord a => (a -> a) -> [a] -> [a]
clausura3 f xs = aux xs xs
  where aux ys vs | null ns   = sort vs
                  | otherwise = aux ns (vs ++ ns)
          where ns = nub (map f ys) \\ vs

-- 4ª solución
-- ===========

clausura4 :: Ord a => (a -> a) -> [a] -> [a]
clausura4 f xs = S.toList (clausura4' f (S.fromList xs))

clausura4' :: Ord a => (a -> a) -> S.Set a -> S.Set a
clausura4' f xs = aux xs xs
  where aux ys vs | S.null ns = vs
                  | otherwise = aux ns (vs `S.union` ns)
          where ns = S.map f ys `S.difference` vs
```

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

verifica :: IO ()
verifica = hspec spec

specG :: ((Int -> Int) -> [Int] -> [Int]) -> Spec
specG clausura = do
  it "e1" $
    clausura (\x -> -x) [0,1,2]         `shouldBe`  [-2,-1,0,1,2]
  it "e2" $
    clausura (\x -> (x+1) `mod` 5) [0]  `shouldBe`  [0,1,2,3,4]

spec :: Spec
spec = do
  describe "def. 1" $ specG clausura1
  describe "def. 2" $ specG clausura2
  describe "def. 3" $ specG clausura3
  describe "def. 4" $ specG clausura4

-- La verificación es
--    λ> verifica
--    8 examples, 0 failures

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

-- La propiedad es
prop_clausura :: (Int -> Int) -> [Int] -> Bool
prop_clausura f xs =
  all (== clausura1 f xs')
      [ clausura2 f xs'
      , clausura3 f xs'
      , clausura4 f xs'
      ]
  where xs' = sort (nub xs)

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

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

```
-- La comparación es
--    λ> length (clausura1 (\x -> (x+1) `mod` 800) [0])
--    800
--    (1.95 secs, 213,481,560 bytes)
--    λ> length (clausura2 (\x -> (x+1) `mod` 800) [0])
--    800
--    (1.96 secs, 213,372,824 bytes)
--    λ> length (clausura3 (\x -> (x+1) `mod` 800) [0])
--    800
--    (0.03 secs, 42,055,128 bytes)
--    λ> length (clausura4 (\x -> (x+1) `mod` 800) [0])
--    800
--    (0.01 secs, 1,779,768 bytes)
--
--    λ> length (clausura3 (\x -> (x+1) `mod` (10^4)) [0])
--    10000
--    (2.50 secs, 8,080,105,816 bytes)
--    λ> length (clausura4 (\x -> (x+1) `mod` (10^4)) [0])
--    10000
--    (0.05 secs, 27,186,920 bytes)
```

Ejercicio 68

Sustitución en una expresión aritmética

```haskell
-- ----------------------------------------------------------------------
-- La expresiones aritméticas se pueden representar mediante el
-- siguiente tipo
--    data Expr = C Int
--              | V Char
--              | S Expr Expr
--              | P Expr Expr
--      deriving (Eq, Show)
-- por ejemplo, la expresión "z*(3+x)" se representa por
-- (P (V 'z') (S (C 3) (V 'x'))).
--
-- Definir la función
--    sustitucion :: Expr -> [(Char, Int)] -> Expr
-- tal que (sustitucion e s) es la expresión obtenida sustituyendo las
-- variables de la expresión e según se indica en la sustitución s. Por
-- ejemplo,
--    λ> sustitucion (P (V 'z') (S (C 3) (V 'x'))) [('x',7),('z',9)]
--    P (C 9) (S (C 3) (C 7))
--    λ> sustitucion (P (V 'z') (S (C 3) (V 'y'))) [('x',7),('z',9)]
--    P (C 9) (S (C 3) (V 'y'))
-- ----------------------------------------------------------------------

module Sustitucion_en_una_expresion_aritmetica where

import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

data Expr = C Int
          | V Char
```

```
          | S Expr Expr
          | P Expr Expr
  deriving (Eq, Show)

-- 1ª solución
-- ===========

sustitucion1 :: Expr -> [(Char, Int)] -> Expr
sustitucion1 e []         = e
sustitucion1 (V c) ((d,n):ps)
  | c == d                = C n
  | otherwise             = sustitucion1 (V c) ps
sustitucion1 (C n) _      = C n
sustitucion1 (S e1 e2) ps = S (sustitucion1 e1 ps) (sustitucion1 e2 ps)
sustitucion1 (P e1 e2) ps = P (sustitucion1 e1 ps) (sustitucion1 e2 ps)

-- 2ª solución
-- ===========

sustitucion2 :: Expr -> [(Char, Int)] -> Expr
sustitucion2 (V c) s = case lookup c s of
                       Just n  -> C n
                       Nothing -> V c
sustitucion2 (C n) _ = C n
sustitucion2 (S e1 e2) s = S (sustitucion2 e1 s) (sustitucion2 e2 s)
sustitucion2 (P e1 e2) s = P (sustitucion2 e1 s) (sustitucion2 e2 s)

-- 3ª solución
-- ===========

sustitucion3 :: Expr -> [(Char, Int)] -> Expr
sustitucion3 (V c) s     = maybe (V c) C (lookup c s)
sustitucion3 (C n) _     = C n
sustitucion3 (S e1 e2) s = S (sustitucion3 e1 s) (sustitucion3 e2 s)
sustitucion3 (P e1 e2) s = P (sustitucion3 e1 s) (sustitucion3 e2 s)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Expr -> [(Char, Int)] -> Expr) -> Spec
specG sustitucion = do
  it "e1" $
```

```
    sustitucion (P (V 'z') (S (C 3) (V 'x'))) [('x',7),('z',9)]
    `shouldBe` P (C 9) (S (C 3) (C 7))
  it "e2" $
    sustitucion (P (V 'z') (S (C 3) (V 'y'))) [('x',7),('z',9)]
    `shouldBe` P (C 9) (S (C 3) (V 'y'))

spec :: Spec
spec = do
  describe "def. 1" $ specG sustitucion1
  describe "def. 2" $ specG sustitucion2
  describe "def. 3" $ specG sustitucion3

-- La verificación es
--    λ> verifica
  --    6 examples, 0 failures


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

-- Generador de caracteres solo entre 'a' y 'z'
genVar :: Gen Char
genVar = elements ['a'..'z']

-- (exprArbitraria n) es un generador de expresiones de tamaño
-- aproximado a n. Por
-- ejemplo,
--    λ> generate (exprArbitraria 4)
--    S (S (P (V 'l') (V 'e')) (S (V 'p') (V 'j'))) (V 'm')
exprArbitraria :: Int -> Gen Expr
exprArbitraria 0 = oneof [
  V <$> genVar,
  C <$> arbitrary
  ]
exprArbitraria n = oneof [
  V <$> genVar,
  C <$> arbitrary,
  S <$> subExpr <*> subExpr,
  P <$> subExpr <*> subExpr
  ]
  where subExpr = exprArbitraria (n `div` 2)

instance Arbitrary Expr where
  arbitrary = sized exprArbitraria
```

```haskell
-- La propiedad es
prop_sustitucion :: Expr -> [(Char, Int)] -> Bool
prop_sustitucion e s =
  all (== sustitucion1 e s)
      [sustitucion2 e s,
       sustitucion3 e s
      ]

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

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

-- (exprArbitrariaProf n) es un generador de expresiones de tamaño
-- aproximado a n. Por ejemplo,
--    λ> generate (exprArbitrariaProf 3)
--    S (S (S (C 5) (C 1)) (S (C 11) (V 's'))) (S (S (V 't') (C (-13))) (S (V 'a') (V 'r'
exprArbitrariaProf :: Int -> Gen Expr
exprArbitrariaProf 0 = oneof [
  V <$> genVar,
  C <$> arbitrary
  ]
exprArbitrariaProf n = oneof [
  S <$> exprArbitrariaProf (n-1) <*> exprArbitrariaProf (n-1),
  P <$> exprArbitrariaProf (n-1) <*> exprArbitrariaProf (n-1)
  ]

-- La comparación es
--    λ> ej <- generate (exprArbitrariaProf 20)
--    λ> let r = sustitucion1 ej (zip ['a'..'z'] [1..]) in r == r
--    True
--    (7.19 secs, 2,905,973,720 bytes)
--    λ> let r = sustitucion2 ej (zip ['a'..'z'] [1..]) in r == r
--    True
--    (1.75 secs, 470,342,024 bytes)
--    λ> let r = sustitucion3 ej (zip ['a'..'z'] [1..]) in r == r
--    True
--    (1.73 secs, 482,919,872 bytes)
```

Ejercicio 69

Laberinto numérico

```haskell
-- ----------------------------------------------------------------------
-- El problema del laberinto numérico consiste en, dados un par de
-- números enteros positivos, encontrar la longitud del camino más corto
-- entre ellos usando sólo las siguientes operaciones:
-- + multiplicar por 2,
-- + dividir por 2 (sólo para los pares) y
-- + sumar 2.
--
-- Por ejemplo, un camino mínimo
-- + de  3 a 12 es [3,6,12],
-- + de 12 a  3 es [12,6,3],
-- + de  9 a  2 es [9,18,20,10,12,6,8,4,2] y
-- + de  2 a  9 es [2,4,8,16,18,9].
--
-- Definir la función
--    longitudCaminoMinimo :: Int -> Int -> Int
-- tal que (longitudCaminoMinimo x y) es la longitud del camino mínimo
-- desde x hasta y en el laberinto numérico.
--    longitudCaminoMinimo 3 12  ==  2
--    longitudCaminoMinimo 12 3  ==  2
--    longitudCaminoMinimo 9 2   ==  8
--    longitudCaminoMinimo 2 9   ==  5
-- ----------------------------------------------------------------------

module Laberinto_numerico where

import Data.List (sort, nub)
import qualified Data.Set as Set
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª solución
```

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

longitudCaminoMinimo1 :: Int -> Int -> Int
longitudCaminoMinimo1 x y =
  head [n | n <- [0..], y `elem` orbita n [x]]

-- (orbita n xs) es el conjunto de números que se pueden obtener aplicando
-- como máximo n veces las operaciones a los elementos de xs. Por ejemplo,
--    orbita 0 [12]  ==  [12]
--    orbita 1 [12]  ==  [6,12,14,24]
--    orbita 2 [12]  ==  [3,6,7,8,12,14,16,24,26,28,48]
orbita :: Int -> [Int] -> [Int]
orbita 0 xs = sort xs
orbita n xs = sort (nub (ys ++ concat [sucesores x | x <- ys]))
  where ys = orbita (n-1) xs

-- (sucesores x) es la lista de los sucesores de x; es decir, los
-- números obtenidos aplicándole la operaciones a x. Por ejemplo,
--    sucesores 3 == [6,5]
--    sucesores 4 == [8,6,2]
sucesores :: Int -> [Int]
sucesores x = [2*x, x+2] ++ [x `div` 2 | even x]

-- 2ª solución
-- ===========

longitudCaminoMinimo2 :: Int -> Int -> Int
longitudCaminoMinimo2 x y
  | x == y    = 0
  | otherwise = anchura [(x, 0)] (Set.singleton x)
  where
    anchura [] _ = -1
    anchura ((nodo, dist):cola) visitados
      | nodo == y = dist
      | otherwise = anchura (cola ++ [(n, dist + 1) | n <- nuevos])
                            (foldr Set.insert visitados nuevos)
      where
        nuevos = filter (`Set.notMember` visitados) (sucesores nodo)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec
```

```
specG :: (Int -> Int -> Int) -> Spec
specG longitudCaminoMinimo = do
  it "e1" $
    longitudCaminoMinimo 3 12 `shouldBe` 2
  it "e2" $
    longitudCaminoMinimo 12 3 `shouldBe` 2
  it "e3" $
    longitudCaminoMinimo 9 2  `shouldBe` 8
  it "e4" $
    longitudCaminoMinimo 2 9  `shouldBe` 5

spec :: Spec
spec = do
  describe "def. 1" $ specG longitudCaminoMinimo1
  describe "def. 2" $ specG longitudCaminoMinimo2

-- La verificación es
--    λ> verifica
--    8 examples, 0 failures

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

-- La propiedad es
prop_longitudCaminoMinimo :: Positive Int -> Positive Int -> Bool
prop_longitudCaminoMinimo (Positive x) (Positive y) =
  longitudCaminoMinimo1 x y == longitudCaminoMinimo2 x y

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

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

-- La comparación es
--    λ> longitudCaminoMinimo1 1 511
--    17
--    (2.29 secs, 58,152,384 bytes)
--    λ> longitudCaminoMinimo2 1 511
--    17
--    (0.20 secs, 683,548,744 bytes)
```