

# Ejercicios de “Informática de 1º de Matemáticas” (2012–13)

José A. Alonso Jiménez

---

Grupo de Lógica Computacional  
Dpto. de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla  
Sevilla, 1 de julio de 2013

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

**Se permite:**

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

**Bajo las condiciones siguientes:**

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



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



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

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

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

# Índice general

<b>1</b>	<b>Definiciones por composición de funciones aritméticas</b>	<b>7</b>
<b>2</b>	<b>Definiciones por composición de funciones sobre listas y booleanos</b>	<b>11</b>
<b>3</b>	<b>Definiciones con condicionales, guardas o patrones</b>	<b>15</b>
<b>4</b>	<b>Definiciones por comprensión (1)</b>	<b>23</b>
<b>5</b>	<b>Definiciones por comprensión (2)</b>	<b>31</b>
<b>6</b>	<b>Definiciones por comprensión con cadenas: El cifrado César</b>	<b>39</b>
<b>7</b>	<b>Definiciones por recursión</b>	<b>45</b>
<b>8</b>	<b>Definiciones por recursión: Ordenación por mezcla</b>	<b>53</b>
<b>9</b>	<b>Definiciones por recursión y por comprensión (1)</b>	<b>57</b>
<b>10</b>	<b>Definiciones por recursión y por comprensión (2)</b>	<b>71</b>
<b>11</b>	<b>Definiciones por recursión y por comprensión (4)</b>	<b>75</b>
<b>12</b>	<b>Funciones sobre cadenas</b>	<b>87</b>
<b>13</b>	<b>Problemas sobre el 2013</b>	<b>95</b>
<b>14</b>	<b>Funciones de orden superior y definiciones por plegados</b>	<b>123</b>
<b>15</b>	<b>Funciones de orden superior y definiciones por plegados (2)</b>	<b>133</b>
<b>16</b>	<b>Evaluación perezosa y listas infinitas (1)</b>	<b>147</b>
<b>17</b>	<b>Evaluación perezosa y listas infinitas (2)</b>	<b>157</b>

<b>18 Tipos de datos algebraicos</b>	<b>173</b>
<b>19 Tipos de datos algebraicos: árboles binarios</b>	<b>179</b>
<b>20 Tipos de datos: Modelización de juego de cartas</b>	<b>187</b>
<b>21 Resolución de problemas matemáticos</b>	<b>197</b>
<b>22 Combinatoria</b>	<b>207</b>
<b>23 Cálculo numérico</b>	<b>225</b>
<b>24 Operaciones con el TAD de polinomios</b>	<b>235</b>
<b>25 División y factorización de polinomios mediante la regla de Ruffini</b>	<b>245</b>
<b>26 Vectores y matrices</b>	<b>253</b>
<b>27 Aplicaciones de la programación funcional con listas infinitas</b>	<b>273</b>
<b>28 Ejercicios complementarios</b>	<b>279</b>
<b>29 Implementación del TAD de los grafos mediante listas</b>	<b>295</b>
<b>30 Problemas básicos con el TAD de los grafos</b>	<b>301</b>
<b>31 Demostración de propiedades por inducción</b>	<b>315</b>
<b>32 Relaciones binarias homogéneas</b>	<b>349</b>

# Introducción

Este libro es una recopilación de las soluciones de ejercicios de la asignatura de “Informática” (de 1º del Grado en Matemáticas) correspondientes al curso 2012–13.

El objetivo de los ejercicios es complementar la introducción a la programación funcional y a la algorítmica con Haskell presentada en los temas del curso. Los apuntes de los temas se encuentran en [Temas de “Programación funcional”](#) <sup>1</sup>.

Los ejercicios sigue el orden de las relaciones de problemas propuestos durante el curso y, resueltos de manera colaborativa, en la wiki.

---

<sup>1</sup><http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/2012-13-IM-temas-PF.pdf>



# Relación 1

## Definiciones por composición de funciones aritméticas

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se plantean ejercicios con definiciones  
-- de funciones por composición de las primitivas de Haskell sobre  
-- aritmética.  
--  
-- Para solucionar los ejercicios puede ser útil el "Resumen de  
-- funciones de Haskell" que se encuentra en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/doc/resumen\_Haskell.pdf  
  
-- -----  
-- Ejercicio 1. Definir la función media3 tal que (media3 x y z) es  
-- la media aritmética de los números x, y y z. Por ejemplo,  
-- media3 1 3 8 == 4.0  
-- media3 (-1) 0 7 == 2.0  
-- media3 (-3) 0 3 == 0.0  
-- -----  
  
media3 x y z = (x+y+z)/3  
  
-- -----  
-- Ejercicio 2. Definir la función sumaMonedas tal que  
-- (sumaMonedas a b c d e) es la suma de los euros correspondientes a
```

```
-- a monedas de 1 euro, b de 2 euros, c de 5 euros, d 10 euros y
-- e de 20 euros. Por ejemplo,
-- sumaMonedas 0 0 0 0 1 == 20
-- sumaMonedas 0 0 8 0 3 == 100
-- sumaMonedas 1 1 1 1 1 == 38
```

```
-----
sumaMonedas a b c d e = 1*a+2*b+5*c+10*d+20*e
```

```
-----
-- Ejercicio 3. Definir la función volumenEsfera tal que
-- (volumenEsfera r) es el volumen de la esfera de radio r. Por ejemplo,
-- volumenEsfera 10 == 4188.790204786391
-- Indicación: Usar la constante pi.
```

```
-----
volumenEsfera r = (4/3)*pi*r^3
```

```
-----
-- Ejercicio 4. Definir la función areaDeCoronaCircular tal que
-- (areaDeCoronaCircular r1 r2) es el área de una corona circular de
-- radio interior r1 y radio exterior r2. Por ejemplo,
-- areaDeCoronaCircular 1 2 == 9.42477796076938
-- areaDeCoronaCircular 2 5 == 65.97344572538566
-- areaDeCoronaCircular 3 5 == 50.26548245743669
```

```
-----
areaDeCoronaCircular r1 r2 = pi*(r2^2 - r1^2)
```

```
-----
-- Ejercicio 5. Definir la función ultimaCifra tal que (ultimaCifra x)
-- es la última cifra del número x. Por ejemplo,
-- ultimaCifra 325 == 5
-- Indicación: Usar la función rem
```

```
-----
ultimaCifra x = rem x 10
```

```
-----
-- Ejercicio 6. Definir la función maxTres tal que (maxTres x y z) es
```



```
-- el máximo de x, y y z. Por ejemplo,  
--   maxTres 6 2 4 == 6  
--   maxTres 6 7 4 == 7  
--   maxTres 6 7 9 == 9  
-- Indicación: Usar la función max.  
-----
```

```
maxTres x y z = max x (max y z)
```



## Relación 2

# Definiciones por composición de funciones sobre listas y booleanos

```
-----  
-- Introducción --  
-----  
  
-- En esta relación se plantean ejercicios con definiciones  
-- de funciones por composición de las primitivas de Haskell sobre  
-- listas y booleanos.  
--  
-- Para solucionar los ejercicios puede ser útil el "Resumen de  
-- funciones de Haskell" que se encuentra en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/doc/resumen\_Haskell.pdf  
-----  
-- Ejercicio 1. Definir la función rotal tal que (rotal xs) es la lista  
-- obtenida poniendo el primer elemento de xs al final de la lista. Por  
-- ejemplo,  
-- rotal [3,2,5,7] == [2,5,7,3]  
-----  
  
rotal xs = tail xs ++ [head xs]  
-----  
-- Ejercicio 2. Definir la función rota tal que (rota n xs) es la lista
```

```
-- obtenida poniendo los n primeros elementos de xs al final de la
-- lista. Por ejemplo,
--   rota 1 [3,2,5,7] == [2,5,7,3]
--   rota 2 [3,2,5,7] == [5,7,3,2]
--   rota 3 [3,2,5,7] == [7,3,2,5]
```

```
rota n xs = drop n xs ++ take n xs
```

```
-- -----
-- Ejercicio 3. Definir la función rango tal que (rango xs) es la
-- lista formada por el menor y mayor elemento de xs.
--   rango [3,2,7,5] == [2,7]
-- Indicación: Se pueden usar minimum y maximum.
```

```
rango xs = [minimum xs, maximum xs]
```

```
-- -----
-- Ejercicio 4. Definir la función palindromo tal que (palindromo xs) se
-- verifica si xs es un palíndromo; es decir, es lo mismo leer xs de
-- izquierda a derecha que de derecha a izquierda. Por ejemplo,
--   palindromo [3,2,5,2,3] == True
--   palindromo [3,2,5,6,2,3] == False
```

```
palindromo xs = xs == reverse xs
```

```
-- -----
-- Ejercicio 5. Definir la función interior tal que (interior xs) es la
-- lista obtenida eliminando los extremos de la lista |. Por ejemplo,
--   interior [2,5,3,7,3] == [5,3,7]
--   interior [2..7]      == [3,4,5,6]
```

```
interior xs = tail (init xs)
```

```
-- -----
-- Ejercicio 6. Definir la función finales tal que (finales n xs) es la
-- lista formada por los n finales elementos de xs. Por ejemplo,
```

```
--      finales 3 [2,5,4,7,9,6] == [7,9,6]
```

```
finales n xs = drop (length xs - n) xs
```

```
-- -----  
-- Ejercicio 7. Definir la función segmento tal que (segmento m n xs) es  
-- la lista de los elementos de xs comprendidos entre las posiciones m y  
-- n. Por ejemplo,
```

```
--      segmento 3 4 [3,4,1,2,7,9,0] == [1,2]  
--      segmento 3 5 [3,4,1,2,7,9,0] == [1,2,7]  
--      segmento 5 3 [3,4,1,2,7,9,0] == []
```

```
segmento m n xs = drop (m-1) (take n xs)
```

```
-- -----  
-- Ejercicio 8. Definir la función extremos tal que (extremos n xs) es  
-- la lista formada por los n primeros elementos de xs y los n finales  
-- elementos de xs. Por ejemplo,
```

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

```
extremos n xs = take n xs ++ drop (length xs - n) xs
```

```
-- -----  
-- Ejercicio 9. Definir la función mediano tal que (mediano x y z) es el  
-- número mediano de los tres números x, y y z. Por ejemplo,
```

```
--      mediano 3 2 5 == 3  
--      mediano 2 4 5 == 4  
--      mediano 2 6 5 == 5  
--      mediano 2 6 6 == 6
```

```
-- Indicación: Usar maximum y minimum.
```

```
mediano x y z = x + y + z - minimum [x,y,z] - maximum [x,y,z]
```

```
-- -----  
-- Ejercicio 10. Definir la función tresIguales tal que  
-- (tresIguales x y z) se verifica si los elementos x, y y z son
```

```
-- iguales. Por ejemplo,  
-- tresIguales 4 4 4 == True  
-- tresIguales 4 3 4 == False
```

---

```
tresIguales x y z = x == y && y == z
```

---

```
-- Ejercicio 11. Definir la función tresDiferentes tal que  
-- (tresDiferentes x y z) se verifica si los elementos x, y y z son  
-- distintos. Por ejemplo,  
-- tresDiferentes 3 5 2 == True  
-- tresDiferentes 3 5 3 == False
```

---

```
tresDiferentes x y z = x /= y && x /= z && y /= z
```

---

```
-- Ejercicio 12. Definir la función cuatroIguales tal que  
-- (cuatroIguales x y z u) se verifica si los elementos x, y, z y u son  
-- iguales. Por ejemplo,  
-- cuatroIguales 5 5 5 5 == True  
-- cuatroIguales 5 5 4 5 == False  
-- Indicación: Usar la función tresIguales.
```

---

```
cuatroIguales x y z u = x == y && tresIguales y z u
```

# Relación 3

## Definiciones con condicionales, guardas o patrones

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se presentan ejercicios con definiciones elementales  
-- (no recursivas) de funciones que usan condicionales, guardas o  
-- patrones.  
--  
-- Estos ejercicios se corresponden con el tema 4 cuyas transparencias  
-- se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-4t.pdf  
  
-- -----  
-- Ejercicio 1. Definir la función divisionSegura tal que  
-- (divisionSegura x y) es x/y si y no es cero e y 9999 en caso  
-- contrario. Por ejemplo,  
-- divisionSegura 7 2 == 3.5  
-- divisionSegura 7 0 == 9999.0  
-- -----  
  
divisionSegura _ 0 = 9999  
divisionSegura x y = x/y  
  
-- -----  
-- Ejercicio 2. La disyunción excluyente xor de dos fórmulas se verifica
```

```
-- si una es verdadera y la otra es falsa.
-----
-- Ejercicio 2.1. Definir la función xor_1 que calcule la disyunción
-- excluyente a partir de la tabla de verdad. Usar 4 ecuaciones, una por
-- cada línea de la tabla.
```

```
xor_1 :: Bool -> Bool -> Bool
xor_1 True  True  = False
xor_1 True  False = True
xor_1 False True  = True
xor_1 False False = False
```

```
-----
-- Ejercicio 2.2. Definir la función xor_2 que calcule la disyunción
-- excluyente a partir de la tabla de verdad y patrones. Usar 2
-- ecuaciones, una por cada valor del primer argumento.
```

```
xor_2 :: Bool -> Bool -> Bool
xor_2 True  y = not y
xor_2 False y = y
```

```
-----
-- Ejercicio 2.3. Definir la función xor_3 que calcule la disyunción
-- excluyente a partir de la disyunción (||), conjunción (&&) y negación
-- (not). Usar 1 ecuación.
```

```
xor_3 :: Bool -> Bool -> Bool
xor_3 x y = (x || y) && not (x && y)
```

```
-----
-- Ejercicio 2.4. Definir la función xor_4 que calcule la disyunción
-- excluyente a partir de desigualdad (/=). Usar 1 ecuación.
```

```
xor_4 :: Bool -> Bool -> Bool
xor_4 x y = x /= y
```



```
-----  
-- Ejercicio 3. Definir la función modulo tal que (modulo v) es el  
-- módulo del vector v. Por ejemplo,  
--   modulo (3,4) == 5.0  
-----
```

```
modulo (x,y) = sqrt(x^2+y^2)
```

```
-----  
-- Ejercicio 4. Las dimensiones de los rectángulos puede representarse  
-- por pares; por ejemplo, (5,3) representa a un rectángulo de base 5 y  
-- altura 3. Definir la función mayorRectangulo tal que  
-- (mayorRectangulo r1 r2) es el rectángulo de mayor área ente r1 y r2.  
-- Por ejemplo,  
--   mayorRectangulo (4,6) (3,7) == (4,6)  
--   mayorRectangulo (4,6) (3,8) == (4,6)  
--   mayorRectangulo (4,6) (3,9) == (3,9)  
-----
```

```
mayorRectangulo (a,b) (c,d) | a*b >= c*d = (a,b)  
                           | otherwise = (c,d)
```

```
-----  
-- Ejercicio 5. Definir la función cuadrante tal que (cuadrante p) es  
-- es cuadrante del punto p (se supone que p no está sobre los  
-- ejes). Por ejemplo,  
--   cuadrante (3,5) == 1  
--   cuadrante (-3,5) == 2  
--   cuadrante (-3,-5) == 3  
--   cuadrante (3,-5) == 4  
-----
```

```
cuadrante (x,y)  
  | x > 0 && y > 0 = 1  
  | x < 0 && y > 0 = 2  
  | x < 0 && y < 0 = 3  
  | x > 0 && y < 0 = 4
```

```
-----  
-- Ejercicio 6. Definir la función intercambia tal que (intercambia p)
```

```
-- es el punto obtenido intercambiando las coordenadas del punto p. Por
-- ejemplo,
--   intercambia (2,5) == (5,2)
--   intercambia (5,2) == (2,5)
-----
```

```
intercambia (x,y) = (y,x)
```

```
-- -----
-- Ejercicio 7. Definir la función simetricoH tal que (simetricoH p) es
-- el punto simétrico de p respecto del eje horizontal. Por ejemplo,
--   simetricoH (2,5) == (2,-5)
--   simetricoH (2,-5) == (2,5)
-----
```

```
simetricoH (x,y) = (x,-y)
```

```
-- -----
-- Ejercicio 8. Definir la función distancia tal que (distancia p1 p2)
-- es la distancia entre los puntos p1 y p2. Por ejemplo,
--   distancia (1,2) (4,6) == 5.0
-----
```

```
distancia (x1,y1) (x2,y2) = sqrt((x1-x2)^2+(y1-y2)^2)
```

```
-- -----
-- Ejercicio 9. Definir la función puntoMedio tal que (puntoMedio p1 p2)
-- es el punto medio entre los puntos p1 y p2. Por ejemplo,
--   puntoMedio (0,2) (0,6) == (0.0,4.0)
--   puntoMedio (-1,2) (7,6) == (3.0,4.0)
-----
```

```
puntoMedio (x1,y1) (x2,y2) = ((x1+x2)/2, (y1+y2)/2)
```

```
-- -----
-- Ejercicio 10. Los números complejos pueden representarse mediante
-- pares de números complejos. Por ejemplo, el número  $2+5i$  puede
-- representarse mediante el par (2,5).
-----
```

```
-- Ejercicio 10.1. Definir la función sumaComplejos tal que
```

```
-- (sumaComplejos x y) es la suma de los números complejos x e y. Por
-- ejemplo,
-- sumaComplejos (2,3) (5,6) == (7,9)
-----
```

```
sumaComplejos (a,b) (c,d) = (a+c, b+d)
-----
```

```
-- Ejercicio 10.2. Definir la función productoComplejos tal que
-- (productoComplejos x y) es el producto de los números complejos x e
-- y. Por ejemplo,
-- productoComplejos (2,3) (5,6) == (-8,27)
-----
```

```
productoComplejos (a,b) (c,d) = (a*c-b*d, a*d+b*c)
-----
```

```
-- Ejercicio 10.3. Definir la función conjugado tal que (conjugado x) es
-- el conjugado del número complejo z. Por ejemplo,
-- conjugado (2,3) == (2,-3)
-----
```

```
conjugado (a,b) = (a,-b)
-----
```

```
-- Ejercicio 11. Definir la función intercala que reciba dos listas xs e
-- ys de dos elementos cada una, y devuelva una lista de cuatro
-- elementos, construida intercalando los elementos de xs e ys. Por
-- ejemplo,
-- intercala [1,4] [3,2] == [1,3,4,2]
-----
```

```
intercala [x1,x2] [y1,y2] = [x1,y1,x2,y2]
-----
```

```
-- Ejercicio 12. Definir una función ciclo que permute cíclicamente los
-- elementos de una lista, pasando el último elemento al principio de la
-- lista. Por ejemplo,
-- ciclo [2, 5, 7, 9] == [9,2,5,7]
-- ciclo [] == [9,2,5,7]
```

```
-- ciclo [2] == [2]
```

```
ciclo [] = []
```

```
ciclo xs = last xs : init xs
```

```
-- -----
-- Ejercicio 13. Definir la función numeroMayor tal que
-- (numeroMayor x y) es el mayor número de dos cifras que puede
-- construirse con los dígitos x e y. Por ejemplo,
-- numeroMayor 2 5 == 52
-- numeroMayor 5 2 == 52
-- -----
```

```
numeroMayor x y = a*10 + b
  where a = max x y
        b = min x y
```

```
-- -----
-- Ejercicio 14. Definir la función numeroDeRaices tal que
-- (numeroDeRaices a b c) es el número de raíces reales de la ecuación
--  $a*x^2 + b*x + c = 0$ . Por ejemplo,
-- numeroDeRaices 2 0 3 == 0
-- numeroDeRaices 4 4 1 == 1
-- numeroDeRaices 5 23 12 == 2
-- -----
```

```
numeroDeRaices a b c | d < 0    = 0
                    | d == 0    = 1
                    | otherwise = 2
  where d = b^2 - 4*a*c
```

```
-- -----
-- Ejercicio 15. (Raíces de una ecuación de segundo grado) Definir la
-- función raices de forma que (raices a b c) devuelve la lista de las
-- raíces reales de la ecuación  $ax^2 + bx + c = 0$ . Por ejemplo,
-- raices 1 (-2) 1 == [1.0, 1.0]
-- raices 1 3 2 == [-1.0, -2.0]
-- -----
```

```
-- 1ª solución
raices_1 a b c = [(-b+d)/t,(-b-d)/t]
  where d = sqrt (b^2 - 4*a*c)
        t = 2*a

-- 2ª solución
raices_2 a b c
  | d >= 0    = [(-b+e)/(2*a), (-b-e)/(2*a)]
  | otherwise = error "No tiene raíces reales"
  where d = b^2-4*a*c
        e = sqrt d

-----
-- Ejercicio 16. En geometría, la fórmula de Herón, descubierta por
-- Herón de Alejandría, dice que el área de un triángulo cuyo lados
-- miden a, b y c es la raíz cuadrada de  $s(s-a)(s-b)(s-c)$  donde s es el
-- semiperímetro
--    $s = (a+b+c)/2$ 
-- Definir la función area tal que (area a b c) es el área de un
-- triángulo de lados a, b y c. Por ejemplo,
--   area 3 4 5 == 6.0
-----

area a b c = sqrt (s*(s-a)*(s-b)*(s-c))
  where s = (a+b+c)/2

-----
-- Ejercicio 17. Los números racionales pueden representarse mediante
-- pares de números enteros. Por ejemplo, el número 2/5 puede
-- representarse mediante el par (2,5).
-----
-- Ejercicio 17.1. Definir la función formaReducida tal que
-- (formaReducida x) es la forma reducida del número racional x. Por
-- ejemplo,
--   formaReducida (4,10) == (2,5)
-----

formaReducida (a,b) = (a 'div' c, b 'div' c)
  where c = gcd a b
```

```
-----  
-- Ejercicio 17.2. Definir la función sumaRacional tal que  
-- (sumaRacional x y) es la suma de los números racionales x e y. Por ejemplo,  
-- sumaRacional (2,3) (5,6) == (3,2)  
-----
```

```
sumaRacional (a,b) (c,d) = formaReducida (a*d+b*c, b*d)
```

```
-----  
-- Ejercicio 17.3. Definir la función productoRacional tal que  
-- (productoRacional x y) es el producto de los números racionales x e  
-- y. Por ejemplo,  
-- productoRacional (2,3) (5,6) == (5,9)  
-----
```

```
productoRacional (a,b) (c,d) = formaReducida (a*c, b*d)
```

```
-----  
-- Ejercicio 17.4. Definir la función igualdadRacional tal que  
-- (igualdadRacional x y) se verifica si los números racionales x e  
-- y son iguales. Por ejemplo,  
-- igualdadRacional (6,9) (10,15) == True  
-- igualdadRacional (6,9) (11,15) == False  
-----
```

```
igualdadRacional (a,b) (c,d) =  
  formaReducida (a,b) == formaReducida (c,d)
```

# Relación 4

## Definiciones por comprensión (1)

```
-- -----  
-- Introducción --  
-- -----
```

```
-- En esta relación se presentan ejercicios con definiciones por  
-- comprensión correspondientes al tema 5 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-5.pdf
```

```
-- -----  
-- Ejercicio 1. Definir, por comprensión, la función  
-- sumaDeCuadrados :: Integer -> Integer  
-- tal que (sumaDeCuadrados n) es la suma de los cuadrados de los  
-- primeros n números; es decir,  $1^2 + 2^2 + \dots + n^2$ . Por ejemplo,  
-- sumaDeCuadrados 3 == 14  
-- sumaDeCuadrados 100 == 338350  
-- -----
```

```
sumaDeCuadrados :: Integer -> Integer  
sumaDeCuadrados n = sum [x^2 | x <- [1..n]]
```

```
-- -----  
-- Ejercicio 2. Definir por comprensión la función  
-- replica :: Int -> a -> [a]  
-- tal que (replica n x) es la lista formada por n copias del elemento
```

```
-- x. Por ejemplo,
--   replica 3 True == [True, True, True]
-- Nota: La función replica es equivalente a la predefinida replicate.
```

```
-----
replica :: Int -> a -> [a]
replica n x = [x | _ <- [1..n]]
```

```
-----
-- Ejercicio 3.1. Definir la función suma tal (suma n) es la suma de los
-- n primeros números. Por ejemplo,
--   suma 3 == 6
```

```
-----
suma n = sum [1..n]
```

```
-- Otra definición más eficiente es
```

```
suma2 n = (1+n)*n 'div' 2
```

```
-----
-- Ejercicio 3.2. Los triángulos aritméticos se forman como sigue
```

```
--   1
--   2 3
--   4 5 6
--   7 8 9 10
--  11 12 13 14 15
--  16 16 18 19 20 21
```

```
-- Definir la función línea tal que (línea n) es la línea n-ésima de los
-- triángulos aritméticos. Por ejemplo,
```

```
--   línea 4 == [7,8,9,10]
--   línea 5 == [11,12,13,14,15]
```

```
-----
línea n = [suma (n-1)+1..suma n]
```

```
-- La definición puede mejorarse
```

```
línea2 n = [s+1..s+n]
           where s = suma (n-1)
```

```
-- Una variante más eficiente es
```



```
linea3 n = [s+1..s+n]
          where s = suma2 (n-1)
```

```
-- La mejora de la eficiencia se puede observar como sigue:
```

```
-- ghci> :set +s
-- ghci> head (linea 1000000)
-- 499999500001
-- (17.94 secs, 309207420 bytes)
-- ghci> head (linea3 1000000)
-- 499999500001
-- (0.01 secs, 525496 bytes)
```

```
-----
-- Ejercicio 3.3. Definir la función triangulo tal que (triangulo n) es
-- el triángulo aritmético de altura n. Por ejemplo,
-- triangulo 3 == [[1],[2,3],[4,5,6]]
-- triangulo 4 == [[1],[2,3],[4,5,6],[7,8,9,10]]
-----
```

```
triangulo n = [linea m | m <- [1..n]]
```

```
-----
-- Ejercicio 4. Un entero positivo es perfecto si es igual a la suma de
-- sus factores, excluyendo el propio número.
```

```
-- Definir por comprensión la función
-- perfectos :: Int -> [Int]
-- tal que (perfectos n) es la lista de todos los números perfectos
-- menores que n. Por ejemplo,
-- perfectos 500 == [6,28,496]
-- Indicación: Usar la función factores del tema 5.
```

```
-- La función factores del tema es
```

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

```
-- La definición es
```

```
perfectos :: Int -> [Int]
perfectos n = [x | x <- [1..n], sum (init (factores x)) == x]
```

```

-----
-- Ejercicio 5. Un número natural  $n$  se denomina abundante si es menor
-- que la suma de sus divisores propios. Por ejemplo, 12 y 30 son
-- abundantes pero 5 y 28 no lo son.
-----

```

```

-----
-- Ejercicio 5.1. Definir la función numeroAbundante tal que
-- (numeroAbundante n) se verifica si  $n$  es un número abundante. Por
-- ejemplo,

```

```

--     numeroAbundante 5  == False
--     numeroAbundante 12 == True
--     numeroAbundante 28 == False
--     numeroAbundante 30 == True
-----

```

```

divisores :: Int -> [Int]
divisores n = [m | m <- [1..n-1], n `mod` m == 0]

```

```

numeroAbundante :: Int -> Bool
numeroAbundante n = n < sum (divisores n)

```

```

-----
-- Ejercicio 5.2. Definir la función numerosAbundantesMenores tal que
-- (numerosAbundantesMenores n) es la lista de números abundantes
-- menores o iguales que  $n$ . Por ejemplo,
--     numerosAbundantesMenores 50 == [12,18,20,24,30,36,40,42,48]
-----

```

```

numerosAbundantesMenores :: Int -> [Int]
numerosAbundantesMenores n = [x | x <- [1..n], numeroAbundante x]

```

```

-----
-- Ejercicio 5.3. Definir la función todosPares tal que (todosPares n)
-- se verifica si todos los números abundantes menores o iguales que  $n$ 
-- son pares. Por ejemplo,
--     todosPares 10    == True
--     todosPares 100  == True
--     todosPares 1000 == False
-----

```

```
todosPares :: Int -> Bool
```

```
todosPares n = and [even x | x <- numerosAbundantesMenores n]
```

```
-----  
-- Ejercicio 5.4. Definir la constante primerAbundanteImpar que calcule  
-- el primer número natural abundante impar. Determinar el valor de  
-- dicho número.  
-----
```

```
primerAbundanteImpar :: Int
```

```
primerAbundanteImpar = head [x | x <- [1..], numeroAbundante x, odd x]
```

```
-- Su cálculo es  
-- ghci> primerAbundanteImpar  
-- 945  
-----
```

```
-- Ejercicio 6 (Problema 1 del proyecto Euler) Definir la función  
-- euler1 :: Integer -> Integer  
-- (euler1 n) es la suma de todos los múltiplos de 3 ó 5 menores que  
-- n. Por ejemplo,  
-- euler1 10 == 23  
--  
-- Calcular la suma de todos los múltiplos de 3 ó 5 menores que 1000.  
-----
```

```
euler1 :: Integer -> Integer
```

```
euler1 n = sum [x | x <- [1..n-1], multiplo x 3 || multiplo x 5]  
  where multiplo x y = mod x y == 0
```

```
-- Cálculo:  
-- ghci> euler1 1000  
-- 233168  
-----
```

```
-- Ejercicio 7. Definir la función  
-- circulo :: Int -> Int  
-- tal que (circulo n) es el la cantidad de pares de números naturales  
-- (x,y) que se encuentran dentro del círculo de radio n. Por ejemplo,  
-- circulo 3 == 9  
-----
```

```

--   circulo 4 == 15
--   circulo 5 == 22
-----

circulo :: Int -> Int
circulo n = length [(x,y) | x <- [0..n], y <- [0..n], x^2+y^2 < n^2]

-- La eficiencia puede mejorarse con
circulo' :: Int -> Int
circulo' n = length [(x,y) | x <- [0..m], y <- [0..m], x^2+y^2 < n^2]
    where m = raizCuadradaEntera n

-- (raizCuadradaEntera n) es la parte entera de la raíz cuadrada de
-- n. Por ejemplo,
--   raizCuadradaEntera 17 == 4
raizCuadradaEntera :: Int -> Int
raizCuadradaEntera n = truncate (sqrt (fromIntegral n))

-----

-- Ejercicio 8.1. Definir la función aproxE tal que (aproxE n) es la
-- lista cuyos elementos son los términos de la sucesión  $(1+1/m)^m$ 
-- desde 1 hasta n. Por ejemplo,
--   aproxE 1 == [2.0]
--   aproxE 4 == [2.0,2.25,2.37037037037037,2.44140625]
-----

aproxE n = [(1+1/m)**m | m <- [1..n]]

-----

-- Ejercicio 8.2. ¿Cuál es el límite de la sucesión  $(1+1/m)^m$  ?
-----

-- El límite de la sucesión es el número e.

-----

-- Ejercicio 8.3. Definir la función errorE tal que (errorE x) es el
-- menor número de términos de la sucesión  $(1+1/m)^m$  necesarios para
-- obtener su límite con un error menor que x. Por ejemplo,
--   errorAproxE 0.1 == 13.0
--   errorAproxE 0.01 == 135.0

```

```
--      errorAproxE 0.001 == 1359.0
--      Indicación: En Haskell, e se calcula como (exp 1).
-----

errorAproxE x = head [m | m <- [1..], abs((exp 1) - (1+1/m)**m) < x]
```

```
-----
--      Ejercicio 9.1. Definir la función aproxLimSeno tal que
--      (aproxLimSeno n) es la lista cuyos elementos son los términos de la
--      sucesión
--      sen(1/m)
--      -----
--      1/m
--      desde 1 hasta n. Por ejemplo,
--      aproxLimSeno 1 == [0.8414709848078965]
--      aproxLimSeno 2 == [0.8414709848078965, 0.958851077208406]
-----
```

```
aproxLimSeno n = [sin(1/m)/(1/m) | m <- [1..n]]
```

```
-----
--      Ejercicio 9.2. ¿Cuál es el límite de la sucesión sen(1/m)/(1/m) ?
-----
```

```
--      El límite es 1.
```

```
-----
--      Ejercicio 9.3. Definir la función errorLimSeno tal que
--      (errorLimSeno x) es el menor número de términos de la sucesión
--      sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
--      que x. Por ejemplo,
--      errorLimSeno 0.1 == 2.0
--      errorLimSeno 0.01 == 5.0
--      errorLimSeno 0.001 == 13.0
--      errorLimSeno 0.0001 == 41.0
-----
```

```
errorLimSeno x = head [m | m <- [1..], abs(1 - sin(1/m)/(1/m)) < x]
```

```
-- Ejercicio 10.1. Definir la función calculaPi tal que (calculaPi n) es
-- la aproximación del número pi calculada mediante la expresión
--  $4*(1 - 1/3 + 1/5 - 1/7 + \dots + (-1)**n/(2*n+1))$ 
-- Por ejemplo,
-- calculaPi 3 == 2.8952380952380956
-- calculaPi 300 == 3.1449149035588526
```

```
calculaPi n = 4 * sum [(-1)**x/(2*x+1) | x <- [0..n]]
```

```
-- Ejercicio 10.2. Definir la función errorPi tal que
-- (errorPi x) es el menor número de términos de la serie
--  $4*(1 - 1/3 + 1/5 - 1/7 + \dots + (-1)**n/(2*n+1))$ 
-- necesarios para obtener pi con un error menor que x. Por ejemplo,
-- errorPi 0.1 == 9.0
-- errorPi 0.01 == 99.0
-- errorPi 0.001 == 999.0
```

```
errorPi x = head [n | n <- [1..], abs (pi - (calculaPi n)) < x]
```

# Relación 5

## Definiciones por comprensión (2)

```
-----  
-- Introducción  
-----  
  
-- En esta relación se presentan ejercicios con definiciones por  
-- comprensión correspondientes al tema 5 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-5.pdf  
  
-----  
-- Ejercicio 1.1. Una terna (x,y,z) de enteros positivos es pitagórica  
-- si  $x^2 + y^2 = z^2$ . Usando una lista por comprensión, definir la  
-- función  
-- pitagoricas :: Int -> [(Int,Int,Int)]  
-- tal que (pitagoricas n) es la lista de todas las ternas pitagóricas  
-- cuyas componentes están entre 1 y n. Por ejemplo,  
-- pitagoricas 10 == [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]  
-----  
  
pitagoricas :: Int -> [(Int,Int,Int)]  
pitagoricas n = [(x,y,z) | x <- [1..n],  
                          y <- [1..n],  
                          z <- [1..n],  
                          x2 + y2 == z2]
```

```

-----
-- Ejercicio 1.2. Definir la función
--   numeroDePares :: (Int,Int,Int) -> Int
-- tal que (numeroDePares t) es el número de elementos pares de la terna
-- t. Por ejemplo,
--   numeroDePares (3,5,7) == 0
--   numeroDePares (3,6,7) == 1
--   numeroDePares (3,6,4) == 2
--   numeroDePares (4,6,4) == 3
-----

```

```

numeroDePares :: (Int,Int,Int) -> Int
numeroDePares (x,y,z) = length [1 | n <- [x,y,z], even n]

```

```

-----
-- Ejercicio 1.3. Definir la función
--   conjetura :: Int -> Bool
-- tal que (conjetura n) se verifica si todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n tiene un número impar de números
-- pares. Por ejemplo,
--   conjetura 10 == True
-----

```

```

conjetura :: Int -> Bool
conjetura n = and [odd (numeroDePares t) | t <- pitagoricas n]

```

```

-----
-- Ejercicio 1.4. Demostrar la conjetura para todas las ternas
-- pitagóricas.
-----

```

```

-- Sea (x,y,z) una terna pitagórica. Entonces  $x^2+y^2=z^2$ . Pueden darse
-- 4 casos:
--
-- Caso 1: x e y son pares. Entonces,  $x^2$ ,  $y^2$  y  $z^2$  también lo
-- son. Luego el número de componentes pares es 3 que es impar.
--
-- Caso 2: x es par e y es impar. Entonces,  $x^2$  es par,  $y^2$  es impar y
--  $z^2$  es impar. Luego el número de componentes pares es 1 que es impar.
--

```



```

-- Caso 3: x es impar e y es par. Análogo al caso 2.
--
-- Caso 4: x e y son impares. Entonces, x^2 e y^2 también son impares y
-- z^2 es par. Luego el número de componentes pares es 1 que es impar.
--
-----
-- Ejercicio 2.1. (Problema 9 del Proyecto Euler). Una terna pitagórica
-- es una terna de números naturales (a,b,c) tal que a<b<c y
-- a^2+b^2=c^2. Por ejemplo (3,4,5) es una terna pitagórica.
--
-- Definir la función
--   ternasPitagoricas :: Integer -> [[Integer]]
-- tal que (ternasPitagoricas x) es la lista de las ternas pitagóricas
-- cuya suma es x. Por ejemplo,
--   ternasPitagoricas 12 == [(3,4,5)]
--   ternasPitagoricas 60 == [(10,24,26),(15,20,25)]
--
-----

ternasPitagoricas :: Integer -> [(Integer,Integer,Integer)]
ternasPitagoricas x = [(a,b,c) | a <- [1..x],
                                b <- [a+1..x],
                                c <- [x-a-b],
                                a^2 + b^2 == c^2]

--
-----
-- Ejercicio 2.2. Definir la constante euler9 tal que euler9 es producto
-- abc donde (a,b,c) es la única terna pitagórica tal que a+b+c=1000.
-- Calcular el valor de euler9.
--
-----

euler9 = a*b*c
  where (a,b,c) = head (ternasPitagoricas 1000)

-- El cálculo del valor de euler9 es
--   ghci> euler9
--   31875000
--
-----
-- Ejercicio 3. El producto escalar de dos listas de enteros xs y ys de
-- longitud n viene dado por la suma de los productos de los elementos

```

```

-- correspondientes.
--
-- Definir por comprensión la función
-- productoEscalar :: [Int] -> [Int] -> Int
-- tal que (productoEscalar xs ys) es el producto escalar de las listas
-- xs e ys. Por ejemplo,
-- productoEscalar [1,2,3] [4,5,6] == 32
-----

productoEscalar :: [Int] -> [Int] -> Int
productoEscalar xs ys = sum [x*y | (x,y) <- zip xs ys]

-----

-- Ejercicio 4. Definir, por comprensión, la función
-- sumaConsecutivos :: [Int] -> [Int]
-- tal que (sumaConsecutivos xs) es la suma de los pares de elementos
-- consecutivos de la lista xs. Por ejemplo,
-- sumaConsecutivos [3,1,5,2] == [4,6,7]
-- sumaConsecutivos [3] == []
-----

sumaConsecutivos :: [Int] -> [Int]
sumaConsecutivos xs = [x+y | (x,y) <- zip xs (tail xs)]

-----

-- Ejercicio 5. En el tema se ha definido la función
-- posiciones :: Eq a => a -> [a] -> [Int]
-- tal que (posiciones x xs) es la lista de las posiciones ocupadas por
-- el elemento x en la lista xs. Por ejemplo,
-- posiciones 5 [1,5,3,5,5,7] == [1,3,4]
--
-- Definir, usando la función busca (definida en el tema 5), la función
-- posiciones' :: Eq a => a -> [a] -> [Int]
-- tal que posiciones' sea equivalente a posiciones.
-----

-- La definición de posiciones es
posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
  [i | (x',i) <- zip xs [0..n], x == x']

```

```

    where n = length xs - 1

-- La definición de busca es
busca :: Eq a => a -> [(a, b)] -> [b]
busca c t = [v | (c', v) <- t, c' == c]

-- La redefinición de posiciones es
posiciones' :: Eq a => a -> [a] -> [Int]
posiciones' x xs = busca x (zip xs [0..])

-----
-- Ejercicio 6. Los polinomios pueden representarse de forma dispersa o
-- densa. Por ejemplo, el polinomio  $6x^4-5x^2+4x-7$  se puede representar
-- de forma dispersa por [6,0,-5,4,-7] y de forma densa por
-- [(4,6),(2,-5),(1,4),(0,-7)].
--
-- Definir la función
--   densa :: [Int] -> [(Int,Int)]
-- tal que (densa xs) es la representación densa del polinomio cuya
-- representación dispersa es xs. Por ejemplo,
--   densa [6,0,-5,4,-7] == [(4,6),(2,-5),(1,4),(0,-7)]
--   densa [6,0,0,3,0,4] == [(5,6),(2,3),(0,4)]
-----

densa :: [Int] -> [(Int,Int)]
densa xs = [(x,y) | (x,y) <- zip [n-1,n-2..0] xs, y /= 0]
    where n = length xs

-----
-- Ejercicio 7. La función
--   pares :: [a] -> [b] -> [(a,b)]
-- definida por
--   pares xs ys = [(x,y) | x <- xs, y <- ys]
-- toma como argumento dos listas y devuelve la listas de los pares con
-- el primer elemento de la primera lista y el segundo de la
-- segunda. Por ejemplo,
--   ghci> pares [1..3] [4..6]
--   [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
--
-- Definir, usando dos listas por comprensión con un generador cada una,

```

```

-- la función
-- pares' :: [a] -> [b] -> [(a,b)]
-- tal que pares' sea equivalente a pares.
--
-- Indicación: Utilizar la función predefinida concat y encajar una
-- lista por comprensión dentro de la otra.
-----

-- La definición de pares es
pares :: [a] -> [b] -> [(a,b)]
pares xs ys = [(x,y) | x <- xs, y <- ys]

-- La redefinición de pares es
pares' :: [a] -> [b] -> [(a,b)]
pares' xs ys = concat [(x,y) | y <- ys] | x <- xs]

-----

-- Ejercicio 8. La bases de datos sobre actividades de personas pueden
-- representarse mediante listas de elementos de la forma (a,b,c,d),
-- donde a es el nombre de la persona, b su actividad, c su fecha de
-- nacimiento y d la de su fallecimiento. Un ejemplo es la siguiente que
-- usaremos a lo largo de este ejercicio,
-----

personas :: [(String,String,Int,Int)]
personas = [("Cervantes","Literatura",1547,1616),
            ("Velazquez","Pintura",1599,1660),
            ("Picasso","Pintura",1881,1973),
            ("Beethoven","Musica",1770,1823),
            ("Poincare","Ciencia",1854,1912),
            ("Quevedo","Literatura",1580,1654),
            ("Goya","Pintura",1746,1828),
            ("Einstein","Ciencia",1879,1955),
            ("Mozart","Musica",1756,1791),
            ("Botticelli","Pintura",1445,1510),
            ("Borromini","Arquitectura",1599,1667),
            ("Bach","Musica",1685,1750)]

-----

-- Ejercicio 8.1. Definir la función nombres tal que (nombres bd) es

```

```
-- la lista de los nombres de las personas de la base de datos bd. Por
-- ejemplo,
-- ghci> nombres personas
-- ["Cervantes","Velazquez","Picasso","Beethoven","Poincare",
-- "Quevedo","Goya","Einstein","Mozart","Botticelli","Borromini","Bach"]
-----
```

```
nombres :: [(String,String,Int,Int)] -> [String]
nombres bd = [x | (x,_,_,_) <- bd]
```

```
-- Ejercicio 8.2. Definir la función músicos tal que (músicos bd) es
-- la lista de los nombres de los músicos de la base de datos bd. Por
-- ejemplo,
-- ghci> músicos personas
-- ["Beethoven","Mozart","Bach"]
-----
```

```
músicos :: [(String,String,Int,Int)] -> [String]
músicos bd = [x | (x,"Musica",_,_) <- bd]
```

```
-- Ejercicio 8.3. Definir la función seleccion tal que (seleccion bd m)
-- es la lista de los nombres de las personas de la base de datos bd
-- cuya actividad es m. Por ejemplo,
-- ghci> seleccion personas "Pintura"
-- ["Velazquez","Picasso","Goya","Botticelli"]
-- ghci> seleccion personas "Musica"
-- ["Beethoven","Mozart","Bach"]
-----
```

```
seleccion :: [(String,String,Int,Int)] -> String -> [String]
seleccion bd m = [ x | (x,m',_,_) <- bd, m == m' ]
```

```
-- Ejercicio 8.4. Definir, usando el apartado anterior, la función
-- músicos' tal que (músicos' bd) es la lista de los nombres de los
-- músicos de la base de datos bd. Por ejemplo,
-- ghci> músicos' personas
-- ["Beethoven","Mozart","Bach"]
```

---

```
musicos' :: [(String,String,Int,Int)] -> [String]
musicos' bd = seleccion bd "Musica"
```

---

```
-- Ejercicio 8.5. Definir la función vivas tal que (vivas bd a) es la
-- lista de los nombres de las personas de la base de datos bd que
-- estaban vivas en el año a. Por ejemplo,
-- ghci> vivas personas 1600
-- ["Cervantes","Velazquez","Quevedo","Borromini"]
```

---

```
vivas :: [(String,String,Int,Int)] -> Int -> [String]
vivas ps a = [x | (x,_,a1,a2) <- ps, a1 <= a, a <= a2]
```

# Relación 6

## Definiciones por comprensión con cadenas: El cifrado César

```
-- -----  
-- Introducción --  
-- -----  
  
-- En el tema 5, cuyas transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-5.pdf  
-- se estudió, como aplicación de las definiciones por comprensión, el  
-- cifrado César. El objetivo de esta relación es modificar el programa  
-- de cifrado César para que pueda utilizar también letras  
-- mayúsculas. Por ejemplo,  
-- *Main> descifra "Ytit Ufwf Sfif"  
-- "Todo Para Nada"  
-- Para ello, se propone la modificación de las funciones  
-- correspondientes del tema 5.  
  
-- -----  
-- Importación de librerías auxiliares --  
-- -----  
  
import Data.Char  
  
-- (minuscuala2int c) es el entero correspondiente a la letra minúscula  
-- c. Por ejemplo,  
-- minuscuala2int 'a' == 0  
-- minuscuala2int 'd' == 3
```

```

--      minuscula2int 'z' == 25
minuscula2int :: Char -> Int
minuscula2int c = ord c - ord 'a'

--      (mayuscula2int c) es el entero correspondiente a la letra mayúscula
--      c. Por ejemplo,
--      mayuscula2int 'A' == 0
--      mayuscula2int 'D' == 3
--      mayuscula2int 'Z' == 25
mayuscula2int :: Char -> Int
mayuscula2int c = ord c - ord 'A'

--      (int2minuscula n) es la letra minúscula correspondiente al entero
--      n. Por ejemplo,
--      int2minuscula 0 == 'a'
--      int2minuscula 3 == 'd'
--      int2minuscula 25 == 'z'
int2minuscula :: Int -> Char
int2minuscula n = chr (ord 'a' + n)

--      (int2mayuscula n) es la letra minúscula correspondiente al entero
--      n. Por ejemplo,
--      int2mayuscula 0 == 'A'
--      int2mayuscula 3 == 'D'
--      int2mayuscula 25 == 'Z'
int2mayuscula :: Int -> Char
int2mayuscula n = chr (ord 'A' + n)

--      (desplaza n c) es el carácter obtenido desplazando n caracteres el
--      carácter c. Por ejemplo,
--      desplaza 3 'a' == 'd'
--      desplaza 3 'y' == 'b'
--      desplaza (-3) 'd' == 'a'
--      desplaza (-3) 'b' == 'y'
--      desplaza 3 'A' == 'D'
--      desplaza 3 'Y' == 'B'
--      desplaza (-3) 'D' == 'A'
--      desplaza (-3) 'B' == 'Y'
desplaza :: Int -> Char -> Char
desplaza n c

```



```

| elem c ['a'..'z'] = int2minusculta ((minusculta2int c+n) `mod` 26)
| elem c ['A'..'Z'] = int2mayusculta ((mayusculta2int c+n) `mod` 26)
| otherwise         = c

-- (codifica n xs) es el resultado de codificar el texto xs con un
-- desplazamiento n. Por ejemplo,
-- *Main> codifica 3 "En Todo La Medida"
-- "Hq Wrgr Od Phlgd"
-- *Main> codifica (-3) "Hq Wrgr Od Phlgd"
-- "En Todo La Medida"
codifica :: Int -> String -> String
codifica n xs = [desplaza n x | x <- xs]

-- tabla es la lista de la frecuencias de las letras en castellano, Por
-- ejemplo, la frecuencia de la 'a' es del 12.53%, la de la 'b' es
-- 1.42%.
tabla :: [Float]
tabla = [12.53, 1.42, 4.68, 5.86, 13.68, 0.69, 1.01,
         0.70, 6.25, 0.44, 0.01, 4.97, 3.15, 6.71,
         8.68, 2.51, 0.88, 6.87, 7.98, 4.63, 3.93,
         0.90, 0.02, 0.22, 0.90, 0.52]

-- (porcentaje n m) es el porcentaje de n sobre m. Por ejemplo,
-- porcentaje 2 5 == 40.0
porcentaje :: Int -> Int -> Float
porcentaje n m = (fromIntegral n / fromIntegral m) * 100

-- (letras xs) es la cadena formada por las letras de la cadena xs. Por
-- ejemplo,
-- letras "Esto Es Una Prueba" == "EstoEsUnaPrueba"
letras :: String -> String
letras xs = [x | x <- xs, elem x (['a'..'z']++['A'..'Z'])]

-- (ocurrencias x xs) es el número de veces que ocurre el carácter x en
-- la cadena xs. Por ejemplo,
-- ocurrencias 'a' "Salamanca" == 4
ocurrencias :: Char -> String -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']

-- (frecuencias xs) es la frecuencia de cada una de las letras de la

```

```

-- cadena xs. Por ejemplo,
--   *Main> frecuencias "En Todo La Medida"
--   [14.3,0,0,21.4,14.3,0,0,0,7.1,0,0,7.1,
--     7.1,7.1,14.3,0,0,0,0,7.1,0,0,0,0,0,0]
frecuencias :: String -> [Float]
frecuencias xs =
  [porcentaje (ocurrencias x xs') n | x <- ['a'..'z']]
  where xs' = [toLower x | x <- xs]
        n   = length (letras xs)

-- (chiCud os es) es la medida chi cuadrado de las distribuciones os y
-- es. Por ejemplo,
--   chiCud [3,5,6] [3,5,6] == 0.0
--   chiCud [3,5,6] [5,6,3] == 3.9666667
chiCud :: [Float] -> [Float] -> Float
chiCud os es = sum [(o-e)^2/e | (o,e) <- zip os es]

-- (rota n xs) es la lista obtenida rotando n posiciones los elementos
-- de la lista xs. Por ejemplo,
--   rota 2 "manolo" == "noloma"
rota :: Int -> [a] -> [a]
rota n xs = drop n xs ++ take n xs

-- (descifra xs) es la cadena obtenida descodificando la cadena xs por
-- el anti-desplazamiento que produce una distribución de letras con la
-- menor desviación chi cuadrado respecto de la tabla de distribución de
-- las letras en castellano. Por ejemplo,
--   *Main> codifica 5 "Todo Para Nada"
--   "Ytit Ufwf Sfif"
--   *Main> descifra "Ytit Ufwf Sfif"
--   "Todo Para Nada"
descifra :: String -> String
descifra xs = codifica (-factor) xs
  where
    factor = head (posiciones (minimum tabChi) tabChi)
    tabChi = [chiCud (rota n tabla') tabla | n <- [0..25]]
    tabla' = frecuencias xs

posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =

```

---

```
[i | (x',i) <- zip xs [0..], x == x']
```



# Relación 7

## Definiciones por recursión

```
-----  
-- Introducción -----  
-----  
  
-- En esta relación se presentan ejercicios con definiciones por  
-- recursión correspondientes al tema 6 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-6.pdf  
-----  
  
-- Ejercicio 1. Definir por recursión la función  
-- potencia :: Integer -> Integer -> Integer  
-- tal que (potencia x n) es x elevado al número natural n. Por ejemplo,  
-- potencia 2 3 == 8  
-----  
  
potencia :: Integer -> Integer -> Integer  
potencia m 0 = 1  
potencia m n = m*(potencia m (n-1))  
-----  
  
-- Ejercicio 2. Definir por recursión la función  
-- replicate' :: Int -> a -> [a]  
-- tal que (replicate' n x) es la lista formado por n copias del  
-- elemento x. Por ejemplo,  
-- replicate' 3 2 == [2,2,2]  
-----
```

```

replicate' :: Int -> a -> [a]
replicate' 0 _      = []
replicate' n x = x : replicate' (n-1) x

```

```

-----
-- Ejercicio 3. El doble factorial de un número n se define por
--   n!! = n*(n-2)* ... * 3 * 1, si n es impar
--   n!! = n*(n-2)* ... * 4 * 2, si n es par
--   1!! = 1
--   0!! = 1
-- Por ejemplo,
--   8!! = 8*6*4*2 = 384
--   9!! = 9*7*5*3*1 = 945
-- Definir, por recursión, la función
--   dobleFactorial :: Integer -> Integer
-- tal que (dobleFactorial n) es el doble factorial de n. Por ejemplo,
--   dobleFactorial 8 == 384
--   dobleFactorial 9 == 945
-----

```

```

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

```

```

-----
-- Ejercicio 4. Dados dos números naturales, a y b, es posible
-- calcular su máximo común divisor mediante el Algoritmo de
-- Euclides. Este algoritmo se puede resumir en la siguiente fórmula:
--   mcd(a,b) = a,                si b = 0
--             = mcd (b, a módulo b), si b > 0
--
-- Definir la función
--   mcd :: Integer -> Integer -> Integer
-- tal que (mcd a b) es el máximo común divisor de a y b calculado
-- mediante el algoritmo de Euclides. Por ejemplo,
--   mcd 30 45 == 15
-----

```

```
mcd :: Integer -> Integer -> Integer
mcd a 0 = a
mcd a b = mcd b (a `mod` b)

-----
-- Ejercicio 5. (Problema 5 del proyecto Euler) El problema se encuentra
-- en http://goo.gl/L5bb y consiste en calcular el menor número
-- divisible por los números del 1 al 20. Lo resolveremos mediante los
-- distintos apartados de este ejercicio.
-----

-- Ejercicio 5.1. Definir por recursión la función
-- menorDivisible :: Integer -> Integer -> Integer
-- tal que (menorDivisible a b) es el menor número divisible por los
-- números del a al b. Por ejemplo,
-- menorDivisible 2 5 == 60
-- Indicación: Usar la función lcm tal que (lcm x y) es el mínimo común
-- múltiplo de x e y.
-----

menorDivisible :: Integer -> Integer -> Integer
menorDivisible a b
  | a == b    = a
  | otherwise = lcm a (menorDivisible (a+1) b)

-----
-- Ejercicio 5.2. Definir la constante
-- euler5 :: Integer
-- tal que euler5 es el menor número divisible por los números del 1 al
-- 20 y calcular su valor.
-----

euler5 :: Integer
euler5 = menorDivisible 1 20

-- El cálculo es
-- ghci> euler5
-- 232792560
```

```

-----
-- Ejercicio 6. En un templo hindú se encuentran tres varillas de
-- platino. En una de ellas, hay 64 anillos de oro de distintos radios,
-- colocados de mayor a menor.
--
-- El trabajo de los monjes de ese templo consiste en pasarlos todos a
-- la tercera varilla, usando la segunda como varilla auxiliar, con las
-- siguientes condiciones:
--   * En cada paso sólo se puede mover un anillo.
--   * Nunca puede haber un anillo de mayor diámetro encima de uno de
--     menor diámetro.
-- La leyenda dice que cuando todos los anillos se encuentren en la
-- tercera varilla, será el fin del mundo.
--
-- Definir la función
--   numPasosHanoi :: Integer -> Integer
-- tal que (numPasosHanoi n) es el número de pasos necesarios para
-- trasladar n anillos. Por ejemplo,
--   numPasosHanoi 2 == 3
--   numPasosHanoi 7 == 127
--   numPasosHanoi 64 == 18446744073709551615
-----

-- Sean A, B y C las tres varillas. La estrategia recursiva es la
-- siguiente:
-- * Caso base (N=1): Se mueve el disco de A a C.
-- * Caso inductivo (N=M+1): Se mueven M discos de A a C. Se mueve el disco
--   de A a B. Se mueven M discos de C a B.
-- Por tanto,

numPasosHanoi :: Integer -> Integer
numPasosHanoi 1 = 1
numPasosHanoi n = 1 + 2 * numPasosHanoi (n-1)

-----
-- Ejercicio 7. Definir por recursión la función
--   and' :: [Bool] -> Bool
-- tal que (and' xs) se verifica si todos los elementos de xs son
-- verdadero. Por ejemplo,
--   and' [1+2 < 4, 2:[3] == [2,3]] == True

```



```
-- and' [1+2 < 3, 2:[3] == [2,3]] == False
```

```
and' :: [Bool] -> Bool
and' [] = True
and' (b:bs) = b && and' bs
```

```
-- Ejercicio 8. Definir por recursión la función
-- elem' :: Eq a => a -> [a] -> Bool
-- tal que (elem' x xs) se verifica si x pertenece a la lista xs. Por
-- ejemplo,
-- elem' 3 [2,3,5] == True
-- elem' 4 [2,3,5] == False
```

```
elem' :: Eq a => a -> [a] -> Bool
elem' x [] = False
elem' x (y:ys) | x == y = True
                | otherwise = elem' x ys
```

```
-- Ejercicio 9. Definir por recursión la función
-- last' :: [a] -> a
-- tal que (last' xs) es el último elemento de xs. Por ejemplo,
-- last' [2,3,5] => 5
```

```
last' :: [a] -> a
last' [x] = x
last' (_:xs) = last' xs
```

```
-- Ejercicio 10. Definir por recursión la función
-- concat' :: [[a]] -> [a]
-- tal que (concat' xss) es la lista obtenida concatenando las listas de
-- xss. Por ejemplo,
-- concat' [[1..3],[5..7],[8..10]] == [1,2,3,5,6,7,8,9,10]
```

```
concat' :: [[a]] -> [a]
concat' []      = []
concat' (xs:xss) = xs ++ concat' xss
```

```
-----
-- Ejercicio 11. Definir por recursión la función
--   selecciona :: [a] -> Int -> a
-- tal que (selecciona xs n) es el n-ésimo elemento de xs. Por ejemplo,
--   selecciona [2,3,5,7] 2 == 5
-----
```

```
selecciona :: [a] -> Int -> a
selecciona (x:_) 0 = x
selecciona (_:xs) n = selecciona xs (n-1)
```

```
-----
-- Ejercicio 12. Definir por recursión la función
--   take' :: Int -> [a] -> [a]
-- tal que (take' n xs) es la lista de los n primeros elementos de
-- xs. Por ejemplo,
--   take' 3 [4..12] => [4,5,6]
-----
```

```
take' :: Int -> [a] -> [a]
take' 0 _      = []
take' n []     = []
take' n (x:xs) = x : take' (n-1) xs
```

```
-----
-- Ejercicio 13. Definir la función
--   refinada :: [Float] -> [Float]
-- tal que (refinada xs) es la lista obtenida intercalando entre cada
-- dos elementos consecutivos de xs su media aritmética. Por ejemplo,
--   refinada [2,7,1,8] == [2.0,4.5,7.0,4.0,1.0,4.5,8.0]
--   refinada [2]      == [2.0]
--   refinada []       == []
-----
```

```
refinada :: [Float] -> [Float]
refinada (x:y:zs) = x : (x+y)/2 : refinada (y:zs)
```

`refinada xs`      `= xs`



# Relación 8

## Definiciones por recursión: Ordenación por mezcla

```
-----  
-- Introducción  
-----  
  
-- El objetivo de esta relación es definir la ordenación por mezclas y  
-- comprobar su corrección con QuickCheck.  
  
import Test.QuickCheck  
  
-----  
-- Ejercicio 14.1. Definir por recursión la función  
-- mezcla :: Ord a => [a] -> [a] -> [a]  
-- tal que (mezcla xs ys) es la lista obtenida mezclando las listas  
-- ordenadas xs e ys. Por ejemplo,  
-- mezcla [2,5,6] [1,3,4] == [1,2,3,4,5,6]  
-----  
  
mezcla :: Ord a => [a] -> [a] -> [a]  
mezcla [] ys = ys  
mezcla xs [] = xs  
mezcla (x:xs) (y:ys) | x <= y = x : mezcla xs (y:ys)  
                    | otherwise = y : mezcla (x:xs) ys  
  
-----  
-- Ejercicio 14.2. Definir la función
```

```
-- mitades :: [a] -> ([a],[a])
-- tal que (mitades xs) es el par formado por las dos mitades en que se
-- divide xs tales que sus longitudes difieren como máximo en uno. Por
-- ejemplo,
-- mitades [2,3,5,7,9] == ([2,3],[5,7,9])
-----
```

```
mitades :: [a] -> ([a],[a])
mitades xs = splitAt (length xs `div` 2) xs
```

```
-----
-- Ejercicio 14.3. Definir por recursión la función
-- ordMezcla :: Ord a => [a] -> [a]
-- tal que (ordMezcla xs) es la lista obtenida ordenando xs por mezcla
-- (es decir, considerando que la lista vacía y las listas unitarias
-- están ordenadas y cualquier otra lista se ordena mezclando las dos
-- listas que resultan de ordenar sus dos mitades por separado). Por
-- ejemplo,
-- ordMezcla [5,2,3,1,7,2,5] == [1,2,2,3,5,5,7]
-----
```

```
ordMezcla :: Ord a => [a] -> [a]
ordMezcla [] = []
ordMezcla [x] = [x]
ordMezcla xs = mezcla (ordMezcla ys) (ordMezcla zs)
               where (ys,zs) = mitades xs
```

```
-----
-- Ejercicio 14.4. Definir por recursión la función
-- ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada. Por
-- ejemplo,
-- ordenada [2,3,5] == True
-- ordenada [2,5,3] == False
-----
```

```
ordenada :: Ord a => [a] -> Bool
ordenada [] = True
ordenada [_] = True
ordenada (x:y:xs) = x <= y && ordenada (y:xs)
```

```

-----
-- Ejercicio 14.5. Comprobar con QuickCheck que la ordenación por mezcla
-- de una lista es una lista ordenada.
-----

-- La propiedad es
prop_ordMezcla_ordenada :: [Int] -> Bool
prop_ordMezcla_ordenada xs = ordenada (ordMezcla xs)

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

-----
-- Ejercicio 14.6. Definir por recursión la función
-- borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
-- x en la lista xs. Por ejemplo,
-- borra 1 [1,2,1] == [2,1]
-- borra 3 [1,2,1] == [1,2,1]
-----

borra :: Eq a => a -> [a] -> [a]
borra x [] = []
borra x (y:ys) | x == y = ys
                | otherwise = y : borra x ys

-----
-- Ejercicio 14.7. Definir por recursión la función
-- esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutación de
-- ys. Por ejemplo,
-- esPermutacion [1,2,1] [2,1,1] == True
-- esPermutacion [1,2,1] [1,2,2] == False
-----

esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion [] [] = True
esPermutacion [] (y:ys) = False

```

```
esPermutacion (x:xs) ys = elem x ys && esPermutacion xs (borra x ys)
```

```
-----  
-- Ejercicio 14.8. Comprobar con QuickCheck que la ordenación por mezcla  
-- de una lista es una permutación de la lista.  
-----
```

```
-- La propiedad es
```

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

```
prop_ordMezcla_pemutacion xs = esPermutacion (ordMezcla xs) xs
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_ordMezcla_pemutacion
```

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



# Relación 9

## Definiciones por recursión y por comprensión (1)

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se presentan ejercicios con dos definiciones (una  
-- por recursión y otra por comprensión) y la comprobación de la  
-- equivalencia de las dos definiciones con QuickCheck. Los ejercicios  
-- corresponden a los temas 5 y 6 cuyas transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-5.pdf  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-6.pdf  
  
-- -----  
-- Importación de librerías auxiliares --  
-- -----  
  
import Test.QuickCheck  
import Data.List  
  
-- -----  
-- Ejercicio 1.1. Definir, por recursión, la función  
-- sumaCuadradosR :: Integer -> Integer  
-- tal que (sumaCuadradosR n) es la suma de los cuadrados de los números  
-- de 1 a n. Por ejemplo,  
-- sumaCuadradosR 4 == 30  
-- -----
```

```
sumaCuadradosR :: Integer -> Integer
```

```
sumaCuadradosR 0 = 0
```

```
sumaCuadradosR n = n^2 + sumaCuadradosR (n-1)
```

```
-----
-- Ejercicio 1.2. Comprobar con QuickCheck si sumaCuadradosR n es igual a
--  $n(n+1)(2n+1)/6$ .
-----
```

```
-- La propiedad es
```

```
prop_SumaCuadrados n =
```

```
  n >= 0 ==>
```

```
    sumaCuadradosR n == n * (n+1) * (2*n+1) `div` 6
```

```
-- La comprobación es
```

```
-- Main> quickCheck prop_SumaCuadrados
```

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

```
-----
-- Ejercicio 1.3. Definir, por comprensión, la función
```

```
-- sumaCuadradosC :: Integer --> Integer
```

```
-- tal que (sumaCuadradosC n) es la suma de los cuadrados de los números  
-- de 1 a n. Por ejemplo,
```

```
-- sumaCuadradosC 4 == 30
```

```
sumaCuadradosC :: Integer -> Integer
```

```
sumaCuadradosC n = sum [x^2 | x <- [1..n]]
```

```
-----
-- Ejercicio 1.4. Comprobar con QuickCheck que las funciones
```

```
-- sumaCuadradosR y sumaCuadradosC son equivalentes sobre los números  
-- naturales.
```

```
-- La propiedad es
```

```
prop_sumaCuadradosR n =
```

```
  n >= 0 ==> sumaCuadradosR n == sumaCuadradosC n
```

```
-- La comprobación es
-- *Main> quickCheck prop_sumaCuadrados
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 2.1. Se quiere formar una escalera con bloques cuadrados,
-- de forma que tenga un número determinado de escalones. Por ejemplo,
-- una escalera con tres escalones tendría la siguiente forma:
```

```
--     XX
--     XXXX
--     XXXXXX
```

```
-- Definir, por recursión, la función
```

```
-- numeroBloquesR :: Integer -> Integer
-- tal que (numeroBloquesR n) es el número de bloques necesarios para
-- construir una escalera con n escalones. Por ejemplo,
-- numeroBloquesR 1 == 2
-- numeroBloquesR 3 == 12
-- numeroBloquesR 10 == 110
```

```
-----
numeroBloquesR :: Integer -> Integer
numeroBloquesR 0 = 0
numeroBloquesR n = 2*n + numeroBloquesR (n-1)
```

```
-----
-- Ejercicio 2.2. Definir, por comprensión, la función
```

```
-- numeroBloquesC :: Integer -> Integer
-- tal que (numeroBloquesC n) es el número de bloques necesarios para
-- construir una escalera con n escalones. Por ejemplo,
-- numeroBloquesC 1 == 2
-- numeroBloquesC 3 == 12
-- numeroBloquesC 10 == 110
```

```
-----
numeroBloquesC :: Integer -> Integer
numeroBloquesC n = sum [2*x | x <- [1..n]]
```

```
-----
-- Ejercicio 2.3. Comprobar con QuickCheck que (numeroBloquesC n) es
-- igual a  $n+n^2$ .
```

```

-----
-- La propiedad es
prop_numeroBloquesR n =
  n > 0 ==> numeroBloquesC n == n+n^2

```

```

-- La comprobación es
-- *Main> quickCheck prop_numeroBloques
-- +++ OK, passed 100 tests.

```

```

-----
-- Ejercicio 3.1. Definir, por recursión, la función
-- sumaCuadradosImparesR :: Integer -> Integer
-- tal que (sumaCuadradosImparesR n) es la suma de los cuadrados de los
-- números impares desde 1 hasta n.
-- sumaCuadradosImparesR 1 == 1
-- sumaCuadradosImparesR 7 == 84
-- sumaCuadradosImparesR 4 == 10
-----

```

```

sumaCuadradosImparesR :: Integer -> Integer
sumaCuadradosImparesR 1 = 1
sumaCuadradosImparesR n
  | odd n    = n^2 + sumaCuadradosImparesR (n-1)
  | otherwise = sumaCuadradosImparesR (n-1)

```

```

-----
-- Ejercicio 3.2. Definir, por comprensión, la función
-- sumaCuadradosImparesC :: Integer -> Integer
-- tal que (sumaCuadradosImparesC n) es la suma de los cuadrados de los
-- números impares desde 1 hasta n.
-- sumaCuadradosImparesC 1 == 1
-- sumaCuadradosImparesC 7 == 84
-- sumaCuadradosImparesC 4 == 10
-----

```

```

sumaCuadradosImparesC :: Integer -> Integer
sumaCuadradosImparesC n = sum [x^2 | x <- [1..n], odd x]

```

```

-- Otra definición más simple es

```

```
sumaCuadradosImparesC' :: Integer -> Integer
sumaCuadradosImparesC' n = sum [x^2 | x <- [1,3..n]]
```

```
-----
-- Ejercicio 4.1. Definir, por recursión, la función
--   digitosR :: Integer -> [Integer]
-- tal que (digitosR n) es la lista de los dígitos del número n. Por
-- ejemplo,
--   digitosR 320274 == [3,2,0,2,7,4]
-----
```

```
digitosR :: Integer -> [Integer]
digitosR n = reverse (digitosR' n)
```

```
digitosR' n
  | n < 10    = [n]
  | otherwise = (n 'rem' 10) : digitosR' (n 'div' 10)
```

```
-----
-- Ejercicio 4.2. Definir, por comprensión, la función
--   digitosC :: Integer -> [Integer]
-- tal que (digitosC n) es la lista de los dígitos del número n. Por
-- ejemplo,
--   digitosC 320274 == [3,2,0,2,7,4]
-- Indicación: Usar las funciones show y read.
-----
```

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

```
-----
-- Ejercicio 4.3. Comprobar con QuickCheck que las funciones digitosR y
-- digitosC son equivalentes.
-----
```

```
-- La propiedad es
prop_digitos n =
  n >= 0 ==>
  digitosR n == digitosC n
```

```
-- La comprobación es
-- *Main> quickCheck prop_digitos
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 5.1. Definir, por recursión, la función
-- sumaDigitosR :: Integer -> Integer
-- tal que (sumaDigitosR n) es la suma de los dígitos de n. Por ejemplo,
-- sumaDigitosR 3 == 3
-- sumaDigitosR 2454 == 15
-- sumaDigitosR 20045 == 11
-----
```

```
sumaDigitosR :: Integer -> Integer
sumaDigitosR n
  | n < 10    = n
  | otherwise = n `rem` 10 + sumaDigitosR (n `div` 10)
```

```
-----
-- Ejercicio 5.2. Definir, sin usar recursión, la función
-- sumaDigitosNR :: Integer -> Integer
-- tal que (sumaDigitosNR n) es la suma de los dígitos de n. Por ejemplo,
-- sumaDigitosNR 3 == 3
-- sumaDigitosNR 2454 == 15
-- sumaDigitosNR 20045 == 11
-----
```

```
sumaDigitosNR :: Integer -> Integer
sumaDigitosNR n = sum (digitosC n)
```

```
-----
-- Ejercicio 5.3. Comprobar con QuickCheck que las funciones sumaDigitosR
-- y sumaDigitosNR son equivalentes.
-----
```

```
-- La propiedad es
prop_sumaDigitos n =
  n >= 0 ==>
  sumaDigitosR n == sumaDigitosNR n
```

```
-- La comprobación es
-- *Main> quickCheck prop_sumaDigitos
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 6. Definir la función
-- esDigito :: Integer -> Integer -> Bool
-- tal que (esDigito x n) se verifica si x es un dígito de n. Por
-- ejemplo,
-- esDigito 4 1041 == True
-- esDigito 3 1041 == False
-----
```

```
esDigito :: Integer -> Integer -> Bool
esDigito x n = x `elem` digitosC n
```

```
-----
-- Ejercicio 7. Definir la función
-- numeroDeDigitos :: Integer -> Integer
-- tal que (numeroDeDigitos x) es el número de dígitos de x. Por ejemplo,
-- numeroDeDigitos 34047 == 5
-----
```

```
numeroDeDigitos :: Integer -> Int
numeroDeDigitos x = length (digitosC x)
```

```
-----
-- Ejercicio 8.1 Definir, por recursión, la función
-- listaNumeroR :: [Integer] -> Integer
-- tal que (listaNumeroR xs) es el número formado por los dígitos xs. Por
-- ejemplo,
-- listaNumeroR [5] == 5
-- listaNumeroR [1,3,4,7] == 1347
-- listaNumeroR [0,0,1] == 1
-----
```

```
listaNumeroR :: [Integer] -> Integer
listaNumeroR xs = listaNumeroR' (reverse xs)
```

```
listaNumeroR' :: [Integer] -> Integer
```

```

listaNumeroR' [x]      = x
listaNumeroR' (x:xs) = x + 10 * (listaNumeroR' xs)

```

```

-----
-- Ejercicio 8.2. Definir, por comprensión, la función
--   listaNumeroC :: [Integer] -> Integer
-- tal que (listaNumeroC xs) es el número formado por los dígitos xs. Por
-- ejemplo,
--   listaNumeroC [5]           == 5
--   listaNumeroC [1,3,4,7]    == 1347
--   listaNumeroC [0,0,1]      == 1
-----

```

```

listaNumeroC :: [Integer] -> Integer
listaNumeroC xs = sum [y*10^n | (y,n) <- zip (reverse xs) [0..]]

```

```

-----
-- Ejercicio 9.1. Definir, por recursión, la función
--   pegaNumerosR :: Integer -> Integer -> Integer
-- tal que (pegaNumerosR x y) es el número resultante de "pegar" los
-- números x e y. Por ejemplo,
--   pegaNumerosR 12 987 == 12987
--   pegaNumerosR 1204 7 == 12047
--   pegaNumerosR 100 100 == 100100
-----

```

```

pegaNumerosR :: Integer -> Integer -> Integer
pegaNumerosR x y
  | y < 10    = 10*x+y
  | otherwise = 10 * pegaNumerosR x (y 'div' 10) + (y 'mod' 10)

```

```

-----
-- Ejercicio 9.2. Definir, sin usar recursión, la función
--   pegaNumerosNR :: Integer -> Integer -> Integer
-- tal que (pegaNumerosNR x y) es el número resultante de "pegar" los
-- números x e y. Por ejemplo,
--   pegaNumerosNR 12 987 == 12987
--   pegaNumerosNR 1204 7 == 12047
--   pegaNumerosNR 100 100 == 100100
-----

```



```
pegaNumerosNR :: Integer -> Integer -> Integer
pegaNumerosNR x y = listaNumeroC (digitosC x ++ digitosC y)
```

```
-- Otra definición es
```

```
pegaNumerosNR2 :: Integer -> Integer -> Integer
pegaNumerosNR2 x y = (x * (10^(numeroDeDigitos y))) + y
```

```
-----
-- Ejercicio 9.3. Comprobar con QuickCheck que las funciones
-- pegaNumerosR y pegaNumerosNR son equivalentes.
-----
```

```
-- La propiedad es
```

```
prop_pegaNumeros x y =
  x >= 0 && y >= 0 ==>
  pegaNumerosR x y == pegaNumerosNR x y
```

```
-- La comprobación es
```

```
-- *Main> quickCheck prop_pegaNumeros
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 10.1. Definir, por recursión, la función
-- primerDigitoR :: Integer -> Integer
-- tal que (primerDigitoR n) es el primer dígito de n. Por ejemplo,
-- primerDigitoR 425 == 4
-----
```

```
primerDigitoR :: Integer -> Integer
primerDigitoR n
  | n < 10    = n
  | otherwise = primerDigitoR (n `div` 10)
```

```
-----
-- Ejercicio 10.2. Definir, sin usar recursión, la función
-- primerDigitoNR :: Integer -> Integer
-- tal que (primerDigitoNR n) es la primera digito de n. Por ejemplo,
-- primerDigitoNR 425 == 4
-----
```

```

primerDigitoNR :: Integer -> Integer
primerDigitoNR n = head (digitosC n)

-- Otra definición equivalente es
primerDigitoNR2 :: Integer -> Integer
primerDigitoNR2 n = n `div` 10^((numeroDeDigitos n)-1)

-----
-- Ejercicio 10.3. Comprobar con QuickCheck que las funciones
-- primerDigitoR y primerDigitoNR son equivalentes.
-----

-- La propiedad es
prop_primerDigito x =
  x >= 0 ==>
    primerDigitoR x == primerDigitoNR x

-- La comprobación es
-- *Main> quickCheck prop_primerDigito
-- +++ OK, passed 100 tests.

-----
-- Ejercicio 11. Definir la función
-- ultimoDigito :: Integer -> Integer
-- tal que (ultimoDigito n) es el último dígito de n. Por ejemplo,
-- ultimoDigito 425 == 5
-----

ultimoDigito :: Integer -> Integer
ultimoDigito n = n `rem` 10

-----
-- Ejercicio 12.1. Definir la función
-- inverso :: Integer -> Integer
-- tal que (inverso n) es el número obtenido escribiendo los dígitos de n
-- en orden inverso. Por ejemplo,
-- inverso 42578 == 87524
-- inverso 203 == 302
-----

```

```
inverso :: Integer -> Integer
inverso n = listaNumeroC (reverse (digitosC n))

-----
-- Ejercicio 12.2. Definir, usando show y read, la función
--   inverso' :: Integer -> Integer
-- tal que (inverso' n) es el número obtenido escribiendo los dígitos de n
-- en orden inverso'. Por ejemplo,
--   inverso' 42578 == 87524
--   inverso' 203  ==   302
-----

inverso' :: Integer -> Integer
inverso' n = read (reverse (show n))

-----
-- Ejercicio 12.3. Comprobar con QuickCheck que las funciones
-- inverso e inverso' son equivalentes.
-----

-- La propiedad es
prop_inverso n =
  n >= 0 ==>
  inverso n == inverso' n

-- La comprobación es
-- *Main> quickCheck prop_inverso
-- +++ OK, passed 100 tests.

-----
-- Ejercicio 13. Definir la función
--   capicua :: Integer -> Bool
-- tal que (capicua n) se verifica si los dígitos que n son los mismos
-- de izquierda a derecha que de derecha a izquierda. Por ejemplo,
--   capicua 1234 = False
--   capicua 1221 = True
--   capicua 4   = True
-----
```

```
capicua :: Integer -> Bool
capicua n = n == inverso n
```

```
-- -----
-- Ejercicio 14. (Problema 16 del proyecto Euler) El problema se
-- encuentra en http://goo.gl/4uWh y consiste en calcular la suma de los
-- dígitos de  $2^{1000}$ . Lo resolveremos mediante los distintos apartados de
-- este ejercicio.
-- -----
```

```
-- -----
-- Ejercicio 14.1. Definir la función
--   euler16 :: Integer -> Integer
-- tal que (euler16 n) es la suma de los dígitos de  $2^n$ . Por ejemplo,
--   euler16 4 == 7
-- -----
```

```
euler16 :: Integer -> Integer
euler16 n = sumaDigitosNR (2^n)
```

```
-- -----
-- Ejercicio 14.2. Calcular la suma de los dígitos de  $2^{1000}$ .
-- -----
```

```
-- El cálculo es
--   *Main> euler16 1000
--   1366
-- -----
```

```
-- Ejercicio 15. En el enunciado de uno de los problemas de las
-- Olimpiadas matemáticas de Brasil se define el primitivo de un número
-- como sigue:
```

```
--   Dado un número natural  $N$ , multiplicamos todos sus dígitos,
--   repetimos este procedimiento hasta que quede un solo dígito al
--   cual llamamos primitivo de  $N$ . Por ejemplo para 327:  $3 \times 2 \times 7 = 42$  y
--    $4 \times 2 = 8$ . Por lo tanto, el primitivo de 327 es 8.
--
```

```
-- Definir la función
--   primitivo :: Integer -> Integer
-- tal que (primitivo n) es el primitivo de  $n$ . Por ejemplo.
```

```

-- primitivo 327 == 8
-----

primitivo :: Integer -> Integer
primitivo n | n < 10    = n
            | otherwise = primitivo (producto n)

-- (producto n) es el producto de las cifras de n. Por ejemplo,
-- producto 327 == 42
producto :: Integer -> Integer
producto n = product (digitosC n)

-----

-- Ejercicio 16. Dos números son equivalentes si la media de sus dígitos
-- son iguales. Por ejemplo, 3205 y 41 son equivalentes ya que
--  $(3+2+0+5)/4 = (4+1)/2$ . Definir la función
-- equivalentes :: Integer -> Integer -> Bool
-- tal que (equivalentes x y) se verifica si los números x e y son
-- equivalentes. Por ejemplo,
-- equivalentes 3205 41 == True
-- equivalentes 3205 25 == False
-----

equivalentes :: Integer -> Integer -> Bool
equivalentes x y = media (digitosC x) == media (digitosC y)

-- (media xs) es la media de la lista xs. Por ejemplo,
-- media [3,2,0,5] == 2.5
media :: [Integer] -> Float
media xs = (fromIntegral (sum xs)) / (fromIntegral (length xs))

-----

-- Ejercicio 17. Un número x es especial si el número de ocurrencia de
-- cada dígito d de x en  $x^2$  es el doble del número de ocurrencia de d
-- en x. Por ejemplo, 72576 es especial porque tiene un 2, un 5, un 6 y
-- dos 7 y su cuadrado es 5267275776 que tiene exactamente dos 2, dos 5,
-- dos 6 y cuatro 7.
--
-- Definir la función
-- especial :: Integer -> Bool

```

```
-- tal que (especial x) se verifica si x es un número especial. Por
-- ejemplo,
--     especial 72576 == True
--     especial 12   == False
-- Calcular el menor número especial mayor que 72576.
```

```
-----
especial :: Integer -> Bool
```

```
especial x =
    sort (ys ++ ys) == sort (show (x^2))
  where ys = show x
```

```
-- El cálculo es
-- ghci> head [x | x <- [72577..], especial x]
-- 406512
```

# Relación 10

## Definiciones por recursión y por comprensión (2)

```
-- -----  
-- Introducción --  
-- -----
```

```
-- En esta relación se presentan ejercicios con dos definiciones (una  
-- por recursión y otra por comprensión). Los ejercicios corresponden a  
-- los temas 5 y 6 cuyas transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-5.pdf  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-6.pdf
```

```
-- -----  
-- Ejercicio 1.1. Definir, por comprensión, la función  
-- cuadradosC :: [Integer] -> [Integer]  
-- tal que (cuadradosC xs) es la lista de los cuadrados de xs. Por  
-- ejemplo,  
-- cuadradosC [1,2,3] == [1,4,9]  
-- -----
```

```
cuadradosC :: [Integer] -> [Integer]  
cuadradosC xs = [x^2 | x <- xs]
```

```
-- -----  
-- Ejercicio 1.2. Definir, por recursión, la función  
-- cuadradosR :: [Integer] -> [Integer]  
-- tal que (cuadradosR xs) es la lista de los cuadrados de xs. Por
```

```
-- ejemplo,
--   cuadradosR [1,2,3] == [1,4,9]
```

```
-----
cuadradosR :: [Integer] -> [Integer]
cuadradosR []      = []
cuadradosR (x:xs) = x^2 : cuadradosR xs
```

```
-----
-- Ejercicio 2.1. Definir, por comprensión, la función
--   imparesC :: [Integer] -> [Integer]
-- tal que (imparesC xs) es la lista de los números impares de xs. Por
-- ejemplo,
--   imparesC [1,2,3] == [1,3]
```

```
-----
imparesC :: [Integer] -> [Integer]
imparesC xs = [x | x <- xs, odd x]
```

```
-----
-- Ejercicio 2.2. Definir, por recursión, la función
--   imparesR :: [Integer] -> [Integer]
-- tal que (imparesR xs) es la lista de los números impares de xs. Por
-- ejemplo,
--   imparesR [1,2,3] == [1,3]
```

```
-----
imparesR :: [Integer] -> [Integer]
imparesR [] = []
imparesR (x:xs) | odd x    = x : imparesR xs
                 | otherwise = imparesR xs
```

```
-----
-- Ejercicio 3.1. Definir, por comprensión, la función
--   imparesCuadradosC :: [Integer] -> [Integer]
-- tal que (imparesCuadradosC xs) es la lista de los cuadrados de los
-- números impares de xs. Por ejemplo,
--   imparesCuadradosC [1,2,3] == [1,9]
```



```
imparesCuadradosC :: [Integer] -> [Integer]
imparesCuadradosC xs = [x^2 | x <- xs, odd x]
```

```
-----
-- Ejercicio 3.2. Definir, por recursión, la función
--   imparesCuadradosR :: [Integer] -> [Integer]
-- tal que (imparesCuadradosR xs) es la lista de los cuadrados de los
-- números impares de xs. Por ejemplo,
--   imparesCuadradosR [1,2,3] == [1,9]
-----
```

```
imparesCuadradosR :: [Integer] -> [Integer]
imparesCuadradosR [] = []
imparesCuadradosR (x:xs) | odd x = x^2 : imparesCuadradosR xs
                          | otherwise = imparesCuadradosR xs
```

```
-----
-- Ejercicio 4.1. Definir, por comprensión, la función
--   sumaCuadradosImparesC :: [Integer] -> Integer
-- tal que (sumaCuadradosImparesC xs) es la suma de los cuadrados de los
-- números impares de la lista xs. Por ejemplo,
--   sumaCuadradosImparesC [1,2,3] == 10
-----
```

```
sumaCuadradosImparesC :: [Integer] -> Integer
sumaCuadradosImparesC xs = sum [x^2 | x <- xs, odd x]
```

```
-----
-- Ejercicio 4.2. Definir, por recursión, la función
--   sumaCuadradosImparesR :: [Integer] -> Integer
-- tal que (sumaCuadradosImparesR xs) es la suma de los cuadrados de los
-- números impares de la lista xs. Por ejemplo,
--   sumaCuadradosImparesR [1,2,3] == 10
-----
```

```
sumaCuadradosImparesR :: [Integer] -> Integer
sumaCuadradosImparesR [] = 0
sumaCuadradosImparesR (x:xs)
  | odd x = x^2 + sumaCuadradosImparesR xs
  | otherwise = sumaCuadradosImparesR xs
```

```
-----  
-- Ejercicio 5.1. Definir, usando funciones predefinidas, la función  
--   entreL :: Integer -> Integer -> [Integer]  
-- tal que (entreL m n) es la lista de los números entre m y n. Por  
-- ejemplo,  
--   entreL 2 5 == [2,3,4,5]  
-----
```

```
entreL :: Integer -> Integer -> [Integer]  
entreL m n = [m..n]
```

```
-----  
-- Ejercicio 5.2. Definir, por recursión, la función  
--   entreR :: Integer -> Integer -> [Integer]  
-- tal que (entreR m n) es la lista de los números entre m y n. Por  
-- ejemplo,  
--   entreR 2 5 == [2,3,4,5]  
-----
```

```
entreR :: Integer -> Integer -> [Integer]  
entreR m n | m > n    = []  
           | otherwise = m : entreR (m+1) n
```

# Relación 11

## Definiciones por recursión y por comprensión (4)

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se presentan ejercicios con dos definiciones (una  
-- por recursión y otra por comprensión) y la comprobación de la  
-- equivalencia de las dos definiciones con QuickCheck. Los ejercicios  
-- corresponden a los temas 5 y 6 cuyas transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-11/temas/tema-5.pdf  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-6.pdf  
  
-- -----  
-- Importación de librerías auxiliares --  
-- -----  
  
import Data.List  
import Test.QuickCheck  
  
-- -----  
-- Ejercicio 1.1. Definir, por comprensión, la función  
-- mitadPares :: [Int] -> [Int]  
-- tal que (mitadPares xs) es la lista de las mitades de los elementos  
-- de xs que son pares. Por ejemplo,  
-- mitadPares [0,2,1,7,8,56,17,18] == [0,1,4,28,9]  
-- -----
```

```

mitadPares :: [Int] -> [Int]
mitadPares xs = [x `div` 2 | x <- xs, even x]

-----
-- Ejercicio 1.2. Definir, por recursión, la función
--   mitadParesRec :: [Int] -> [Int]
-- tal que (mitadParesRec []) es la lista de las mitades de los elementos
-- de xs que son pares. Por ejemplo,
--   mitadParesRec [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
-----

mitadParesRec :: [Int] -> [Int]
mitadParesRec [] = []
mitadParesRec (x:xs)
  | even x    = x `div` 2 : mitadParesRec xs
  | otherwise = mitadParesRec xs

-----
-- Ejercicio 1.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-----

-- La propiedad es
prop_mitadPares :: [Int] -> Bool
prop_mitadPares xs =
  mitadPares xs == mitadParesRec xs

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

-- Ejercicio 2.1. Definir, por comprensión, la función
--   enRangoC :: Int -> Int -> [Int] -> [Int]
-- tal que (enRangoC a b xs) es la lista de los elementos de xs mayores o
-- iguales que a y menores o iguales que b. Por ejemplo,
--   enRangoC 5 10 [1..15] == [5,6,7,8,9,10]
--   enRangoC 5 10 [7,2,9,3] == [7,9]
--   enRangoC 10 5 [1..15] == []

```

```
-- enRangoC 5 5 [1..15] == [5]
```

```
-----
enRangoC :: Int -> Int -> [Int] -> [Int]
enRangoC a b xs = [x | x <- xs, a <= x, x <= b]
```

```
-----
-- Ejercicio 2.2. Definir, por recursión, la función
-- enRangoR :: Int -> Int -> [Int] -> [Int]
-- tal que (enRangoR a b []) es la lista de los elementos de xs
-- mayores o iguales que a y menores o iguales que b. Por ejemplo,
-- enRangoR 5 10 [1..15] == [5,6,7,8,9,10]
-- enRangoR 5 10 [7,2,9,3] == [7,9]
-- enRangoR 10 5 [1..15] == []
-- enRangoR 5 5 [1..15] == [5]
```

```
-----
enRangoR :: Int -> Int -> [Int] -> [Int]
enRangoR a b [] = []
enRangoR a b (x:xs)
  | a <= x && x <= b = x : enRangoR a b xs
  | otherwise       = enRangoR a b xs
```

```
-----
-- Ejercicio 2.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
```

```
-- La propiedad es
```

```
prop_enRangoC :: Int -> Int -> [Int] -> Bool
prop_enRangoC a b xs =
  enRangoC a b xs == enRangoR a b xs
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_enRango
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 3.1. Definir, por comprensión, la función
-- sumaPositivosC :: [Int] -> Int
```

```
-- tal que (sumaPositivosC xs) es la suma de los números positivos de
-- xs. Por ejemplo,
-- sumaPositivosC [0,1,-3,-2,8,-1,6] == 15
-----
```

```
sumaPositivosC :: [Int] -> Int
sumaPositivosC xs = sum [x | x <- xs, x > 0]
```

```
-- Ejercicio 3.2. Definir, por recursión, la función
-- sumaPositivosR :: [Int] -> Int
-- tal que (sumaPositivosR xs) es la suma de los números positivos de
-- xs. Por ejemplo,
-- sumaPositivosR [0,1,-3,-2,8,-1,6] == 15
-----
```

```
sumaPositivosR :: [Int] -> Int
sumaPositivosR [] = 0
sumaPositivosR (x:xs) | x > 0 = x + sumaPositivosR xs
                      | otherwise = sumaPositivosR xs
```

```
-- Ejercicio 3.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-----
```

```
-- La propiedad es
prop_sumaPositivosC :: [Int] -> Bool
prop_sumaPositivosC xs =
  sumaPositivosC xs == sumaPositivosR xs
```

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

```
-- Ejercicio 4.1. La suma de la serie
--  $1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$ 
-- es  $\pi^2/6$ . Por tanto,  $\pi$  se puede aproximar mediante la raíz cuadrada
-- de 6 por la suma de la serie.
```

```
--
-- Definir, por comprensión, la función aproximaPiC tal que
-- (aproximaPiC n) es la aproximación de pi obtenida mediante n
-- términos de la serie. Por ejemplo,
--   aproximaPiC 4    == sqrt(6*(1/1^2 + 1/2^2 + 1/3^2 + 1/4^2))
--                   == 2.9226129861250305
--   aproximaPiC 1000 == 3.1406380562059946
-----
```

```
aproximaPiC n = sqrt(6*sum [1/x^2 | x <- [1..n]])
```

```
--
-- Ejercicio 4.2. Definir, por recursión, la función aproximaPiR tal
-- que (aproximaPiR n) es la aproximación de pi obtenida mediante n
-- términos de la serie. Por ejemplo,
--   aproximaPiR 4    == sqrt(6*(1/1^2 + 1/2^2 + 1/3^2 + 1/4^2))
--                   == 2.9226129861250305
--   aproximaPiR 1000 == 3.1406380562059946
-----
```

```
aproximaPiR n = sqrt(6*aproximaPiR' n)
```

```
aproximaPiR' 1 = 1
aproximaPiR' n = 1/n^2 + aproximaPiR' (n-1)
```

```
--
-- Ejercicio 5.1. Definir por recursión la función
--   sustituyeImpar :: [Int] -> [Int]
-- tal que (sustituyeImpar xs) es la lista obtenida sustituyendo cada
-- número impar de xs por el siguiente número par. Por ejemplo,
--   sustituyeImpar [2,5,7,4] == [2,6,8,4]
-----
```

```
sustituyeImpar :: [Int] -> [Int]
sustituyeImpar []      = []
sustituyeImpar (x:xs) | odd x    = (x+1): sustituyeImpar xs
                      | otherwise = x:sustituyeImpar xs
```

```
--
-- Ejercicio 5.2. Comprobar con QuickChek la siguiente propiedad: para
```

```
-- cualquier lista de números enteros xs, todos los elementos de la
-- lista (sustituyeImpar xs) son números pares.
```

```
-- La propiedad es
```

```
prop_sustituyeImpar :: [Int] -> Bool
prop_sustituyeImpar xs = and [even x | x <- sustituyeImpar xs]
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_sustituyeImpar
-- +++ OK, passed 100 tests.
```

```
-- Ejercicio 6.1. Una persona es tan agarrada que sólo compra cuando le
-- hacen un descuento del 10% y el precio (con el descuento) es menor o
-- igual que 199.
```

```
-- Definir, usando comprensión, la función
```

```
-- agarradoC :: [Float] -> Float
-- tal que (agarradoC ps) es el precio que tiene que pagar por una compra
-- cuya lista de precios es ps. Por ejemplo,
-- agarradoC [45.00, 199.00, 220.00, 399.00] == 417.59998
```

```
agarradoC :: [Float] -> Float
```

```
agarradoC ps = sum [p * 0.9 | p <- ps, p * 0.9 <= 199]
```

```
-- Ejercicio 6.2. Definir, por recursión, la función
```

```
-- agarradoR :: [Float] -> Float
-- tal que (agarradoR ps) es el precio que tiene que pagar por una compra
-- cuya lista de precios es ps. Por ejemplo,
-- agarradoR [45.00, 199.00, 220.00, 399.00] == 417.59998
```

```
agarradoR :: [Float] -> Float
```

```
agarradoR [] = 0
```

```
agarradoR (p:ps)
```

```
  | precioConDescuento <= 199 = precioConDescuento + agarradoR ps
```

```
  | otherwise                  = agarradoR ps
```



```
where precioConDescuento = p * 0.9
```

```
-----  
-- Ejercicio 6.3. Comprobar con QuickCheck que ambas definiciones son  
-- similares; es decir, el valor absoluto de su diferencia es menor que  
-- una décima.  
-----
```

```
-- La propiedad es
```

```
prop_agarrado :: [Float] -> Bool
```

```
prop_agarrado xs = abs (agarradoR xs - agarradoC xs) <= 0.1
```

```
-- La comprobación es
```

```
-- *Main> quickCheck prop_agarrado
```

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

```
-----  
-- Ejercicio 7.1. Definir la función
```

```
-- factores :: Integer -> Integer
```

```
-- tal que (factores n) es la lista de los factores de n. Por ejemplo,
```

```
-- factores 60 == [1,2,3,4,5,6,10,12,15,20,30,60]  
-----
```

```
factores :: Integer -> [Integer]
```

```
factores n = [x | x <- [1..n], rem n x == 0]
```

```
-----  
-- Ejercicio 7.2. Definir la función
```

```
-- primo :: Integer -> Bool
```

```
-- tal que (primo n) se verifica si n es primo. Por ejemplo,
```

```
-- primo 7 == True
```

```
-- primo 9 == False  
-----
```

```
primo :: Integer -> Bool
```

```
primo x = factores x == [1,x]
```

```
-----  
-- Ejercicio 7.3. Definir la función
```

```
-- factoresPrimos :: Integer -> [Integer]
```

```
-- tal que (factoresPrimos n) es la lista de los factores primos de
-- n. Por ejemplo,
--   factoresPrimos 60 == [2,3,5]
```

```
-----
factoresPrimos :: Integer -> [Integer]
factoresPrimos n = [x | x <- factores n, primo x]
```

```
-----
-- Ejercicio 8.1. Definir, por recursión, la función
--   mayorExponenteR :: Integer -> Integer -> Integer
-- tal que (mayorExponenteR a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
--   mayorExponenteR 2 8    == 3
--   mayorExponenteR 2 9    == 0
--   mayorExponenteR 5 100  == 2
--   mayorExponenteR 2 60   == 2
```

```
-----
mayorExponenteR :: Integer -> Integer -> Integer
mayorExponenteR a b
  | rem b a /= 0 = 0
  | otherwise   = 1 + mayorExponenteR a (b `div` a)
```

```
-----
-- Ejercicio 8.2. Definir, por recursión, la función
--   mayorExponenteC :: Integer -> Integer -> Integer
-- tal que (mayorExponenteC a b) es el exponente de la mayor potencia de
-- a que divide a b. Por ejemplo,
--   mayorExponenteC 2 8    == 3
--   mayorExponenteC 5 100  == 2
--   mayorExponenteC 5 101  == 0
```

```
-----
mayorExponenteC :: Integer -> Integer -> Integer
mayorExponenteC a b = head [x-1 | x <- [0..], mod b (a^x) /= 0]
```

```
-----
-- Ejercicio 8.3. Definir la función
--   factorizacion :: Integer -> [(Integer,Integer)]
```

```
-- tal que (factorizacion n) es la factorización de n. Por ejemplo,
--   factorizacion 60 == [(2,2),(3,1),(5,1)]
-----
```

```
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n = [(x,mayorExponenteR x n) | x <- factoresPrimos n]
-----
```

```
-- Ejercicio 8.4. Definir, por recursión, la función
--   expansionR :: [(Integer,Integer)] -> Integer
-- tal que (expansionR xs) es la expansión de la factorización de
-- xs. Por ejemplo,
--   expansionR [(2,2),(3,1),(5,1)] == 60
-----
```

```
expansionR :: [(Integer,Integer)] -> Integer
expansionR [] = 1
expansionR ((x,y):zs) = x^y * expansionR zs
-----
```

```
-- Ejercicio 8.5. Definir, por comprensión, la función
--   expansionC :: [(Integer,Integer)] -> Integer
-- tal que (expansionC xs) es la expansión de la factorización de
-- xs. Por ejemplo,
--   expansionC [(2,2),(3,1),(5,1)] == 60
-----
```

```
expansionC :: [(Integer,Integer)] -> Integer
expansionC xs = product [x^y | (x,y) <- xs]
-----
```

```
-- Ejercicio 8.6. Definir la función
--   prop_factorizacion :: Integer -> Bool
-- tal que (prop_factorizacion n) se verifica si para todo número
-- natural x, menor o igual que n, se tiene que
-- (expansionC (factorizacion x)) es igual a x. Por ejemplo,
--   prop_factorizacion 100 == True
-----
```

```
prop_factorizacion n =
```

```

and [expansionC (factorizacion x) == x | x <- [1..n]]

-----
-- Ejercicio 9.1. [Problema 357 del Project Euler] Un número natural n
-- es especial si para todo divisor d de n, d+n/d es primo. Definir la
-- función
--   especial :: Integer -> Bool
-- tal que (especial x) se verifica si x es especial. Por ejemplo,
--   especial 30 == True
--   especial 20 == False
-----

especial :: Integer -> Bool
especial x = and [primo (d + x 'div' d) | d <- factores x]

-----
-- Ejercicio 9.2. Definir la función
--   sumaEspeciales :: Integer -> Integer
-- tal que (sumaEspeciales n) es la suma de los números especiales
-- menores o iguales que n. Por ejemplo,
--   sumaEspeciales 100 == 401
-----

-- Por comprensión
sumaEspeciales :: Integer -> Integer
sumaEspeciales n = sum [x | x <- [1..n], especial x]

-- Por recursión
sumaEspecialesR :: Integer -> Integer
sumaEspecialesR 0 = 0
sumaEspecialesR n | especial n = n + sumaEspecialesR (n-1)
                  | otherwise  = sumaEspecialesR (n-1)

-----
-- Ejercicio 10.1. La distancia de Hamming entre dos listas es el número
-- de posiciones en que los correspondientes elementos son
-- distintos. Por ejemplo, la distancia de Hamming entre "roma" y "loba"
-- es 2 (porque hay 2 posiciones en las que los elementos
-- correspondientes son distintos: la 1ª y la 3ª).
--

```

```
-- Definir por comprensión la función
-- distancia :: Eq a => [a] -> [a] -> Int
-- tal que (distanciaC xs ys) es la distancia de Hamming entre xs e
-- ys. Por ejemplo,
-- distanciaC "romano" "comino" == 2
-- distanciaC "romano" "camino" == 3
-- distanciaC "roma" "comino" == 2
-- distanciaC "roma" "camino" == 3
-- distanciaC "romano" "ron" == 1
-- distanciaC "romano" "cama" == 2
-- distanciaC "romano" "rama" == 1
```

```
-----
distanciaC :: Eq a => [a] -> [a] -> Int
distanciaC xs ys = sum [1 | (x,y) <- zip xs ys, x /= y]
```

```
-----
-- Ejercicio 10.2. Definir por recursión la función
-- distanciaR :: Eq a => [a] -> [a] -> Int
-- tal que (distanciaR xs ys) es la distancia de Hamming entre xs e
-- ys. Por ejemplo,
-- distanciaR "romano" "comino" == 2
-- distanciaR "romano" "camino" == 3
-- distanciaR "roma" "comino" == 2
-- distanciaR "roma" "camino" == 3
-- distanciaR "romano" "ron" == 1
-- distanciaR "romano" "cama" == 2
-- distanciaR "romano" "rama" == 1
```

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

```
-----
-- Ejercicio 11. Definir la función
-- traspuesta :: [[a]] -> [[a]]
-- tal que (traspuesta m) es la traspuesta de la matriz m. Por ejemplo,
```

```

--   traspuesta [[1,2,3],[4,5,6]]    == [[1,4],[2,5],[3,6]]
--   traspuesta [[1,4],[2,5],[3,6]] == [[1,2,3],[4,5,6]]
-----

traspuesta :: [[a]] -> [[a]]
traspuesta ([]:_) = []
traspuesta xss =
  primeros xss : traspuesta (restos xss)

-- (primeros xss) es la lista de los primeros elementos de xss. Por
-- ejemplo,
--   primeros [[1,2,3],[4,5,6]]    == [1,4]
--   primeros [[1,4],[2,5],[3,6]] == [1,2,3]
primeros xss = [head xs | xs <- xss]

-- (restos xss) es la lista de los restos de xss. Por ejemplo,
--   restos [[1,2,3],[4,5,6]]    == [[2,3],[5,6]]
--   restos [[1,4],[2,5],[3,6]] == [[4],[5],[6]]
restos xss  = [tail xs | xs <- xss]

-----

-- Ejercicio 12. [2.5 puntos] Definir la función
--   sumas :: Int -> [Int] -> [Int]
-- tal que (sumas n xs) es la lista de los números que se pueden obtener
-- como suma de n, o menos, elementos de xs. Por ejemplo,
--   sumas 0 [2,5]    == [0]
--   sumas 1 [2,5]    == [2,5,0]
--   sumas 2 [2,5]    == [4,7,2,10,5,0]
--   sumas 3 [2,5]    == [6,9,4,12,7,2,15,10,5,0]
--   sumas 2 [2,3,5]  == [4,5,7,2,6,8,3,10,5,0]
-----

sumas :: Int -> [Int] -> [Int]
sumas 0 _ = [0]
sumas _ [] = [0]
sumas n (x:xs) = [x+y | y <- sumas (n-1) (x:xs)] ++ sumas n xs

```

# Relación 12

## Funciones sobre cadenas

```
-----  
-- Importación de librerías auxiliares --  
-----  
  
import Data.Char  
import Data.List  
import Test.QuickCheck  
  
-----  
-- Ejercicio 1.1. Definir, por comprensión, la función  
-- sumaDigitosC :: String -> Int  
-- tal que (sumaDigitosC xs) es la suma de los dígitos de la cadena  
-- xs. Por ejemplo,  
-- sumaDigitosC "SE 2431 X" == 10  
-- Nota: Usar las funciones (isDigit c) que se verifica si el carácter c  
-- es un dígito y (digitToInt d) que es el entero correspondiente al  
-- dígito d.  
-----  
  
sumaDigitosC :: String -> Int  
sumaDigitosC xs = sum [digitToInt x | x <- xs, isDigit x]  
  
-----  
-- Ejercicio 1.2. Definir, por recursión, la función  
-- sumaDigitosR :: String -> Int  
-- tal que (sumaDigitosR xs) es la suma de los dígitos de la cadena  
-- xs. Por ejemplo,
```

```
-- sumaDigitosR "SE 2431 X" == 10
-- Nota: Usar las funciones isDigit y digitToInt.
```

```
-----
sumaDigitosR :: String -> Int
sumaDigitosR [] = 0
sumaDigitosR (x:xs)
  | isDigit x = digitToInt x + sumaDigitosR xs
  | otherwise = sumaDigitosR xs
```

```
-----
-- Ejercicio 1.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
```

```
-----
-- La propiedad es
prop_sumaDigitosC :: String -> Bool
prop_sumaDigitosC xs =
  sumaDigitosC xs == sumaDigitosR xs
```

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

```
-----
-- Ejercicio 2.1. Definir, por comprensión, la función
-- mayusculaInicial :: String -> String
-- tal que (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
-- mayusculaInicial "sEviLLa" == "Sevilla"
-- Nota: Usar las funciones (toLower c) que es el carácter c en
-- minúscula y (toUpper c) que es el carácter c en mayúscula.
```

```
-----
mayusculaInicial :: String -> String
mayusculaInicial [] = []
mayusculaInicial (x:xs) = toUpper x : [toLower x | x <- xs]
```

```
-----
-- Ejercicio 2.2. Definir, por recursión, la función
```



```
-- mayusculaInicialRec :: String -> String
-- tal que (mayusculaInicialRec xs) es la palabra xs con la letra
-- inicial en mayúscula y las restantes en minúsculas. Por ejemplo,
-- mayusculaInicialRec "sEviLLa" == "Sevilla"
-----
```

```
mayusculaInicialRec :: String -> String
mayusculaInicialRec [] = []
mayusculaInicialRec (x:xs) = toUpper x : aux xs
  where aux (x:xs) = toLower x : aux xs
        aux []     = []
-----
```

```
-- Ejercicio 2.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-----
```

```
-- La propiedad es
prop_mayusculaInicial :: String -> Bool
prop_mayusculaInicial xs =
  mayusculaInicial xs == mayusculaInicialRec xs
-----
```

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

```
-- Ejercicio 3.1. Se consideran las siguientes reglas de mayúsculas
-- iniciales para los títulos:
```

```
-- * la primera palabra comienza en mayúscula y
-- * todas las palabras que tienen 4 letras como mínimo empiezan
-- con mayúsculas
```

```
-- Definir, por comprensión, la función
```

```
-- titulo :: [String] -> [String]
-- tal que (titulo ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
-- ghci> titulo ["eL", "arTE", "DE", "La", "proGraMacion"]
-- ["El", "Arte", "de", "la", "Programacion"]
-----
```

```

titulo :: [String] -> [String]
titulo []      = []
titulo (p:ps) = mayusculaInicial p : [transforma p | p <- ps]

-- (transforma p) es la palabra p con mayúscula inicial si su longitud
-- es mayor o igual que 4 y es p en minúscula en caso contrario
transforma :: String -> String
transforma p | length p >= 4 = mayusculaInicial p
              | otherwise     = minuscula p

-- (minuscula xs) es la palabra xs en minúscula.
minuscula :: String -> String
minuscula xs = [toLower x | x <- xs]

-----
-- Ejercicio 3.2. Definir, por recursión, la función
--   tituloRec :: [String] -> [String]
-- tal que (tituloRec ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--   ghci> tituloRec ["eL","arTE","DE","La","proGraMacion"]
--   ["El","Arte","de","la","Programacion"]
-----

tituloRec :: [String] -> [String]
tituloRec []      = []
tituloRec (p:ps) = mayusculaInicial p : tituloRecAux ps
  where tituloRecAux []      = []
        tituloRecAux (p:ps) = transforma p : tituloRecAux ps

-----
-- Ejercicio 3.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-----

-- La propiedad es
prop_titulo :: [String] -> Bool
prop_titulo xs = titulo xs == tituloRec xs

-- La comprobación es
--   ghci> quickCheck prop_titulo

```

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

-- -----
-- Ejercicio 4.1. Definir, por comprensión, la función
--   buscaCrucigrama :: Char -> Int -> Int -> [String] -> [String]
-- tal que (buscaCrucigrama l pos lon ps) es la lista de las palabras de
-- la lista de palabras ps que tienen longitud lon y poseen la letra l en
-- la posición pos (comenzando en 0). Por ejemplo,
--   ghci> buscaCrucigrama 'c' 1 7 ["ocaso", "casa", "ocupado"]
--   ["ocupado"]
-- -----
```

```
buscaCrucigrama :: Char -> Int -> Int -> [String] -> [String]
```

```
buscaCrucigrama l pos lon ps =
  [p | p <- ps,
       length p == lon,
       0 <= pos, pos < length p,
       p !! pos == l]
```

```
-- -----
-- Ejercicio 4.2. Definir, por recursión, la función
--   buscaCrucigramaR :: Char -> Int -> Int -> [String] -> [String]
-- tal que (buscaCrucigramaR l pos lon ps) es la lista de las palabras
-- de la lista de palabras ps que tienen longitud lon y poseen la letra l
-- en la posición pos (comenzando en 0). Por ejemplo,
--   ghci> buscaCrucigramaR 'c' 1 7 ["ocaso", "acabado", "ocupado"]
--   ["acabado", "ocupado"]
-- -----
```

```
buscaCrucigramaR :: Char -> Int -> Int -> [String] -> [String]
```

```
buscaCrucigramaR letra pos lon [] = []
buscaCrucigramaR letra pos lon (p:ps)
  | length p == lon && 0 <= pos && pos < length p && p !! pos == letra
    = p : buscaCrucigramaR letra pos lon ps
  | otherwise
    = buscaCrucigramaR letra pos lon ps
```

```
-- -----
-- Ejercicio 4.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
```

```

-----
-- La propiedad es
prop_buscaCrucigrama :: Char -> Int -> Int -> [String] -> Bool
prop_buscaCrucigrama letra pos lon ps =
    buscaCrucigrama letra pos lon ps == buscaCrucigramaR letra pos lon ps

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

-----
-- Ejercicio 5.1. Definir, por comprensión, la función
--   posiciones :: String -> Char -> [Int]
-- tal que (posiciones xs y) es la lista de la posiciones del carácter y
-- en la cadena xs. Por ejemplo,
--   posiciones "Salamamca" 'a' == [1,3,5,8]
-----

posiciones :: String -> Char -> [Int]
posiciones xs y = [n | (x,n) <- zip xs [0..], x == y]

-----
-- Ejercicio 5.2. Definir, por recursión, la función
--   posicionesR :: String -> Char -> [Int]
-- tal que (posicionesR xs y) es la lista de la posiciones del
-- carácter y en la cadena xs. Por ejemplo,
--   posicionesR "Salamamca" 'a' == [1,3,5,8]
-----

posicionesR :: String -> Char -> [Int]
posicionesR xs y = posicionesAux xs y 0
  where
    posicionesAux [] y n = []
    posicionesAux (x:xs) y n | x == y    = n : posicionesAux xs y (n+1)
                              | otherwise = posicionesAux xs y (n+1)

-----
-- Ejercicio 5.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.

```

```
-----  
  
-- La propiedad es  
prop_posiciones :: String -> Char -> Bool  
prop_posiciones xs y =  
    posiciones xs y == posicionesR xs y
```

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

```
-----  
  
-- Ejercicio 6.1. Definir, por recursión, la función  
-- contieneR :: String -> String -> Bool  
-- tal que (contieneR xs ys) se verifica si ys es una subcadena de  
-- xs. Por ejemplo,  
-- contieneR "escasamente" "casa" == True  
-- contieneR "escasamente" "cante" == False  
-- contieneR "" "" == True  
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica  
-- si ys es un prefijo de xs.
```

```
-----  
  
contieneR :: String -> String -> Bool  
contieneR _ [] = True  
contieneR [] ys = False  
contieneR xs ys = isPrefixOf ys xs || contieneR (tail xs) ys
```

```
-----  
  
-- Ejercicio 6.2. Definir, por comprensión, la función  
-- contiene :: String -> String -> Bool  
-- tal que (contiene xs ys) se verifica si ys es una subcadena de  
-- xs. Por ejemplo,  
-- contiene "escasamente" "casa" == True  
-- contiene "escasamente" "cante" == False  
-- contiene "casado y casada" "casa" == True  
-- contiene "" "" == True  
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica  
-- si ys es un prefijo de xs.
```

```

contiene :: String -> String -> Bool
contiene xs ys =
    or [isPrefixOf ys zs | zs <- sufijos xs]

-- (sufijos xs) es la lista de sufijos de xs. Por ejemplo,
--   sufijos "abc" == ["abc","bc","c",""]
sufijos :: String -> [String]
sufijos xs = [drop i xs | i <- [0..length xs]]

-- Notas:
-- 1. La función sufijos es equivalente a la predefinida tails.
-- 2. contiene se puede definir usando la predefinida isInfixOf

contiene2 :: String -> String -> Bool
contiene2 xs ys = isInfixOf ys xs

-----
-- Ejercicio 6.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-----

-- La propiedad es
prop_contiene :: String -> String -> Bool
prop_contiene xs ys =
    contieneR xs ys == contiene xs ys

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

```

# Relación 13

## Problemas sobre el 2013

```
-----  
-- Introducción --  
-----
```

```
-- Al comienzo de cada año suelen aparecer problemas matemáticos  
-- relacionados con el número del año. El objetivo de esta relación es  
-- resolver con Haskell problemas relativos al 2013.
```

```
-----  
-- Importación de librerías auxiliares --  
-----
```

```
import Data.List
```

```
-----  
-- Ejercicio 1. [Potencias de 2013] En los apartados de este ejercicio  
-- se resuelven problemas sobre potencias del 2013 como los siguientes:  
-- (a) ¿Cuál es el menor  $n$ , tal que  $n$  aparece en  $2013^n$  en la posición  
--  $n$ ?  
-- (b) ¿Cuál es el menor  $n$  (mayor que 1), tal que dentro  $2013^n$  aparece  
-- el 2013? ¿En qué posición aparece?  
-----
```

```
-----  
-- Ejercicio 1.1. Definir la función  
-- posiciones :: Eq a => [a] -> [a] -> [Int]  
-- tal que (posiciones xs ys) es la lista de las posiciones que ocupa xs
```

```

-- dentro de ys. Por ejemplo,
--   posiciones "ac" "bacdacaec" == [2,5]
-----

posiciones :: Eq a => [a] -> [a] -> [Int]
posiciones xs ys = aux ys 1
  where aux [] n = []
        aux (y:ys) n | isPrefixOf xs (y:ys) = n : aux ys (n+1)
                    | otherwise             = aux ys (n+1)

-----

-- Ejercicio 1.2. Definir la función
--   posicionesEnPotencia :: Integer -> Integer -> [Int]
-- tal que (posicionesEnPotencia n m) es la lista de las posiciones del
-- número n en 2013^m. Por ejemplo,
--   posicionesEnPotencia 1 1 == [3]
--   posicionesEnPotencia 2 2 == [4]
--   posicionesEnPotencia 3 3 == []
--   posicionesEnPotencia 4 4 == [3,11]
--   posicionesEnPotencia 5 5 == [4,11]
--   posicionesEnPotencia 6 6 == [1,2,5]
--   posicionesEnPotencia 58 58 == [26,32,55,59,79]
-----

posicionesEnPotencia :: Integer -> Integer -> [Int]
posicionesEnPotencia n m = posiciones (show n) (show (2013^m))

-----

-- Ejercicio 1.3. Definir el ejercicio
--   enPosicion :: Eq a => [a] -> [a] -> Int -> Bool
-- tal que (enPosicion xs ys n) se verifica si xs está en ys en la
-- posición n. Por ejemplo,
--   enPosicion "ab" "cabdab" 2 == True
--   enPosicion "ab" "cabdab" 5 == True
--   enPosicion "ab" "cabdab" 4 == False
-----

enPosicion :: Eq a => [a] -> [a] -> Int -> Bool
enPosicion xs ys n = isPrefixOf xs (drop (n-1) ys)

```



```
-----  
-- Ejercicio 1.4. Definir el ejercicio  
--   enPosicionPotencia :: Integer -> Integer -> Int -> Bool  
-- tal que (enPosicionPotencia x y n) se verifica si x está en 2013^y en  
-- la posición n. Por ejemplo,  
--   enPosicionPotencia 4 4 3 == True  
--   enPosicionPotencia 4 4 5 == False  
-----
```

```
enPosicionPotencia :: Integer -> Integer -> Int -> Bool  
enPosicionPotencia x y n =  
  enPosicion (show x) (show (2013^y)) n
```

```
-----  
-- Ejercicio 1.5. Calcular el menor n, tal que n aparece en la posición  
-- n de 2013^n.  
-----
```

```
menorFijo :: Integer  
menorFijo =  
  head [n | n <- [1..], enPosicionPotencia n n (fromIntegral n)]
```

```
-- El cálculo es  
--   ghci> menorFijo  
--   83
```

```
-----  
-- Ejercicio 1.6. Calcular el menor n (mayor que 1), tal que dentro  
-- 2013^n aparezca el 2013. ¿En qué posición aparece el 2013?  
-----
```

```
menorPotenciaConteniendo2013 :: Integer  
menorPotenciaConteniendo2013 =  
  head [n | n <- [2..], not (null (posicionesEnPotencia 2013 n))]
```

```
-- El cálculo es  
--   ghci> menorPotenciaConteniendo2013  
--   98  
--   ghci> posicionesEnPotencia 2013 98  
--   [23,170]
```

```

-----
-- Ejercicio 2. [2013 en factoriales] Con los apartados de este
-- ejercicio se resuelve el siguiente problema:
--   ¿Cuál es el primer factorial que tiene a "2013" dentro?
-----

```

```

-----
-- Ejercicio 2.1. Definir la función
--   fact :: Integer -> Integer
-- tal que (fact n) es el factorial de n. Por ejemplo,
--   fact 5 == 120
--   fact 18 == 6402373705728000
-----

```

```

fact :: Integer -> Integer
fact n = product [1..n]

```

```

-----
-- Ejercicio 2.2. Definir la función
--   enFactorial :: Integer -> Integer
-- tal que (enFactorial n) es el menor x tal que n ocurre dentro del
-- factorial de x. Por ejemplo,
--   enFactorial 20 == 5
--   enFactorial 23 == 18
-----

```

```

enFactorial :: Integer -> Integer
enFactorial n =
  head [x | x <- [0..], isInfixOf (show n) (show (fact x))]

```

```

-----
-- Ejercicio 2.3. Calcular el menor n tal que 2013 ocurre dentro del
-- factorial de n. ¿Qué posición ocupa?
-----

```

```

-- El cálculo es
--   ghci> enFactorial 2013
--   68
--   ghci> fact 68

```

```
-- 24800355424368305996009904185691715810473992013553676723717
-- 107380182214457121832960000000000000000
-- ghci> posiciones (show 2013) (show (fact 68))
-- [44]
```

```
-----
-- Ejercicio 3. [2013 en potencias de 2] Con los apartados de este
-- ejercicio se resuelve el siguiente problema
-- ¿Cuál es la primera potencia de 2 que tiene "2013" dentro? ¿y las
-- siguientes?
-----
```

```
-----
-- Ejercicio 3.1. Definir la función
-- enPotenciaDeDos :: Integer -> [Integer]
-- tal que (enPotenciaDeDos n) es la lista de los números x tales que n
-- aparece dentro de 2^x. Por ejemplo,
-- head (enPotenciaDeDos 21) == 18
-- 2^18 == 262144
-----
```

```
enPotenciaDeDos :: Integer -> [Integer]
enPotenciaDeDos n =
  [x | x <- [0..], isInfixOf (show n) (show (2^x))]
```

```
-----
-- Ejercicio 3.2. Calcular los 10 primeros números x tales que 2013
-- aparece dentro de 2^x.
-----
```

```
-- El cálculo es
-- ghci> take 10 (enPotenciaDeDos 2013)
-- [163,310,613,619,643,644,702,736,784,865]
```

```
-----
-- Ejercicio 4. [Criba de Eratóstenes] En los siguientes ejercicios se
-- usarán los números primos. Para ello usaremos el método de la criba
-- de Eratóstenes definida en los distintos apartados de este ejercicio.
-----
```

```

-----
-- Ejercicio 4.1. Definir la función
--   elimina :: Integer -> [Integer] -> [Integer]
-- tal que (elimina n xs) es la lista obtenida eliminando en la lista xs
-- los múltiplos de n. Por ejemplo,
--   elimina 3 [2,3,8,9,5,6,7] == [2,8,5,7]
-----

```

```

-- Por comprensión:
elimina :: Integer -> [Integer] -> [Integer]
elimina n xs = [ x | x <- xs, x `mod` n /= 0 ]

```

```

-- Por recursión:
eliminaR :: Integer -> [Integer] -> [Integer]
eliminaR n [] = []
eliminaR n (x:xs) | mod x n == 0 = eliminaR n xs
                  | otherwise    = x : eliminaR n xs

```

```

-----
-- Ejercicio 4.2. Definir la función
--   criba :: [Integer] -> [Integer]
-- tal que (criba xs) es la lista obtenida cribando la lista xs con el
-- método de eliminación del ejercicio anterior. Por ejemplo,
--   criba [2..20] == [2,3,5,7,11,13,17,19]
--   take 10 (criba [2..]) == [2,3,5,7,11,13,17,19,23,29]
-----

```

```

criba :: [Integer] -> [Integer]
criba [] = []
criba (n:ns) = n : criba (elimina n ns)

```

```

-----
-- Ejercicio 4.3. Definir la función
--   primos :: [Integer]
-- cuyo valor 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 = criba [2..]

```

```

-----
-- Ejercicio 4.4. Definir la función
--   esPrimo :: Integer -> Bool
-- tal que (esPrimo n) se verifica si n es primo. Por ejemplo,
--   esPrimo 7 == True
--   esPrimo 9 == False
-----

```

```

esPrimo :: Integer -> Bool
esPrimo n = head (dropWhile (<n) primos) == n

```

```

-----
-- Ejercicio 5. [Números de Haga] Un número de Haga es un número natural
-- n tal que la suma de n y sus divisores primos es igual que la de n+1
-- con los suyos. Por ejemplo, 14 es un número de Haga ya que  $14=2*7$ ,
--  $15=3*5$  y  $14+2+7 = 23 = 15+3+5$ .
--
-- En este ejercicio mostramos relaciones entre el 2013 y los números de
-- Haga.
-----

```

```

-----
-- Ejercicio 5.1. Definir la función
--   factorizacion :: Integer -> [Integer]
-- tal que (factorizacion n) es la descomposición de n en factores
-- primos. Por ejemplo,
--   factorizacion 300 == [2,2,3,5,5]
-----

```

```

factorizacion :: Integer -> [Integer]
factorizacion 1 = [1]
factorizacion n | n == x    = [n]
                 | otherwise = x : factorizacion (div n x)
                 where x = menorFactorPrimo n

```

```

-- (menorFactorPrimo n) es el menor factor primo de n. Por ejemplo,
--   menorFactorPrimo 125 == 5
--   menorFactorPrimo 127 == 127
menorFactorPrimo :: Integer -> Integer

```

```
menorFactorPrimo n = head [x | x <- primos, rem n x == 0]
```

```
-----
-- Ejercicio 5.2. Definir la función
--   nHaga :: Integer -> Bool
-- tal que (nHaga n) que se verifique si n es un número de Haga. Por
-- ejemplo,
--   nHaga 14 == True
--   nHaga 15 == False
-----
```

```
nHaga :: Integer -> Bool
```

```
nHaga n =
  n + sum (factorizacion n) == (n+1) + sum (factorizacion (n+1))
```

```
-----
-- Ejercicio 5.3. Comprobar que 2013 es un número de Haga.
-----
```

```
-- La comprobación es
--   ghci> nHaga 2013
--   True
```

```
-----
-- Ejercicio 5.4. Definir la función
--   sucHaga :: [Integer]
-- tal que sucHaga es la sucesión de los números de Haga. Por ejemplo,
--   take 5 sucHaga == [7,14,63,80,224]
-----
```

```
sucHaga :: [Integer]
```

```
sucHaga = [n | n <- [1..], nHaga n]
```

```
-----
-- Ejercicio 5.5. ¿Qué posición ocupa el 2013 en la sucesión de números
-- de Haga? ¿cuál es el anterior? ¿y el siguiente?
-----
```

```
-- El cálculo es
--   ghci> length (takeWhile (<=2013) sucHaga)
```

```

--      14
--      ghci> last (takeWhile (<2013) suchHaga)
--      1479
--      ghci> head (dropWhile (<=2013) suchHaga)
--      2023
--      Es decir, el 2013 es el 14º número de Haga, el anterior es 1479 y el
--      siguiente el 2023.

-----

--      Ejercicio 6. [Números esfénicos] Un número es esfénico si se
--      descompone como producto de 3 primos distintos. Por ejemplo,
--      30 es esfénico      porque  $30 = 2 \cdot 3 \cdot 5$ 
--      10 no es esfénico  porque  $10 = 2 \cdot 5$ 
--      18 no es esfénico  porque  $18 = 2 \cdot 3 \cdot 3$ 
--      En este ejercicio mostramos relaciones entre el 2013 y los números
--      esfénicos.

-----

--      Ejercicio 6.1. Definir la función
--      descomposiciones :: Integer -> [(Integer,Integer,Integer)]
--      tal que (descomposiciones n) es la lista de ternas (x,y,z) de números
--      primos crecientes tales que  $x \cdot y \cdot z$  es igual a n. Por ejemplo,
--      descomposiciones 30 == [(2,3,5)]
--      descomposiciones 10 == []
--      descomposiciones 18 == []

-----

descomposiciones :: Integer -> [(Integer,Integer,Integer)]
descomposiciones n =
  [(x,y,z) | x <- takeWhile (<n) primos,
             y <- takeWhile (<n) (dropWhile (<=x) primos),
             let z = n `div` (x*y),
                 y < z, x*y*z == n, esPrimo z]

-----

--      Ejercicio 6.2. Definir la función
--      esfénico :: Integer -> Bool
--      tal que (esfénico n) se verifica si n es un número esfénico. Por
--      ejemplo,

```

```
-- esfenico 30 == True
-- esfenico 10 == False
-- esfenico 18 == False
```

```
-----
```

```
esfenico :: Integer -> Bool
```

```
esfenico n = not (null (descomposiciones n))
```

```
-----
```

```
-- Ejercicio 6.3. Comprobar que 2013 es un número esfénico.
```

```
-----
```

```
-- La comprobación es
-- ghci> esfenico 2013
-- True
```

```
-----
```

```
-- Ejercicio 6.4. Definir la función
```

```
esfenicos :: [Integer]
-- tal que esfenicos es la sucesión de los números esfénicos. Por ejemplo,
-- take 5 esfenicos == [30,42,66,70,78]
```

```
-----
```

```
esfenicos :: [Integer]
```

```
esfenicos = [n | n <- [1..], esfenico n]
```

```
-----
```

```
-- Ejercicio 6.5. ¿Qué posición ocupa el 2013 en la sucesión de números
-- esfénicos? ¿cuál es el anterior? ¿y el siguiente?
```

```
-----
```

```
-- El cálculo es
-- ghci> length (takeWhile (<=2013) esfenicos)
-- 305
-- ghci> last (takeWhile (<2013) esfenicos)
-- 2006
-- ghci> head (dropWhile (<=2013) esfenicos)
-- 2014
```

```
-- Es decir, 2013 es el 305º número esfénico, el anterior es el 2006 y
-- el siguiente el 2014.
```



```
-----  
-- Ejercicio 7.1. [Números biesfénicos] Un número  $n$  es biesfénico si  
--  $n$  y  $n+1$  son esfénicos. Definir la función  
--   biesfenico :: Integer -> Bool  
-- tal que (biesfenico  $n$ ) se verifica si  $n$  es un número biesfénico. Por  
-- ejemplo,  
--   biesfenico 230 == True  
--   biesfenico 235 == False  
-----
```

```
biesfenico :: Integer -> Bool  
biesfenico n = esfénico n && esfénico (n+1)
```

```
-----  
-- Ejercicio 7.2. Comprobar que 2013 es un número biesfénico.  
-----
```

```
-- La comprobación es  
-- ghci> biesfenico 2013  
-- True
```

```
-----  
-- Ejercicio 7.3. Definir la función  
--   biesfenicos :: [Integer]  
-- tal que biesfenicos es la sucesión de los números biesfénicos. Por  
-- ejemplo,  
--   take 5 biesfenicos == [230,285,429,434,609]  
-----
```

```
biesfenicos :: [Integer]  
biesfenicos = [n | n <- esfénicos, esfénico (n+1)]
```

```
-----  
-- Ejercicio 7.4. ¿Qué posición ocupa el 2013 en la sucesión de números  
-- biesfénicos? ¿cuál es el anterior? ¿y el siguiente?  
-----
```

```
-- El cálculo es  
-- ghci> length (takeWhile (<=2013) biesfenicos)
```

```
--      42
--      ghci> last (takeWhile (<2013) biesfenicos)
--      1946
--      ghci> head (dropWhile (<=2013) biesfenicos)
--      2014
--      Es decir, 2013 es el 42º número biesfénico, el anterior es el 1946 y
--      el siguiente el 2014.
```

```
-----
--      Ejercicio 8.1. [Números triesfénicos] Un número  $n$  es triesfénico si
--       $n$ ,  $n+1$  y  $n+2$  son esfénicos. Definir la función
--      triesfenico :: Integer -> Bool
--      tal que (triesfenico n) se verifica si  $n$  es un número
--      triesfénico. Por ejemplo,
--      triesfenico 98 == True
--      triesfenico 99 == False
-----
```

```
triesfenico :: Integer -> Bool
triesfenico n = biesfenico n && esfénico (n+2)
```

```
-----
--      Ejercicio 8.2. Comprobar que 2013 es un número triesfénico.
-----
```

```
--      La comprobación es
--      ghci> triesfenico 2013
--      True
```

```
-----
--      Ejercicio 8.3. Definir la función
--      triesfenicos :: [Integer]
--      tal que triesfenicos es la sucesión de los números triesfénicos. Por
--      ejemplo,
--      take 2 triesfenicos == [1309,1885]
-----
```

```
triesfenicos :: [Integer]
triesfenicos =
    [n | n <- esfénicos, esfénico (n+1), esfénico (n+2)]
```

```

-----
-- Ejercicio 8.4. ¿Qué posición ocupa el 2013 en la sucesión de números
-- triesfénicos? ¿cuál es el anterior? ¿y el siguiente?
-----

-- El cálculo es
-- ghci> length (takeWhile (<=2013) triesfenicos)
-- 3
-- ghci> last (takeWhile (<2013) triesfenicos)
-- 1885
-- ghci> head (dropWhile (<=2013) triesfenicos)
-- 2665

-----

-- Ejercicio 9.1. [Sumas de esfénicos consecutivos] Definir la función
-- sumaDeEsfenicos :: Integer -> [(Integer,Integer,Integer)]
-- tal que (sumaDeEsfenicos n) es la lista de las ternas de números
-- esfénicos consecutivos cuya suma es n. Por ejemplo,
-- sumaDeEsfenicos 138 == [(30,42,66)]
-----

sumaDeEsfenicos :: Integer -> [(Integer,Integer,Integer)]
sumaDeEsfenicos n =
  [(x,y,z) | x <- takeWhile (<=n) esfenicos,
             let y = head (dropWhile (<=x) esfenicos),
                 z = head (dropWhile (<=y) esfenicos),
                 x+y+z == n]

-----

-- Ejercicio 9.2. Comprobar que 2013 se puede escribir como suma de
-- tres números esfénicos consecutivos
-----

-- La comprobación es
-- ghci> sumaDeEsfenicos 2013
-- [(665,670,678)]

-----

-- Ejercicio 10. [Números 2-Smith] Un número es 2-Smith si la suma de

```

```
-- dígitos de sus factores primos es 2 veces la suma de sus dígitos. Por
-- ejemplo, 32 es un número 2-Smith ya que su factorización es
--  $32=2*2*2*2*2$  y  $2+2+2+2+2 = 2*(3+2)$ ; 330 también lo es ya que su
-- factorización es  $330=2*3*5*11$  y  $2+3+5+(1+1)=2*(3+3+0)$ .
--
-- En este ejercicio mostramos relaciones entre el 2013 y los números
-- 2-Smith.
```

```
-----
-- Ejercicio 10.1. Definir la función
--   digitos :: Integer -> [Integer]
-- tal que (digitos n) es la lista de los dígitos de n. Por ejemplo,
--   digitos 32527 == [3,2,5,2,7]
```

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

```
-----
-- Ejercicio 10.2. Definir la función
--   sumaDigitos :: Integer -> Integer
-- tal que (sumaDigitos n) es la suma de los dígitos de n. Por ejemplo,
--   sumaDigitos 32527 == 19
```

```
sumaDigitos :: Integer -> Integer
sumaDigitos n = sum (digitos n)
```

```
-----
-- Ejercicio 10.3.. Definir la función
--   es2Smith :: Integer -> Bool
-- tal que (es2Smith n) se verifica si n es un número 2-Smith. Por
-- ejemplo,
--   es2Smith 32 == True
--   es2Smith 33 == False
```

```
es2Smith :: Integer -> Bool
es2Smith n =
```

```

sum [sumaDigitos x | x <- factorizacion n] == 2 * sumaDigitos n

-----
-- Ejercicio 10.4. Comprobar que 2013 es un número 2-Smith.
-----

-- La comprobación es
-- ghci> es2Smith 2013
-- True

-----
-- Ejercicio 10.5. Definir la función
--   suc2Smith :: [Integer]
-- tal que suc2Smith es la sucesión de los números 2-Smith. Por
-- ejemplo,
--   take 7 suc2Smith == [32,42,60,70,104,152,231]
-----

suc2Smith :: [Integer]
suc2Smith = [n | n <- [1..], es2Smith n]

-----
-- Ejercicio 10.6. ¿Qué posición ocupa el 2013 en la sucesión de números
-- 2-Smith? ¿cuál es el anterior? ¿y el siguiente?
-----

-- El cálculo es
-- ghci> length (takeWhile (<=2013) suc2Smith)
-- 41
-- ghci> last (takeWhile (<2013) suc2Smith)
-- 1603
-- ghci> head (dropWhile (<=2013) suc2Smith)
-- 2054
-- Es decir, 2013 es el 41º número 2-Smith, el anterior es el 1603 y
-- el siguiente el 2054.

-----
-- Ejercicio 10.7. ¿Cuántos números 2-Smith, hasta el 2013 inclusive, son
-- esfénicos?
-----

```

```

-- El cálculo es
-- ghci> length [n | n <- takeWhile (<=2013) suc2Smith, esfénico n]
-- 13

-----

-- Ejercicio 10.8. ¿Cuál es el primer número 2-Smith que es triesfénico?
-----

-- El cálculo es
-- ghci> head [n | n <- suc2Smith, triesfénico n]
-- 2013

-----

-- Ejercicio 11. [Números de Murthy] Un número de Murthy es un número
-- natural  $n$  tal que  $n$  es múltiplo de 3 construido con los dígitos 0, 1,
-- 2 y 3. Por ejemplo, 1020 es un número de Murthy
--
-- En este ejercicio mostramos relaciones entre el 2013 y los números
-- de Murthy.
-----

-----

-- Ejercicio 11.1. Definir la función
-- subconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
-- subconjunto [2,3,2,1] [1..4] == True
-- subconjunto [2,3,2,6] [1..4] == False
-----

subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]

-----

-- Ejercicio 11.2. Definir la función
-- nMurthy :: Integer -> Bool
-- tal que (nMurthy n) se verifica si  $n$  es un número de Murthy. Por
-- ejemplo,
-- nMurthy 1020 == True

```

```
-- nMurthy 1021 == False
-- nMurthy 1050 == False
```

```
nMurthy :: Integer -> Bool
```

```
nMurthy n = rem n 3 == 0 && subconjunto (digitos n) [0..3]
```

```
-- Ejercicio 11.3. Comprobar que 2013 es un número de Murthy
```

```
-- La comprobación es
-- ghci> nMurthy 2013
-- True
```

```
-- Ejercicio 11.4. Definir la función
```

```
-- sucMurthy :: [Integer]
```

```
-- tal que sucMurthy es la sucesión de los números de Murthy. Por
-- ejemplo,
```

```
-- take 8 sucMurthy == [0,3,12,21,30,33,102,111]
```

```
sucMurthy :: [Integer]
```

```
sucMurthy = [n | n <- [0..], nMurthy n]
```

```
-- Ejercicio 11.5. ¿Qué posición ocupa el 2013 en la sucesión de números
-- de Murthy? ¿cuál es el anterior? ¿y el siguiente?
```

```
-- El cálculo es
```

```
-- ghci> length (takeWhile (<=2013) sucMurthy)
```

```
-- 46
```

```
-- ghci> last (takeWhile (<2013) sucMurthy)
```

```
-- 2010
```

```
-- ghci> head (dropWhile (<=2013) sucMurthy)
```

```
-- 2022
```

```
-- Es decir, 2013 es el 46º número de Murthy, el anterior es el 2010 y
-- el siguiente el 2022.
```

```

-----
-- Ejercicio 12. [Cuadrados en base 6] En este ejercicio mostramos
-- relaciones entre el 2013 y los cuadrados escritos en base 6.
-----

-----
-- Ejercicio 12.1. Definir la función
--   listaNumero :: [Integer] -> Integer
-- tal que (listaNumero xs) es el número formado por los dígitos xs (en
-- orden inverso). Por ejemplo,
--   listaNumero [5]           == 5
--   listaNumero [7,4,3,1]    == 1347
--   listaNumero [1,0,0]      == 1
-----

listaNumero :: [Integer] -> Integer
listaNumero xs = sum [y*10^n | (y,n) <- zip xs [0..]]

-----
-- Ejercicio 12.2. Definir la función
--   enBase6 :: Integer -> Integer
-- tal que (enBase6 n) es el número n en base 6. Por ejemplo,
--   enBase6 5 == 5
--   enBase6 6 == 10
--   enBase6 8 == 12
-----

enBase6 :: Integer -> Integer
enBase6 n = listaNumero (aux n)
  where aux n | n < 6      = [n]
             | otherwise = n `rem` 6 : aux (n `div` 6)

-----
-- Ejercicio 12.3. Definir la función
--   cuadradosEnBase6 :: [Integer]
-- cuadradosEnBase6 es la lista de los cuadrados escritos en base 6. Por
-- ejemplo,
--   take 9 cuadradosEnBase6 == [0,1,4,13,24,41,100,121,144]
-----

```



```
cuadradosEnBase6 :: [Integer]
cuadradosEnBase6 = [enBase6 (n*n) | n <- [0..]]
```

```
-- -----
-- Ejercicio 12.4. Comprobar que 2013 es un cuadrado escrito en base 6.
-- -----
```

```
-- La comprobación es
-- ghci> elem 2013 cuadradosEnBase6
-- True
```

```
-- -----
-- Ejercicio 12.5. ¿Qué posición ocupa el 2013 en la sucesión de los
-- cuadrados escritos en base 6? ¿cuál es el anterior? ¿y el siguiente?
-- -----
```

```
-- El cálculo es
-- ghci> length (takeWhile (<=2013) cuadradosEnBase6)
-- 22
-- ghci> last (takeWhile (<2013) cuadradosEnBase6)
-- 1504
-- ghci> head (dropWhile (<=2013) cuadradosEnBase6)
-- 2124
-- Es decir, 2013 es el 22º cuadrado escrito en base 6, el anterior es
-- el 1504 y el siguiente el 2124.
```

```
-- -----
-- Ejercicio 13 [Fibonacci a partir de 1 y 22]. La sucesión de Fibonacci
-- a partir de 1 y 22 es
-- 1, 22, 23, 45, 68, 113, 181, ...
-- es decir, se empieza en 1 y 22 y se continúa sumando los dos
-- anteriores.
```

```
-- En este ejercicio mostramos relaciones entre el 2013 y la sucesión de
-- Fibonacci a partir de 1 y 22.
```

```
-- -----
-- Ejercicio 13.1. Definir la función
```

```
-- fibonacci_1_22 :: Integer -> Integer
-- tal que (fibonacci_1_22 n) es el n-esimo término de la sucesión de
-- Fibonacci a partir de 1 y 22. Por ejemplo,
-- fibonacci_1_22 5 == 68
```

```
-----
fibonacci_1_22 :: Integer -> Integer
fibonacci_1_22 1 = 1
fibonacci_1_22 2 = 22
fibonacci_1_22 n = fibonacci_1_22 (n-1) + fibonacci_1_22 (n-2)
```

```
-----
-- Ejercicio 13.2. Definir la función
-- sucFib_1_22 :: [Integer]
-- tal que (sucFib_1_22 n) es la sucesión de Fibonacci a partir de 1 y
-- 22. Por ejemplo,
-- take 9 sucFib_1_22 == [1,22,23,45,68,113,181,294,475]
```

```
-----
sucFib_1_22 :: [Integer]
sucFib_1_22 = [fibonacci_1_22 n | n <- [1..]]
```

```
-- Otra definición (más eficiente) es
sucFib_1_22' :: [Integer]
sucFib_1_22' =
  1:22:[x+y | (x,y) <- zip sucFib_1_22' (tail sucFib_1_22')]
```

```
-----
-- Ejercicio 13.3. Comprobar que 2013 es un término de la sucesión de
-- Fibonacci a partir de 1 y 22
```

```
-----
-- La comprobación es
-- ghci> elem 2013 sucFib_1_22
-- True
```

```
-----
-- Ejercicio 13.4. ¿Qué posición ocupa el 2013 en la sucesión de los
-- términos de la sucesión de Fibonacci que comienza con 1 y 22? ¿cuál
-- es el anterior? ¿y el siguiente?
```

```

-----
-- El cálculo es
-- ghci> length (takeWhile (<=2013) sucFib_1_22)
-- 12
-- ghci> last (takeWhile (<2013) sucFib_1_22)
-- 1244
-- ghci> head (dropWhile (<=2013) sucFib_1_22)
-- 3257
-- Es decir, 2013 es el 12º término de la sucesión de Fibonacci que
-- comienza con 1 y 22, el anterior es el 1244 y el siguiente el 3257.

```

```

-----
-- Ejercicio 14. Definir la función
--   sumasDe3Primos :: Integer -> [(Integer,Integer,Integer)]
-- tal que (sumasDe3Primos n) es la lista de ternas de primos (x,y,z)
-- tales que x+y+z es igual a n. Por ejemplo,
--   sumasDe3Primos 11 == [(2,2,7),(3,3,5)]
--   sumasDe3Primos 12 == [(2,3,7),(2,5,5)]
-- ¿De cuántas maneras se puede expresar 2013 como suma de 3 primos?

```

```

sumasDe3Primos :: Integer -> [(Integer,Integer,Integer)]
sumasDe3Primos n =
  [(x,y,z) | x <- takeWhile (<n) primos,
             y <- takeWhile (<n) (dropWhile (<x) primos),
             let z = n-x-y,
                 y <= z, esPrimo z]

```

```

-- El cálculo es
-- ghci> length (sumasDe3Primos 2013)
-- 2253

```

```

-----
-- Ejercicio 15. [Dígitos distintos] Una propiedad del 2013 es que sus
-- dígitos son distintos.
--
-- En este ejercicio mostramos relaciones entre el 2013 y los años con
-- dígitos distintos.
-----

```

```
-----  
-- Ejercicio 15.1. Definir la función  
--   distintos :: Eq a => [a] -> Bool  
-- tal que (distintos xs) se verifica si todos los elementos de xs son  
-- distintos. Por ejemplo,  
--   distintos [3,2,5,7] == True  
--   distintos [3,2,5,2] == False  
-----
```

```
distintos :: Eq a => [a] -> Bool  
distintos [] = True  
distintos (x:xs) = notElem x xs && distintos xs
```

```
-----  
-- Ejercicio 15.2. Definir la función  
--   digitosDistintos :: Integer -> Bool  
-- tal que (digitosDistintos n) se verifica si todos los dígitos de n  
-- son distintos. Por ejemplo,  
--   digitosDistintos 3257 == True  
--   digitosDistintos 3252 == False  
-----
```

```
digitosDistintos :: Integer -> Bool  
digitosDistintos n = distintos (show n)
```

```
-----  
-- Ejercicio 15.3. Definir la función  
--   sucDigitosDistintos :: [Integer]  
-- tal que sucDigitosDistintos es la sucesión de los números con todos  
-- sus dígitos distintos. Por ejemplo,  
--   ghci> take 10 [n | n <- sucDigitosDistintos, n > 119]  
--   [120,123,124,125,126,127,128,129,130,132]  
-----
```

```
sucDigitosDistintos :: [Integer]  
sucDigitosDistintos =  
  [n | n <- [1..], digitosDistintos n]  
-----
```

```
-- Ejercicio 15.4. ¿Qué posición ocupa el 2013 en la sucesión de los
-- números con todos sus dígitos distintos? ¿cuál es el anterior? ¿y el
-- siguiente?
-- -----

-- El cálculo es
-- ghci> length (takeWhile (<=2013) sucDigitosDistintos)
-- 1243
-- ghci> last (takeWhile (<2013) sucDigitosDistintos)
-- 1987
-- ghci> head (dropWhile (<=2013) sucDigitosDistintos)
-- 2014
-- Es decir, 2013 es el 1243º término de la sucesión de números con
-- todos sus dígitos distintos, el anterior es el 1987 y el siguiente el
-- 2014.
-- -----

-- Ejercicio 16. [Dígitos distintos] Una propiedad del 2013 es que sus
-- dígitos son distintos y consecutivos (es decir la lista ordenada de
-- sus dígitos es una lista de números consecutivos).
--
-- En este ejercicio mostramos relaciones entre el 2013 y los años con
-- dígitos distintos y consecutivos
-- -----

-- Ejercicio 16.1. Definir la función
-- distintosConsecutivos :: [Integer] -> Bool
-- tal que (distintosConsecutivos xs) se verifica si todos los elementos
-- de xs son distintos y consecutivos. Por ejemplo,
-- distintosConsecutivos [3,2,5,4] == True
-- distintosConsecutivos [3,2,5,7] == False
-- distintosConsecutivos [3,2,5,2] == False
-- -----

distintosConsecutivos :: [Integer] -> Bool
distintosConsecutivos xs =
  distintos xs && genericLength xs == 1 + maximum xs - minimum xs
-- -----
```

```
-- Ejercicio 16.2. Definir la función
--   digitosDistintosConsecutivos :: Integer -> Bool
-- tal que (digitosDistintosConsecutivos n) se verifica si todos los
-- dígitos de n son distintos y consecutivos. Por ejemplo,
--   digitosDistintosConsecutivos 3254 == True
--   digitosDistintosConsecutivos 3257 == False
--   digitosDistintosConsecutivos 3252 == False
```

```
-----
digitosDistintosConsecutivos :: Integer -> Bool
digitosDistintosConsecutivos n =
  distintosConsecutivos (digitos n)
```

```
-----
-- Ejercicio 16.3. Definir la función
--   sucDigitosDistintosConsecutivos :: [Integer]
-- tal que sucDigitosDistintosConsecutivos es la sucesión de los números
-- con todos sus dígitos distintos y consecutivos. Por ejemplo,
--   ghci> take 10 [n | n <- sucDigitosDistintosConsecutivos, n > 119]
--   [120,123,132,201,210,213,231,234,243,312]
```

```
-----
sucDigitosDistintosConsecutivos :: [Integer]
sucDigitosDistintosConsecutivos =
  [n | n <- [1..], digitosDistintosConsecutivos n]
```

```
-----
-- Ejercicio 16.4. ¿Qué posición ocupa el 2013 en la sucesión de los
-- números con todos sus dígitos distintos y consecutivos? ¿cuál es el
-- anterior? ¿y el siguiente?
```

```
-----
-- El cálculo es
--   ghci> length (takeWhile (<=2013) sucDigitosDistintosConsecutivos)
--   85
--   ghci> last (takeWhile (<2013) sucDigitosDistintosConsecutivos)
--   1432
--   ghci> head (dropWhile (<=2013) sucDigitosDistintosConsecutivos)
--   2031
-- Es decir, 2013 es el 85º término de la sucesión de números con
```

```
-- todos sus dígitos distintos y consecutivos, el anterior es el 1432 y
-- el siguiente el 2031.
```

```
-----
-- Ejercicio 17. [Caracterizaciones de 2013] En este ejercicio se
-- presentan caracterizaciones del 2013 basadas en los ejercicios
-- anteriores.
-----
```

```
-----
-- Ejercicio 17.1. Definir la función
-- pertenece :: Ord a => a -> [a] -> Bool
-- tal que (pertenece x ys) se verifica si x pertenece a ys. Por
-- ejemplo,
-- pertenece 10 [0,2..] == True
-- pertenece 11 [0,2..] == False
-----
```

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

```
-----
-- Ejercicio 17.2. Definir la función
-- interseccion :: Ord a => [a] -> [a] -> [a]
-- tal que (interseccion xs ys) es la intersección de xs e ys. Por
-- ejemplo,
-- interseccion [2..10] [0,2..] == [2,4,6,8,10]
-----
```

```
interseccion :: Ord a => [a] -> [a] -> [a]
interseccion xs ys = [x | x <- xs, pertenece x ys]
```

```
-----
-- Ejercicio 17.3. ¿Cuál es el primer número de Haga que es
-- triesfénico?
-----
```

```
-- El cálculo es
-- ghci> let triesfenicos' = takeWhile (<=2013) [n | n <- triesfenicos]
-- ghci> head (interseccion sucHaga triesfenicos')
```

```
-- 2013
```

```
-----  
-- Ejercicio 17.4. ¿Cuál es el primer número de Haga que es un número  
-- 2-Smith?  
-----
```

```
-- El cálculo es  
-- ghci> head (interseccion sucHaga suc2Smith)  
-- 2013
```

```
-----  
-- Ejercicio 17.5. ¿Cuál es el primer número de Haga que es un número  
-- de Murthy?  
-----
```

```
-- El cálculo es  
-- ghci> head (interseccion sucHaga sucMurthy)  
-- 2013
```

```
-----  
-- Ejercicio 17.6. ¿Cuál es el primer número de Haga que es un  
-- cuadrado escrito en base 6?  
-----
```

```
-- El cálculo es  
-- ghci> head (interseccion sucHaga sucMurthy)  
-- 2013
```

```
-----  
-- Ejercicio 17.7. ¿Cuál es el primer número triesfénico que es un  
-- número 2-Smith?  
-----
```

```
-- El cálculo es  
-- ghci> head (interseccion triesfenicos' suc2Smith)  
-- 2013
```

```
-----  
-- Ejercicio 17.8. ¿Cuál es el primer número triesfénico que es un
```



```
-- número de Murthy?
-- -----

-- El cálculo es
--   ghci> head (interseccion triesfenicos' sucMurthy)
--   2013

-- -----

-- Ejercicio 17.9. ¿Cuál es el primer número triesfénico que es un
-- cuadrado escrito en base 6?
-- -----

-- El cálculo es
--   ghci> head (interseccion triesfenicos' cuadradosEnBase6)
--   2013

-- -----

-- Ejercicio 17.10. ¿Cuál es el primer número triesfénico cuyos dígitos
-- son distintos y consecutivos?
-- -----

-- El cálculo es
--   ghci> head (interseccion triesfenicos' sucDigitosDistintosConsecutivos)
--   2013

-- -----

-- Ejercicio 17.11. ¿Cuál es el primer número 2-Smith cuyos dígitos son
-- distintos y consecutivos?
-- -----

-- El cálculo es
--   ghci> head (interseccion triesfenicos' sucDigitosDistintosConsecutivos)
--   2013

-- -----

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

-- * Potencias de 2013.
--   En http://goo.gl/Ccdni y http://goo.gl/5baU6
```

```
-- * 2013 en factoriales.  
--   En http://goo.gl/cAepr  
-- * 2013 en potencias de 2.  
--   En http://goo.gl/xevTq  
-- * Números de Haga.  
--   En http://oeis.org/A020700 y http://goo.gl/5B9zG  
-- * Números esfénicos.  
--   En http://goo.gl/cSIYD http://oeis.org/A007304 y http://goo.gl/jw3ev  
-- * Números biesfénicos.  
--   En http://oeis.org/A215217  
-- * Números triesfénicos.  
--   En http://oeis.org/A066509 y http://goo.gl/jw3ev  
-- * Números 2-Smith.  
--   En http://oeis.org/A104390 y http://goo.gl/jw3ev  
-- * Números de Murthy.  
--   En http://oeis.org/A061819  
-- * Cuadrados en base 6.  
--   En http://oeis.org/A001741  
-- * Sucesión de Fibonacci a partir de 1 y 22.  
--   En http://oeis.org/A022392
```

# Relación 14

## Funciones de orden superior y definiciones por plegados

```
-- -----  
-- Introducción --  
-- -----
```

```
-- Esta relación tiene contiene ejercicios con funciones de orden  
-- superior y definiciones por plegado correspondientes al tema 7 cuyas  
-- transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-7.pdf
```

```
-- -----  
-- Importación de librerías auxiliares --  
-- -----
```

```
import Test.QuickCheck
```

```
-- -----  
-- Ejercicio 1. Redefinir por recursión la función  
-- takeWhile :: (a -> Bool) -> [a] -> [a]  
-- tal que (takeWhile p xs) es la lista de los elemento de xs hasta el  
-- primero que no cumple la propiedad p. Por ejemplo,  
-- takeWhile' (<7) [2,3,9,4,5] == [2,3]  
-- -----
```

```
takeWhile' :: (a -> Bool) -> [a] -> [a]  
takeWhile' _ [] = []
```

```

takeWhile' p (x:xs)
  | p x      = x : takeWhile' p xs
  | otherwise = []

-----

-- Ejercicio 2. Redefinir por recursión la función
--   dropWhile :: (a -> Bool) -> [a] -> [a]
-- tal que (dropWhile p xs) es la lista de eliminando los elemento de xs
-- hasta el primero que cumple la propiedad p. Por ejemplo,
--   dropWhile' (<7) [2,3,9,4,5] == [9,4,5]
-----

dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' p (x:xs)
  | p x      = dropWhile' p xs
  | otherwise = x:xs

-----

-- Ejercicio 3. Redefinir, usando foldr, la función concat. Por ejemplo,
--   concat' [[1,3],[2,4,6],[1,9]] == [1,3,2,4,6,1,9]
-----

-- La definición por recursión es
concatR :: [[a]] -> [a]
concatR [] = []
concatR (xs:xss) = xs ++ concatR xss

-- La definición por plegado es
concat' :: [[a]] -> [a]
concat' = foldr (++) []

-----

-- Ejercicio 4.1. La función
--   divideMedia :: [Double] -> ([Double],[Double])
-- dada una lista numérica, xs, calcula el par (ys,zs), donde ys
-- contiene los elementos de xs estrictamente menores que la media,
-- mientras que zs contiene los elementos de xs estrictamente mayores
-- que la media. Por ejemplo,
--   divideMedia [6,7,2,8,6,3,4] == ([2.0,3.0,4.0],[6.0,7.0,8.0,6.0])

```

```

--      divideMedia [1,2,3]          == ([1.0],[3.0])
-- Definir la función divideMedia por filtrado, comprensión y
-- recursión.
-----

-- La definición por filtrado es
divideMediaF :: [Double] -> ([Double],[Double])
divideMediaF xs = (filter (<m) xs, filter (>m) xs)
  where m = media xs

-- (media xs) es la media de xs. Por ejemplo,
--      media [1,2,3]          == 2.0
--      media [1,-2,3.5,4]    == 1.625
-- Nota: En la definición de media se usa la función fromIntegral tal
-- que (fromIntegral x) es el número real correspondiente al número
-- entero x.
media :: [Double] -> Double
media xs = (sum xs) / fromIntegral (length xs)

-- La definición por comprensión es
divideMediaC :: [Double] -> ([Double],[Double])
divideMediaC xs = ([x | x <- xs, x < m], [x | x <- xs, x > m])
  where m = media xs

-- La definición por recursión es
divideMediaR :: [Double] -> ([Double],[Double])
divideMediaR xs = divideMediaR' xs
  where m = media xs
        divideMediaR' [] = ([],[ ])
        divideMediaR' (x:xs) | x < m = (x:ys, zs)
                              | x == m = (ys, zs)
                              | x > m = (ys, x:zs)
                              where (ys, zs) = divideMediaR' xs

-----
-- Ejercicio 4.2. Comprobar con QuickCheck que las tres definiciones
-- anteriores divideMediaF, divideMediaC y divideMediaR son
-- equivalentes.
-----

```

```

-- La propiedad es
prop_divideMedia :: [Double] -> Bool
prop_divideMedia xs =
  divideMediaC xs == d &&
  divideMediaR xs == d
  where d = divideMediaF xs

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

-----

-- Ejercicio 4.3. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces la suma
-- de las longitudes de ys y zs es menor o igual que la longitud de xs.
-----

-- La propiedad es
prop_longitudDivideMedia :: [Double] -> Bool
prop_longitudDivideMedia xs =
  length ys + length zs <= length xs
  where (ys,zs) = divideMediaF xs

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

-----

-- Ejercicio 4.4. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces todos los
-- elementos de ys son menores que todos los elementos de zs.
-----

-- La propiedad es
prop_divideMediaMenores :: [Double] -> Bool
prop_divideMediaMenores xs =
  and [y < z | y <- ys, z <- zs]
  where (ys,zs) = divideMediaF xs

-- La comprobación es

```

```

-- ghci> quickCheck prop_divideMediaMenores
-- +++ OK, passed 100 tests.

-----

-- Ejercicio 4.5. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces la
-- media de xs no pertenece a ys ni a zs.
-- Nota: Usar la función notElem tal que (notElem x ys) se verifica si y
-- no pertenece a ys.
-----

-- La propiedad es
prop_divideMediaSinMedia :: [Double] -> Bool
prop_divideMediaSinMedia xs =
  notElem m (ys ++ zs)
  where m      = media xs
        (ys,zs) = divideMediaF xs

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

-----

-- Ejercicio 5. Definir la función
-- segmentos :: (a -> Bool) -> [a] -> [a]
-- tal que (segmentos p xs) es la lista de los segmentos de xs cuyos
-- elementos verifican la propiedad p. Por ejemplo,
-- segmentos even [1,2,0,4,5,6,48,7,2] == [[],[2,0,4],[6,48],[2]]
-----

segmentos :: (a -> Bool) -> [a] -> [[a]]
segmentos _ [] = []
segmentos p xs =
  takeWhile p xs : (segmentos p (dropWhile (not.p) (dropWhile p xs)))

-----

-- Ejercicio 6. Definir la función
-- relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,

```

```
-- relacionados (<) [2,3,7,9]           == True
-- relacionados (<) [2,3,1,9]         == False
```

```
relacionados :: (a -> a -> Bool) -> [a] -> Bool
relacionados r (x:y:zs) = (r x y) && relacionados r (y:zs)
relacionados _ _ = True
```

-- Una definición alternativa es

```
relacionados' :: (a -> a -> Bool) -> [a] -> Bool
relacionados' r xs = and [r x y | (x,y) <- zip xs (tail xs)]
```

-- Ejercicio 7. Definir la función

```
agrupa :: Eq a => [[a]] -> [[a]]
-- tal que (agrupa xss) es la lista de las listas obtenidas agrupando
-- los primeros elementos, los segundos, ... Por ejemplo,
-- agrupa [[1..6],[7..9],[10..20]] == [[1,7,10],[2,8,11],[3,9,12]]
-- agrupa []                       == []
```

```
agrupa :: Eq a => [[a]] -> [[a]]
agrupa [] = []
agrupa xss
  | [] 'elem' xss = []
  | otherwise    = primeros xss : agrupa (restos xss)
  where primeros = map head
        restos   = map tail
```

-- Ejercicio 8.1. Definir por recursión la función

```
superpar :: Int -> Bool
-- tal que (superpar n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
-- superpar 426 == True
-- superpar 456 == False
```

```
superpar :: Int -> Bool
superpar n | n < 10 = even n
```



```

    | otherwise = even n && superpar (n `div` 10)

-- Otra forma equivalente es
superpar' :: Int -> Bool
superpar' 0 = True
superpar' n = even n && superpar' (div n 10)

-----

-- Ejercicio 8.2. Definir por comprensión la función
-- superpar2 :: Int -> Bool
-- tal que (superpar2 n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
-- superpar2 426 == True
-- superpar2 456 == False
-----

superpar2 :: Int -> Bool
superpar2 n = and [even d | d <- digitos n]

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

-----

-- Ejercicio 8.3. Definir, por recursión sobre los dígitos, la función
-- superpar3 :: Int -> Bool
-- tal que (superpar3 n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
-- superpar3 426 == True
-- superpar3 456 == False
-----

superpar3 :: Int -> Bool
superpar3 n = sonPares (digitos n)
  where sonPares []      = True
        sonPares (d:ds) = even d && sonPares ds

-----

-- Ejercicio 8.3. Definir, usando all, la función
-- superpar4 :: Int -> Bool
-- tal que (superpar4 n) se verifica si n es un número par tal que todos

```

```
-- sus dígitos son pares. Por ejemplo,
--   superpar4 426 == True
--   superpar4 456 == False
```

```
-----
superpar4 :: Int -> Bool
superpar4 n = all even (digitos n)
```

```
-----
-- Ejercicio 8.5. Definir, usando filter, la función
--   superpar5 :: Int -> Bool
-- tal que (superpar5 n) se verifica si n es un número par tal que todos
-- sus dígitos son pares. Por ejemplo,
--   superpar5 426 == True
--   superpar5 456 == False
```

```
-----
superpar5 :: Int -> Bool
superpar5 n = filter even (digitos n) == digitos n
```

```
-----
-- Ejercicio 9. Se considera la función
--   filtraAplica :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplica f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--   filtraAplica (4+) (<3) [1..7] => [5,6]
-- Se pide, definir la función
-- 1. por comprensión,
-- 2. usando map y filter,
-- 3. por recursión y
-- 4. por plegado (con foldr).
```

```
-----
-- La definición con lista de comprensión es
filtraAplica_1 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_1 f p xs = [f x | x <- xs, p x]
```

```
-----
-- La definición con map y filter es
filtraAplica_2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_2 f p xs = map f (filter p xs)
```

```
-- La definición por recursión es
filtraAplica_3 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_3 f p [] = []
filtraAplica_3 f p (x:xs) | p x      = f x : filtraAplica_3 f p xs
                          | otherwise = filtraAplica_3 f p xs
```

```
-- La definición por plegado es
filtraAplica_4 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_4 f p = foldr g []
                    where g x y | p x      = f x : y
                              | otherwise = y
```

```
-- La definición por plegado usando lambda es
filtraAplica_4' :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_4' f p =
  foldr (\x y -> if p x then (f x : y) else y) []
```

```
-----
-- Ejercicio 10.1. Definir, mediante recursión, la función
--   maximumR :: Ord a => [a] -> a
-- tal que (maximumR xs) es el máximo de la lista xs. Por ejemplo,
--   maximumR [3,7,2,5]           == 7
--   maximumR ["todo","es","falso"] == "todo"
--   maximumR ["menos","alguna","cosa"] == "menos"
-- Nota: La función maximumR es equivalente a la predefinida maximum.
-----
```

```
maximumR :: Ord a => [a] -> a
maximumR [x]      = x
maximumR (x:y:ys) = max x (maximumR (y:ys))
```

```
-----
-- Ejercicio 10.2. La función de plegado foldr1 está definida por
--   foldr1 :: (a -> a -> a) -> [a] -> a
--   foldr1 _ [x]      = x
--   foldr1 f (x:xs) = f x (foldr1 f xs)
--
-- Definir, mediante plegado con foldr1, la función
--   maximumP :: Ord a => [a] -> a
```

```
-- tal que (maximumR xs) es el máximo de la lista xs. Por ejemplo,  
--   maximumP [3,7,2,5]           == 7  
--   maximumP ["todo","es","falso"] == "todo"  
--   maximumP ["menos","alguna","cosa"] == "menos"  
-- Nota: La función maximumP es equivalente a la predefinida maximum.  
-----
```

```
maximumP :: Ord a => [a] -> a  
maximumP = foldr1 max
```

```
-----  
-- Ejercicio 11. Definir, mediante plegado con foldr1, la función  
--   minimunP :: Ord a => [a] -> a  
-- tal que (minimunR xs) es el máximo de la lista xs. Por ejemplo,  
--   minimunP [3,7,2,5]           == 2  
--   minimunP ["todo","es","falso"] == "es"  
--   minimunP ["menos","alguna","cosa"] == "alguna"  
-- Nota: La función minimunP es equivalente a la predefinida minimun.  
-----
```

```
minimunP :: Ord a => [a] -> a  
minimunP = foldr1 min
```

# Relación 15

## Funciones de orden superior y definiciones por plegados (2)

```
-- -----  
-- Introducción --  
-- -----  
  
-- Esta relación tiene contiene ejercicios con funciones de orden  
-- superior y definiciones por plegado correspondientes al tema 7 cuyas  
-- transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-7.pdf  
-- -----  
-- Importación de librerías auxiliares --  
-- -----  
  
import Data.List  
import Test.QuickCheck  
  
-- -----  
-- Ejercicio 1.1. Definir, mediante recursión, la función  
-- inversaR :: [a] -> [a]  
-- tal que (inversaR xs) es la inversa de la lista xs. Por ejemplo,  
-- inversaR [3,5,2,4,7] == [7,4,2,5,3]  
-- -----  
  
inversaR :: [a] -> [a]  
inversaR [] = []
```

```
inversaR (x:xs) = (inversaR xs) ++ [x]
```

```
-----
-- Ejercicio 1.2. Definir, mediante plegado, la función
--   inversaP :: [a] -> [a]
-- tal que (inversaP xs) es la inversa de la lista xs. Por ejemplo,
--   inversaP [3,5,2,4,7] == [7,4,2,5,3]
-----
```

```
inversaP :: [a] -> [a]
inversaP = foldr f []
  where f x y = y ++ [x]
```

```
-- La definición anterior puede simplificarse a
```

```
inversaP2 :: [a] -> [a]
inversaP2 = foldr f []
  where f x = (++ [x])
```

```
-----
-- Ejercicio 1.3. Definir, por recursión con acumulador, la función
--   inversaR' :: [a] -> [a]
-- tal que (inversaR' xs) es la inversa de la lista xs. Por ejemplo,
--   inversaR' [3,5,2,4,7] == [7,4,2,5,3]
-----
```

```
inversaR' :: [a] -> [a]
inversaR' xs = inversaAux [] xs
  where inversaAux ys []      = ys
        inversaAux ys (x:xs) = inversaAux (x:ys) xs
```

```
-----
-- Ejercicio 1.4. La función de plegado foldl está definida por
--   foldl :: (a -> b -> a) -> a -> [b] -> a
--   foldl f ys xs = aux ys xs
--   where aux ys []      = ys
--         aux ys (x:xs) = aux (f ys x) xs
-- Definir, mediante plegado con foldl, la función
--   inversaP' :: [a] -> [a]
-- tal que (inversaP' xs) es la inversa de la lista xs. Por ejemplo,
--   inversaP' [3,5,2,4,7] == [7,4,2,5,3]
```

```
-----  
inversaP' :: [a] -> [a]  
inversaP' = foldl f []  
  where f ys x = x:ys
```

```
-- La definición anterior puede simplificarse lambda:
```

```
inversaP'2 :: [a] -> [a]  
inversaP'2 = foldl (\ys x -> x:ys) []
```

```
-- La definición puede simplificarse usando flip:
```

```
inversaP'3 :: [a] -> [a]  
inversaP'3 = foldl (flip(:)) []
```

```
-----  
-- Ejercicio 1.5. Comprobar con QuickCheck que las funciones reverse,  
-- inversaP e inversaP' son equivalentes.  
-----
```

```
-- La propiedad es
```

```
prop_inversa :: Eq a => [a] -> Bool  
prop_inversa xs =  
  inversaP xs == ys &&  
  inversaP' xs == ys  
  where ys = reverse xs
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_inversa  
-- +++ OK, passed 100 tests.
```

```
-----  
-- Ejercicio 1.6. Comparar la eficiencia de inversaP e inversaP'  
-- calculando el tiempo y el espacio que usado en evaluar las siguientes  
-- expresiones:
```

```
-- head (inversaP [1..100000])  
-- head (inversaP' [1..100000])  
-----
```

```
-- La sesión es
```

```
-- ghci> :set +s
```

```
-- ghci> head (inversaP [1..100000])
-- 100000
-- (0.41 secs, 20882460 bytes)
-- ghci> head (inversaP' [1..100000])
-- 1
-- (0.00 secs, 525148 bytes)
-- ghci> :unset +s
```

```
-----
-- Ejercicio 2.1. Definir, por recursión con acumulador, la función
--   dec2entR :: [Int] -> Int
-- tal que (dec2entR xs) es el entero correspondiente a la expresión
-- decimal xs. Por ejemplo,
--   dec2entR [2,3,4,5] == 2345
-----
```

```
dec2entR :: [Int] -> Int
dec2entR xs = dec2entR' 0 xs
  where dec2entR' a []      = a
        dec2entR' a (x:xs) = dec2entR' (10*a+x) xs
```

```
-----
-- Ejercicio 2.2. Definir, por plegado con foldl, la función
--   dec2entP :: [Int] -> Int
-- tal que (dec2entP xs) es el entero correspondiente a la expresión
-- decimal xs. Por ejemplo,
--   dec2entP [2,3,4,5] == 2345
-----
```

```
dec2entP :: [Int] -> Int
dec2entP = foldl f 0
  where f a x = 10*a+x
```

```
-- La definición puede simplificarse usando lambda:
```

```
dec2entP' :: [Int] -> Int
dec2entP' = foldl (\a x -> 10*a+x) 0
```

```
-----
-- Ejercicio 3.1. Definir por recursión la función
--   sumaR :: Num b => (a -> b) -> [a] -> b
```



```
-- tal que (suma f xs) es la suma de los valores obtenido aplicando la
-- función f a lo elementos de la lista xs. Por ejemplo,
-- sumaR (*2) [3,5,10] == 36
-- sumaR (/10) [3,5,10] == 1.8
```

```
-----
sumaR :: Num b => (a -> b) -> [a] -> b
sumaR f [] = 0
sumaR f (x:xs) = f x + sumaR f xs
```

```
-----
-- Ejercicio 3.2. Definir por plegado la función
-- sumaP :: Num b => (a -> b) -> [a] -> b
-- tal que (suma f xs) es la suma de los valores obtenido aplicando la
-- función f a lo elementos de la lista xs. Por ejemplo,
-- sumaP (*2) [3,5,10] == 36
-- sumaP (/10) [3,5,10] == 1.8
```

```
-----
sumaP :: Num b => (a -> b) -> [a] -> b
sumaP f = foldr (\x y -> (f x) + y) 0
```

```
-----
-- Ejercicio 4.1. Redefinir, por recursión, la función map. Por ejemplo,
-- mapR (+2) [1,7,3] == [3,9,5]
```

```
-----
mapR :: (a -> b) -> [a] -> [b]
mapR f [] = []
mapR f (x:xs) = f x : mapR f xs
```

```
-----
-- Ejercicio 4.2. Redefinir, usando foldr, la función map. Por ejemplo,
-- mapP (+2) [1,7,3] == [3,9,5]
```

```
-----
mapP :: (a -> b) -> [a] -> [b]
mapP f = foldr g []
  where g x xs = f x : xs
```

-- La definición por plegado usando lambda es

```
mapP1 :: (a -> b) -> [a] -> [b]
mapP1 f = foldr (\x y -> f x:y) []
```

-- Otra definición es

```
mapP2 :: (a -> b) -> [a] -> [b]
mapP2 f = foldr (:) . f []
```

-----  
 -- Ejercicio 5.1. Redefinir, usando foldr, la función filter. Por  
 -- ejemplo,  
 -- filterR (<4) [1,7,3,2] => [1,3,2]  
 -----

```
filterR :: (a -> Bool) -> [a] -> [a]
filterR p [] = []
filterR p (x:xs) | p x      = x : filterR p xs
                  | otherwise = filterR p xs
```

-----  
 -- Ejercicio 5.2. Redefinir, usando foldr, la función filter. Por  
 -- ejemplo,  
 -- filterP (<4) [1,7,3,2] => [1,3,2]  
 -----

```
filterP :: (a -> Bool) -> [a] -> [a]
filterP p = foldr g []
            where g x y | p x      = x:y
                       | otherwise = y
```

-- La definición por plegado y lambda es

```
filterP1 :: (a -> Bool) -> [a] -> [a]
filterP1 p = foldr (\x y -> if (p x) then (x:y) else y) []
```

-----  
 -- Ejercicio 6.1. Definir, mediante recursión, la función  
 -- sumllR :: Num a => [[a]] -> a  
 -- tal que (sumllR xss) es la suma de las sumas de las listas de xss.  
 -- Por ejemplo,  
 -- sumllR [[1,3],[2,5]] == 11  
 -----

```
-----
sumllR :: Num a => [[a]] -> a
sumllR [] = 0
sumllR (xs:xss) = sum xs + sumllR xss
```

```
-----
-- Ejercicio 6.2. Definir, mediante plegado, la función
--   sumllP :: Num a => [[a]] -> a
-- tal que (sumllP xss) es la suma de las sumas de las listas de xss. Por
-- ejemplo,
--   sumllP [[1,3],[2,5]] == 11
-----
```

```
sumllP :: Num a => [[a]] -> a
sumllP = foldr f 0
  where f xs n = sum xs + n
```

```
-- La definición anterior puede simplificarse usando lambda
sumllP' :: Num a => [[a]] -> a
sumllP' = foldr (\xs n -> sum xs + n) 0
```

```
-----
-- Ejercicio 6.3. Definir, mediante recursión con acumulador, la función
--   sumllA :: Num a => [[a]] -> a
-- tal que (sumllA xss) es la suma de las sumas de las listas de xss. Por
-- ejemplo,
--   sumllA [[1,3],[2,5]] == 11
-----
```

```
sumllA :: Num a => [[a]] -> a
sumllA xs = aux 0 xs
  where aux a [] = a
        aux a (xs:xss) = aux (a + sum xs) xss
```

```
-----
-- Ejercicio 6.4. Definir, mediante plegado con foldl, la función
--   sumllAP :: Num a => [[a]] -> a
-- tal que (sumllAP xss) es la suma de las sumas de las listas de xss. Por
-- ejemplo,
```

```

--      sumllAP [[1,3],[2,5]] == 11
-----

sumllAP :: Num a => [[a]] -> a
sumllAP = foldl (\a xs -> a + sum xs) 0

-----

-- Ejercicio 7.1. Definir, mediante recursión, la función
--      borraR :: Eq a => a -> a -> [a]
-- tal que (borraR y xs) es la lista obtenida borrando las ocurrencias de
-- y en xs. Por ejemplo,
--      borraR 5 [2,3,5,6] == [2,3,6]
--      borraR 5 [2,3,5,6,5] == [2,3,6]
--      borraR 7 [2,3,5,6,5] == [2,3,5,6,5]
-----

borraR :: Eq a => a -> [a] -> [a]
borraR z [] = []
borraR z (x:xs) | z == x = borraR z xs
                 | otherwise = x : borraR z xs

-----

-- Ejercicio 7.2. Definir, mediante plegado, la función
--      borraP :: Eq a => a -> a -> [a]
-- tal que (borraP y xs) es la lista obtenida borrando las ocurrencias de
-- y en xs. Por ejemplo,
--      borraP 5 [2,3,5,6] == [2,3,6]
--      borraP 5 [2,3,5,6,5] == [2,3,6]
--      borraP 7 [2,3,5,6,5] == [2,3,5,6,5]
-----

borraP :: Eq a => a -> [a] -> [a]
borraP z = foldr f []
  where f x y | z == x = y
           | otherwise = x:y

-- La definición por plegado con lambda es es
borraP' :: Eq a => a -> [a] -> [a]
borraP' z = foldr (\x y -> if z==x then y else x:y) []

```

```

-----
-- Ejercicio 8.1. Definir, mediante recursión, la función
--   diferenciaR :: Eq a => [a] -> [a] -> [a]
-- tal que (diferenciaR xs ys) es la diferencia del conjunto xs e ys; es
-- decir el conjunto de los elementos de xs que no pertenecen a ys. Por
-- ejemplo,
--   diferenciaR [2,3,5,6] [5,2,7] == [3,6]
-----

```

```

diferenciaR :: Eq a => [a] -> [a] -> [a]
diferenciaR xs ys = aux xs xs ys
  where aux a xs [] = a
        aux a xs (y:ys) = aux (borraR y a) xs ys

```

```

-- La definición, para aproximarse al patrón foldr, se puede escribir como
diferenciaR' :: Eq a => [a] -> [a] -> [a]
diferenciaR' xs ys = aux xs xs ys
  where aux a xs [] = a
        aux a xs (y:ys) = aux (flip borraR a y) xs ys

```

```

-----
-- Ejercicio 8.2. Definir, mediante plegado con foldl, la función
--   diferenciaP :: Eq a => [a] -> [a] -> [a]
-- tal que (diferenciaP xs ys) es la diferencia del conjunto xs e ys; es
-- decir el conjunto de los elementos de xs que no pertenecen a ys. Por
-- ejemplo,
--   diferenciaP [2,3,5,6] [5,2,7] == [3,6]
-----

```

```

diferenciaP :: Eq a => [a] -> [a] -> [a]
diferenciaP xs ys = foldl (flip borraR) xs ys

```

```

-- La definición anterior puede simplificarse a
diferenciaP' :: Eq a => [a] -> [a] -> [a]
diferenciaP' = foldl (flip borraR)

```

```

-----
-- Ejercicio 9.1. Definir mediante plegado la función
--   producto :: Num a => [a] -> a
-- tal que (producto xs) es el producto de los elementos de la lista

```

```
-- xs. Por ejemplo,
--   producto [2,1,-3,4,5,-6] == 720
-----
```

```
producto :: Num a => [a] -> a
producto = foldr (*) 1
```

```
-- Ejercicio 9.2. Definir mediante plegado la función
--   productoPred :: Num a => (a -> Bool) -> [a] -> a
-- tal que (productoPred p xs) es el producto de los elementos de la
-- lista xs que verifican el predicado p. Por ejemplo,
--   productoPred even [2,1,-3,4,-5,6] == 48
-----
```

```
productoPred :: Num a => (a -> Bool) -> [a] -> a
productoPred p = foldr (\x y -> if p x then x*y else y) 1
```

```
-- Ejercicio 9.3. Definir la función
--   productoPos :: (Num a, Ord a) => [a] -> a
-- tal que (productoPos xs) es el producto de los elementos estrictamente
-- positivos de la lista xs. Por ejemplo,
--   productoPos [2,1,-3,4,-5,6] == 48
-----
```

```
productoPos :: (Num a, Ord a) => [a] -> a
productoPos = productoPred (>0)
```

```
-- Ejercicio 10.1. Se denomina cola de una lista xs a una sublista no
-- vacía de xs formada por un elemento y los siguientes hasta el
-- final. Por ejemplo, [3,4,5] es una cola de la lista [1,2,3,4,5].
--
```

```
-- Definir la función
--   colas :: [a] -> [[a]]
-- tal que (colas xs) es la lista de las colas de la lista xs. Por
-- ejemplo,
--   colas []           == [[]]
--   colas [1,2]       == [[1,2],[2],[1]]
```

```

--      colas [4,1,2,5] == [[4,1,2,5],[1,2,5],[2,5],[5],[[]]]
-----

colas :: [a] -> [[a]]
colas []      = [[]]
colas (x:xs) = (x:xs) : colas xs

-----

-- Ejercicio 10.2. Comprobar con QuickCheck que las funciones colas y
-- tails son equivalentes.
-----

-- La propiedad es
prop_colas :: [Int] -> Bool
prop_colas xs = colas xs == tails xs

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

-----

-- Ejercicio 10.3. Se denomina cabeza de una lista xs a una sublista no
-- vacía de la formada por el primer elemento y los siguientes hasta uno
-- dado. Por ejemplo, [1,2,3] es una cabeza de [1,2,3,4,5].
--
-- Definir, por recursión, la función
-- cabezas :: [a] -> [[a]]
-- tal que (cabezas xs) es la lista de las cabezas de la lista xs. Por
-- ejemplo,
-- cabezas []           == [[]]
-- cabezas [1,4]       == [[],[1],[1,4]]
-- cabezas [1,4,5,2,3] == [[],[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-----

cabezas :: [a] -> [[a]]
cabezas []      = [[]]
cabezas (x:xs) = [] : [x:ys | ys <- cabezas xs]

-----

-- Ejercicio 10.4. Definir, por plegado, la función

```

```

--   cabezasP :: [a] -> [[a]]
--   tal que (cabezasP xs) es la lista de las cabezas de la lista xs. Por
--   ejemplo,
--   cabezasP []           == [[]]
--   cabezasP [1,4]       == [[],[1],[1,4]]
--   cabezasP [1,4,5,2,3] == [[],[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-----

```

```

cabezasP :: [a] -> [[a]]
cabezasP = foldr (\x y -> []:[x:ys | ys <- y]) [[]]
-----

```

```

--   Ejercicio 10.5. Definir, mediante funciones de orden superior, la
--   función
--   cabezasS :: [a] -> [[a]]
--   tal que (cabezasS xs) es la lista de las cabezas de la lista xs. Por
--   ejemplo,
--   cabezasS []           == [[]]
--   cabezasS [1,4]       == [[],[1],[1,4]]
--   cabezasS [1,4,5,2,3] == [[],[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-----

```

```

cabezasS :: [a] -> [[a]]
cabezasS xs = reverse (map reverse (colas (reverse xs)))
-----

```

```

--   La anterior definición puede escribirse sin argumentos como
cabezasS' :: [a] -> [[a]]
cabezasS' = reverse . map reverse . (colas . reverse)
-----

```

```

--   Ejercicio 10.6. Comprobar con QuickCheck que las funciones cabezas y
--   inits son equivalentes.
-----

```

```

--   La propiedad es
prop_cabezas :: [Int] -> Bool
prop_cabezas xs = cabezas xs == inits xs
-----

```

```

--   La comprobación es
--   ghci> quickCheck prop_cabezas
-----

```



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



# Relación 16

## Evaluación perezosa y listas infinitas (1)

```
-----  
-- Introducción  
-----  
  
-- En esta relación se presentan ejercicios con listas infinitas y  
-- evaluación perezosa. Estos ejercicios corresponden al tema 10 cuyas  
-- transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-10.pdf  
-----  
-- Importación de librerías auxiliares  
-----  
  
import Test.QuickCheck  
  
-----  
-- Ejercicio 1.1. Definir, por recursión, la función  
-- repite :: a -> [a]  
-- tal que (repite x) es la lista infinita cuyos elementos son x. Por  
-- ejemplo,  
-- repite 5 == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...]  
-- take 3 (repite 5) == [5,5,5]  
-- Nota: La función repite es equivalente a la función repeat definida  
-- en el preludio de Haskell.  
-----
```

```
repite :: a -> [a]
repite x = x : repite x
```

```
-----
-- Ejercicio 1.2. Definir, por comprensión, la función
--   repiteC :: a -> [a]
-- tal que (repiteC x) es la lista infinita cuyos elementos son x. Por
-- ejemplo,
--   repiteC 5           == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
--   take 3 (repiteC 5) == [5,5,5]
-- Nota: La función repiteC es equivalente a la función repeat definida
-- en el prelude de Haskell.
-----
```

```
repiteC :: a -> [a]
repiteC x = [x | _ <- [1..]]
```

```
-----
-- Ejercicio 2.1. Definir, por recursión, la función
--   repiteFinita :: Int -> a -> [a]
-- tal que (repiteFinita n x) es la lista con n elementos iguales a
-- x. Por ejemplo,
--   repiteFinita 3 5 == [5,5,5]
-- Nota: La función repiteFinita es equivalente a la función replicate
-- definida en el prelude de Haskell.
-----
```

```
repiteFinita :: Int -> a -> [a]
repiteFinita 0 x = []
repiteFinita n x = x : repiteFinita (n-1) x
```

```
-----
-- Ejercicio 2.2. Definir, por comprensión, la función
--   repiteFinitaC :: Int -> a -> [a]
-- tal que (repiteFinitaC n x) es la lista con n elementos iguales a
-- x. Por ejemplo,
--   repiteFinitaC 3 5 == [5,5,5]
-- Nota: La función repiteFinitaC es equivalente a la función replicate
-- definida en el prelude de Haskell.
-----
```

```
-----  
repiteFinitaC :: Int -> a -> [a]  
repiteFinitaC n x = [x | _ <- [1..n]]  
-----
```

```
-- Ejercicio 2.3. Definir, usando repite, la función  
-- repiteFinita' :: Int-> a -> [a]  
-- tal que (repiteFinita' n x) es la lista con n elementos iguales a  
-- x. Por ejemplo,  
-- repiteFinita' 3 5 == [5,5,5]  
-- Nota: La función repiteFinita' es equivalente a la función replicate  
-- definida en el preludio de Haskell.  
-----
```

```
repiteFinita' :: Int -> a -> [a]  
repiteFinita' n x = take n (repite x)  
-----
```

```
-- Ejercicio 3.1. Definir, por comprensión, la función  
-- ecoC :: String -> String  
-- tal que (ecoC xs) es la cadena obtenida a partir de la cadena xs  
-- repitiendo cada elemento tantas veces como indica su posición: el  
-- primer elemento se repite 1 vez, el segundo 2 veces y así  
-- sucesivamente. Por ejemplo,  
-- ecoC "abcd" == "abbcccdddd"  
-----
```

```
ecoC :: String -> String  
ecoC xs = concat [replicate i x | (i,x) <- zip [1..] xs]  
-----
```

```
-- Ejercicio 3.2. Definir, por recursión, la función  
-- ecoR :: String -> String  
-- tal que (ecoR xs) es la cadena obtenida a partir de la cadena xs  
-- repitiendo cada elemento tantas veces como indica su posición: el  
-- primer elemento se repite 1 vez, el segundo 2 veces y así  
-- sucesivamente. Por ejemplo,  
-- ecoR "abcd" == "abbcccdddd"  
-----
```

```
ecoR :: String -> String
```

```
ecoR xs = aux 1 xs
```

```
  where aux n [] = []
```

```
        aux n (x:xs) = replicate n x ++ aux (n+1) xs
```

```
-----
-- Ejercicio 4. Definir, usando takeWhile y map, la función
--   potenciasMenores :: Int -> Int -> [Int]
-- tal que (potenciasMenores x y) es la lista de las potencias de x
-- menores que y. Por ejemplo,
--   potenciasMenores 2 1000 == [2,4,8,16,32,64,128,256,512]
-----
```

```
potenciasMenores :: Int -> Int -> [Int]
```

```
potenciasMenores x y = takeWhile (<y) (map (x^) [1..])
```

```
-----
-- Ejercicio 5. (Problema 303 del proyecto Euler) Definir la función
--   multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
-- tal que (multiplosRestringidos n x) es la lista de los múltiplos de n
-- tales que todas sus dígitos verifican la propiedad p. Por ejemplo,
--   take 4 (multiplosRestringidos 5 (<=3)) == [10,20,30,100]
--   take 5 (multiplosRestringidos 3 (<=4)) == [3,12,21,24,30]
--   take 5 (multiplosRestringidos 3 even) == [6,24,42,48,60]
-----
```

```
multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
```

```
multiplosRestringidos n p =
```

```
  [y | y <- [n,2*n..], all p (digitos y)]
```

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

```
--   digitos 327 == [3,2,7]
```

```
digitos :: Int -> [Int]
```

```
digitos n = [read [x] | x <- show n]
```

```
-----
-- Ejercicio 5. Definir, por recursión, la función
--   itera :: (a -> a) -> a -> [a]
-- tal que (itera f x) es la lista cuyo primer elemento es x y los
```

```

-- siguientes elementos se calculan aplicando la función f al elemento
-- anterior. Por ejemplo,
--   ghci> itera (+1) 3
--   [3,4,5,6,7,8,9,10,11,12,{Interrupted!}]
--   ghci> itera (*2) 1
--   [1,2,4,8,16,32,64,{Interrupted!}]
--   ghci> itera ('div' 10) 1972
--   [1972,197,19,1,0,0,0,0,0,0,0,{Interrupted!}]
-- Nota: La función repite es equivalente a la función iterate definida
-- en el preludio de Haskell.

```

```

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

```

```

-----
-- Ejercicio 6.1. Definir, por recursión, la función
--   agrupa :: Int -> [a] -> [[a]]
-- tal que (agrupa n xs) es la lista formada por listas de n elementos
-- consecutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--   ghci> agrupa 2 [3,1,5,8,2,7]
--   [[3,1],[5,8],[2,7]]
--   ghci> agrupa 2 [3,1,5,8,2,7,9]
--   [[3,1],[5,8],[2,7],[9]]
--   ghci> agrupa 5 "todo necio confunde valor y precio"
--   ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]

```

```

-----
agrupa :: Int -> [a] -> [[a]]
agrupa n [] = []
agrupa n xs = take n xs : agrupa n (drop n xs)

```

```

-----
-- Ejercicio 6.2. Definir, de manera no recursiva, la función
--   agrupa' :: Int -> [a] -> [[a]]
-- tal que (agrupa' n xs) es la lista formada por listas de n elementos
-- consecutivos de la lista xs (salvo posiblemente la última que puede
-- tener menos de n elementos). Por ejemplo,
--   ghci> agrupa' 2 [3,1,5,8,2,7]

```

```
-- [[3,1],[5,8],[2,7]]
-- ghci> agrupa' 2 [3,1,5,8,2,7,9]
-- [[3,1],[5,8],[2,7],[9]]
-- ghci> agrupa' 5 "todo necio confunde valor y precio"
-- ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-----
```

```
agrupa' :: Int -> [a] -> [[a]]
agrupa' n = takeWhile (not . null)
           . map (take n)
           . iterate (drop n)
```

```
-- Puede verse su funcionamiento en el siguiente ejemplo,
-- iterate (drop 2) [5..10]
-- ==> [[5,6,7,8,9,10],[7,8,9,10],[9,10],[],[],...]
-- map (take 2) (iterate (drop 2) [5..10])
-- ==> [[5,6],[7,8],[9,10],[],[],[],[],...]
-- takeWhile (not . null) (map (take 2) (iterate (drop 2) [5..10]))
-- ==> [[5,6],[7,8],[9,10]]
-----
```

```
-- Ejercicio 6.3. Definir, y comprobar, con QuickCheck las dos propiedades
-- que caracterizan a la función agrupa:
-- * todos los grupos tienen que tener la longitud determinada (salvo el
-- último que puede tener una longitud menor) y
-- * combinando todos los grupos se obtiene la lista inicial.
-----
```

```
-- La primera propiedad es
prop_AgruparLongitud :: Int -> [Int] -> Property
prop_AgruparLongitud n xs =
  n > 0 && not (null gs) ==>
    and [length g == n | g <- init gs] &&
    0 < length (last gs) && length (last gs) <= n
  where gs = agrupa n xs
```

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



```

-- La segunda propiedad es
prop_AgruparCombinar :: Int -> [Int] -> Property
prop_AgruparCombinar n xs =
  n > 0 ==> concat (agrupa n xs) == xs

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

-----
-- Ejercicio 7.1. Sea la siguiente operación, aplicable a cualquier
-- número entero positivo:
-- * Si el número es par, se divide entre 2.
-- * Si el número es impar, se multiplica por 3 y se suma 1.
-- Dado un número cualquiera, podemos considerar su órbita, es decir,
-- las imágenes sucesivas al iterar la función. Por ejemplo, la órbita
-- de 13 es
-- 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
-- Si observamos este ejemplo, la órbita de 13 es periódica, es decir,
-- se repite indefinidamente a partir de un momento dado). La conjetura
-- de Collatz dice que siempre alcanzaremos el 1 para cualquier número
-- con el que comencemos. Ejemplos:
-- * Empezando en n = 6 se obtiene 6, 3, 10, 5, 16, 8, 4, 2, 1.
-- * Empezando en n = 11 se obtiene: 11, 34, 17, 52, 26, 13, 40, 20,
-- 10, 5, 16, 8, 4, 2, 1.
-- * Empezando en n = 27, la sucesión tiene 112 pasos, llegando hasta
-- 9232 antes de descender a 1: 27, 82, 41, 124, 62, 31, 94, 47,
-- 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274,
-- 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263,
-- 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502,
-- 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
-- 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644,
-- 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308,
-- 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
-- 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,
-- 16, 8, 4, 2, 1.
--
-- Definir la función
-- siguiente :: Integer -> Integer

```

```
-- tal que (siguiente n) es el siguiente de n en la sucesión de
-- Collatz. Por ejemplo,
--   siguiente 13 == 40
--   siguiente 40 == 20
-----
```

```
siguiente n | even n    = n `div` 2
            | otherwise = 3*n+1
-----
```

```
-- Ejercicio 7.2. Definir, por recursión, la función
--   collatz :: Integer -> [Integer]
-- tal que (collatz n) es la órbita de Collatz de n hasta alcanzar el
-- 1. Por ejemplo,
--   collatz 13 == [13,40,20,10,5,16,8,4,2,1]
-----
```

```
collatz :: Integer -> [Integer]
collatz 1 = [1]
collatz n = n : collatz (siguiente n)
-----
```

```
-- Ejercicio 7.3. Definir, sin recursión, la función
--   collatz' :: Integer -> [Integer]
-- tal que (collatz' n) es la órbita de Collatz de n hasta alcanzar el
-- 1. Por ejemplo,
--   collatz' 13 == [13,40,20,10,5,16,8,4,2,1]
-- Indicación: Usar takeWhile e iterate.
-----
```

```
collatz' :: Integer -> [Integer]
collatz' n = (takeWhile (/=1) (iterate siguiente n)) ++ [1]
-----
```

```
-- Ejercicio 7.4. Definir la función
--   menorCollatzMayor :: Int -> Integer
-- tal que (menorCollatzMayor x) es el menor número cuya órbita de
-- Collatz tiene más de x elementos. Por ejemplo,
--   menorCollatzMayor 100 == 27
-----
```

```
menorCollatzMayor :: Int -> Integer
menorCollatzMayor x = head [y | y <- [1..], length (collatz y) > x]
```

```
-----
-- Ejercicio 7.5. Definir la función
--   menorCollatzSupera :: Integer -> Integer
-- tal que (menorCollatzSupera x) es el menor número cuya órbita de
-- Collatz tiene algún elemento mayor que x. Por ejemplo,
--   menorCollatzSupera 100 == 15
-----
```

```
menorCollatzSupera :: Integer -> Integer
menorCollatzSupera x =
  head [y | y <- [1..], maximum (collatz y) > x]
```

```
-- Otra definición alternativa es
```

```
menorCollatzSupera' :: Integer -> Integer
menorCollatzSupera' x = head [n | n <- [1..], t <- collatz' n, t > x]
```



# Relación 17

## Evaluación perezosa y listas infinitas (2)

```
-----  
-- Introducción -----  
-----  
  
-- En esta relación se presentan ejercicios con listas infinitas y  
-- evaluación perezosa. Estos ejercicios corresponden al tema 10 cuyas  
-- transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-10.pdf  
  
-----  
-- Ejercicio 1.1. Definir, usando la criba de Eratóstenes, la constante  
-- primos :: Integral a => [a]  
-- cuyo valor es la lista de los números primos. Por ejemplo,  
-- take 10 primos == [2,3,5,7,11,13,17,19,23,29]  
-----  
  
primos :: Integral a => [a]  
primos = criba [2..]  
  where criba [] = []  
        criba (n:ns) = n : criba (elimina n ns)  
        elimina n xs = [x | x <- xs, x `mod` n /= 0]  
  
-----  
-- Ejercicio 1.2. Definir la función  
-- primo :: Integral a => a -> Bool
```

```
-- tal que (primo n) se verifica si n es primo. Por ejemplo,
--   primo 7 == True
--   primo 9 == False
```

```
-----

primo :: Int -> Bool
primo n = head (dropWhile (<n) primos) == n
```

```
-----

-- Ejercicio 6. Definir la función
--   sumaDeDosPrimos :: Int -> [(Int,Int)]
-- tal que (sumaDeDosPrimos n) es la lista de las distintas
-- descomposiciones de n como suma de dos números primos. Por ejemplo,
--   sumaDeDosPrimos 30 == [(7,23),(11,19),(13,17)]
--   sumaDeDosPrimos 10 == [(3,7),(5,5)]
-- Calcular, usando la función sumaDeDosPrimos, el menor número que
-- puede escribirse de 10 formas distintas como suma de dos primos.
```

```
-----

sumaDeDosPrimos :: Int -> [(Int,Int)]
sumaDeDosPrimos n =
  [(x,n-x) | x <- primosN, x <= n-x, elem (n-x) primosN]
  where primosN = takeWhile (<=n) primos
```

```
-- El cálculo es
--   ghci> head [x | x <- [1..], length (sumaDeDosPrimos x) == 10]
--   114
```

```
-----

-- Ejercicio 3. Definir la función
--   esProductoDeDosPrimos :: Int -> Bool
-- tal que (esProductoDeDosPrimos n) se verifica si n es el producto de
-- dos primos distintos. Por ejemplo,
--   esProductoDeDosPrimos 6 == True
--   esProductoDeDosPrimos 9 == False
```

```
-----

esProductoDeDosPrimos :: Int -> Bool
esProductoDeDosPrimos n =
```

```

[x | x <- primosN,
    mod n x == 0,
    div n x /= x,
    elem (div n x) primosN] /= []
where primosN = takeWhile (<=n) primos

```

```

-----
-- Ejercicio 4.1. [Problema 37 del proyecto Euler] Un número primo es
-- truncable si los números que se obtienen eliminado cifras, de derecha
-- a izquierda, son primos. Por ejemplo, 599 es un primo truncable
-- porque 599, 59 y 5 son primos; en cambio, 577 es un primo no
-- truncable porque 57 no es primo.

```

```

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

```

```

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

```

```

-----
-- Ejercicio 4.2. Definir la función
--   sumaPrimosTruncables :: Int -> Int
-- tal que (sumaPrimosTruncables n) es la suma de los n primeros primos
-- truncables. Por ejemplo,
--   sumaPrimosTruncables 10 == 249
-- Calcular la suma de los 20 primos truncables.

```

```

sumaPrimosTruncables :: Int -> Int
sumaPrimosTruncables n =
  sum (take n [x | x <- primos, primoTruncable x])

```

```

-- El cálculo es

```

```

-- ghci> sumaPrimosTruncables 20
-- 2551

-----

-- Ejercicio 5.1. Definir la función
--   intercala :: a -> [a] -> [[a]]
-- tal que (intercala x ys) es la lista de las listas obtenidas
-- intercalando x entre los elementos de ys. Por ejemplo,
--   intercala 1 [2,3] == [[1,2,3],[2,1,3],[2,3,1]]
-----

-- Una definición recursiva es
intercala1 :: a -> [a] -> [[a]]
intercala1 x [] = [[x]]
intercala1 x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala1 x ys]

-- Otra definición, más eficiente, es
intercala :: a -> [a] -> [[a]]
intercala y xs =
  [take n xs ++ (y : drop n xs) | n <- [0..length xs]]

-- Por ejemplo,
-- ghci> length (intercala1 1 [2..10^4])
-- 10000
-- (11.63 secs, 3016915472 bytes)
-- ghci> length (intercala 1 [2..10^4])
-- 10000
-- (0.02 secs, 2631448 bytes)

-----

-- Ejercicio 5.2. Definir la función
--   permutaciones :: [a] -> [[a]]
-- tal que (permutaciones xs) es la lista de las permutaciones de la
-- lista xs. Por ejemplo,
--   permutaciones "bc" == ["bc","cb"]
--   permutaciones "abc" == ["abc","bac","bca","acb","cab","cba"]
-----

permutaciones :: [a] -> [[a]]
permutaciones [] = [[]]

```



```

permutaciones (x:xs) =
  concat [intercala x ys | ys <- permutaciones xs]

```

```

-----
-- Ejercicio 5.3. Definir la función
--   permutacionesN :: Integer -> [Integer]
-- tal que (permutacionesN x) es la lista de los números obtenidos
-- permutando las cifras de x. Por ejemplo,
--   permutacionesN 352 == [352,532,523,325,235,253]
-----

```

```

permutacionesN :: Int -> [Int]
permutacionesN x = [read ys | ys <- permutaciones (show x)]

```

```

-----
-- Ejercicio 5.4. Un primo permutable es un número primo tal que todos
-- los números obtenidos permutando sus cifras son primos. Por ejemplo,
-- 337 es un primo permutable ya que 337, 373 y 733 son primos.
--

```

```

-- Definir la función
--   primoPermutable :: Integer -> Bool
-- tal que (primoPermutable x) se verifica si x es un primo
-- permutable. Por ejemplo,
--   primoPermutable 17 == True
--   primoPermutable 19 == False
-----

```

```

primoPermutable :: Int -> Bool
primoPermutable x = and [primo y | y <- permutacionesN x]

```

```

-----
-- Ejercicio 6.1. Los números enteros se pueden ordenar como sigue
--   0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5, -6, 6, -7, 7, ...
-- Definir, por comprensión, la constante
--   enteros :: [Int]
-- tal que enteros es la lista de los enteros con la ordenación
-- anterior. Por ejemplo,
--   take 10 enteros == [0,-1,1,-2,2,-3,3,-4,4,-5]
-----

```

```
enteros :: [Int]
enteros = 0 : concat [[-x,x] | x <- [1..]]
```

```
-----
-- Ejercicio 6.2. Definir, por iteración, la constante
--   enteros' :: [Int]
-- tal que enteros' es la lista de los enteros con la ordenación
-- anterior. Por ejemplo,
--   take 10 enteros == [0,-1,1,-2,2,-3,3,-4,4,-5]
-----
```

```
enteros' :: [Int]
enteros' = iterate siguiente 0
  where siguiente x | x >= 0    = -x-1
                  | otherwise = -x
```

```
-----
-- Ejercicio 6.3. Definir, por selección con takeWhile, la función
--   posicion :: Int -> Int
-- tal que (posicion x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
--   posicion 2 == 4
-----
```

```
posicion :: Int -> Int
posicion x = length (takeWhile (/=x) enteros)
```

```
-----
-- Ejercicio 6.4. Definir, por recursión, la función
--   posicionR :: Int -> Int
-- tal que (posicionR x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
--   posicionR 2 == 4
-----
```

```
posicionR :: Int -> Int
posicionR x = aux enteros 0
  where aux (y:ys) n | x == y    = n
                  | otherwise = aux ys (n+1)
```

```

-----
-- Ejercicio 6.5. Definir, por comprensión, la función
--   posicionC :: Int -> Int
-- tal que (posicionC x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
--   posicionC 2 == 4
-----

```

```

posicionC :: Int -> Int
posicionC x = head [n | (n,y) <- zip [0..] enteros, y == x]

```

```

-----
-- Ejercicio 6.6. Definir, sin búsqueda, la función
--   posicion2 :: Int -> Int
-- tal que (posicion2 x) es la posición del entero x en la ordenación
-- anterior. Por ejemplo,
--   posicion2 2 == 4
-----

```

```

-- Definición directa
posicion2 :: Int -> Int
posicion2 x | x >= 0    = 2*x
              | otherwise = 2*(-x)-1

```

```

-----
-- Ejercicio 7. (Problema 10 del Proyecto Euler) Definir la función
--   sumaPrimoMenores :: Integer -> Integer
-- tal que (sumaPrimoMenores n) es la suma de los primos menores que
-- n. Por ejemplo,
--   sumaPrimoMenores 10 == 17
-----

```

```

-- Por recursión
sumaPrimoMenores :: Integer -> Integer
sumaPrimoMenores n = sumaMenores n primos 0
  where sumaMenores n (x:xs) a | n <= x    = a
                                | otherwise = sumaMenores n xs (a+x)

```

```

-- Por comprensión
sumaPrimoMenores2 :: Integer -> Integer

```

```
sumaPrimoMenores2 n = sum (takeWhile (<n) primos)
```

```
-----
-- Ejercicio 8. (Problema 12 del Proyecto Euler) La sucesión de los
-- números triangulares se obtiene sumando los números naturales. Así,
-- el 7º número triangular es
--   1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.
-- Los primeros 10 números triangulares son
--   1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
-- Los divisores de los primeros 7 números triangulares son:
--   1: 1
--   3: 1,3
--   6: 1,2,3,6
--  10: 1,2,5,10
--  15: 1,3,5,15
--  21: 1,3,7,21
--  28: 1,2,4,7,14,28
-- Como se puede observar, 28 es el menor número triangular con más de 5
-- divisores.
--
-- Definir la función
--   euler12 :: Int -> Integer
-- tal que (euler12 n) es el menor número triangular con más de n
-- divisores. Por ejemplo,
--   euler12 5 == 28
-----
```

```
euler12 :: Int -> Integer
```

```
euler12 n = head [x | x <- triangulares, nDivisores x > n]
```

```
-- triangulares es la lista de los números triangulares
--   take 10 triangulares == [1,3,6,10,15,21,28,36,45,55]
```

```
triangulares :: [Integer]
```

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

```
-- Otra definición de triangulares es
```

```
triangulares' :: [Integer]
```

```
triangulares' = scanl (+) 1 [2..]
```

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

```

-- divisores 28 == [1,2,4,7,14,28]
divisores :: Integer -> [Integer]
divisores x = [y | y <- [1..x], mod x y == 0]

-- (nDivisores n) es el número de los divisores de n. Por ejemplo,
-- nDivisores 28 == 6
nDivisores :: Integer -> Int
nDivisores x = length (divisores x)

-----
-- Ejercicio 9.1. Dos números son equivalentes si la media de sus cifras
-- son iguales. Por ejemplo, 3205 y 41 son equivalentes ya que
--  $(3+2+0+5)/4 = (4+1)/2$ . Definir la función
-- equivalentes :: Int -> Int -> Bool
-- tal que (equivalentes x y) se verifica si los números x e y son
-- equivalentes. Por ejemplo,
-- equivalentes 3205 41 == True
-- equivalentes 3205 25 == False
-----

equivalentes :: Int -> Int -> Bool
equivalentes x y = media (cifras x) == media (cifras y)

-- (cifras n) es la lista de las cifras de n. Por ejemplo,
-- cifras 3205 == [3,2,0,5]
cifras :: Int -> [Int]
cifras n = [read [y] | y <- show n]

-- (media xs) es la media de la lista xs. Por ejemplo,
-- media [3,2,0,5] == 2.5
media :: [Int] -> Float
media xs = (fromIntegral (sum xs)) / (fromIntegral (length xs))

-----
-- Ejercicio 9.2. Definir la función
-- relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
-- relacionados (<) [2,3,7,9] == True
-- relacionados (<) [2,3,1,9] == False

```

```
-- relacionados equivalentes [3205,50,5014] == True
```

```
relacionados :: (a -> a -> Bool) -> [a] -> Bool
relacionados r (x:y:zs) = (r x y) && relacionados r (y:zs)
relacionados _ _ = True
```

```
-- Una definición alternativa es
```

```
relacionados' :: (a -> a -> Bool) -> [a] -> Bool
relacionados' r xs = and [r x y | (x,y) <- zip xs (tail xs)]
```

```
-- Ejercicio 9.3. Definir la función
```

```
-- primosEquivalentes :: Int -> [[Int]]
-- tal que (primosEquivalentes n) es la lista de las sucesiones de n
-- números primos consecutivos equivalentes. Por ejemplo,
-- take 2 (primosEquivalentes 2) == [[523,541],[1069,1087]]
-- head (primosEquivalentes 3) == [22193,22229,22247]
```

```
primosEquivalentes :: Int -> [[Int]]
primosEquivalentes n = aux primos
  where aux (x:xs) | relacionados equivalentes ys = ys : aux xs
            | otherwise = aux xs
        where ys = take n (x:xs)
```

```
-- Ejercicio 10. Definir la función
```

```
-- perteneceRango :: Int -> (Int -> Int) -> Bool
-- tal que (perteneceRango x f) se verifica si x pertenece al rango de
-- la función f, suponiendo que f es una función creciente cuyo dominio
-- es el conjunto de los números naturales. Por ejemplo,
-- perteneceRango 5 (\x -> 2*x+1) == True
-- perteneceRango 1234 (\x -> 2*x+1) == False
```

```
perteneceRango :: Int -> (Int -> Int) -> Bool
perteneceRango y f = elem y (takeWhile (<=y) (imagenes f))
  where imagenes f = [f x | x <- [0..]]
```

```

-----
-- Ejercicio 11.1. Definir, por recursión, la función
--   paresOrdenados :: [a] -> [(a,a)]
-- tal que (paresOrdenados xs) es la lista de todos los pares de
-- elementos (x,y) de xs, tales que x ocurren en xs antes que y. Por
-- ejemplo,
--   paresOrdenados [3,2,5,4] == [(3,2),(3,5),(3,4),(2,5),(2,4),(5,4)]
--   paresOrdenados [3,2,5,3] == [(3,2),(3,5),(3,3),(2,5),(2,3),(5,3)]
-----

```

```

paresOrdenados :: [a] -> [(a,a)]
paresOrdenados [] = []
paresOrdenados (x:xs) = [(x,y) | y <- xs] ++ paresOrdenados xs

```

```

-----
-- Ejercicio 11.2. Definir, por plegado, la función
--   paresOrdenados2 :: [a] -> [(a,a)]
-- tal que (paresOrdenados2 xs) es la lista de todos los pares de
-- elementos (x,y) de xs, tales que x ocurren en xs antes que y. Por
-- ejemplo,
--   paresOrdenados2 [3,2,5,4] == [(3,2),(3,5),(3,4),(2,5),(2,4),(5,4)]
--   paresOrdenados2 [3,2,5,3] == [(3,2),(3,5),(3,3),(2,5),(2,3),(5,3)]
-----

```

```

paresOrdenados2 :: [a] -> [(a,a)]
paresOrdenados2 [] = []
paresOrdenados2 (x:xs) =
  foldr (\y ac -> (x,y):ac) (paresOrdenados2 xs) xs

```

```

-----
-- Ejercicio 11.3. Definir, usando repeat, la función
--   paresOrdenados3 :: [a] -> [(a,a)]
-- tal que (paresOrdenados3 xs) es la lista de todos los pares de
-- elementos (x,y) de xs, tales que x ocurren en xs antes que y. Por
-- ejemplo,
--   paresOrdenados3 [3,2,5,4] == [(3,2),(3,5),(3,4),(2,5),(2,4),(5,4)]
--   paresOrdenados3 [3,2,5,3] == [(3,2),(3,5),(3,3),(2,5),(2,3),(5,3)]
-----

```

```

paresOrdenados3 :: [a] -> [(a,a)]

```

```

paresOrdenados3 [] = []
paresOrdenados3 (x:xs) = zip (repeat x) xs ++ paresOrdenados3 xs

```

```

-----
-- Ejercicio 12.1. Definir, por recursión, la función
-- potenciaFunc :: Int -> (a -> a) -> a -> a
-- tal que (potenciaFunc n f x) es el resultado de aplicar n veces la
-- función f a x. Por ejemplo,
-- potenciaFunc 3 (*10) 5 == 5000
-- potenciaFunc 4 (+10) 5 == 45
-----

```

```

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

```

```

-----
-- Ejercicio 12.2. Definir, sin recursión, la función
-- potenciaFunc2 :: Int -> (a -> a) -> a -> a
-- tal que (potenciaFunc2 n f x) es el resultado de aplicar n veces la
-- función f a x. Por ejemplo,
-- potenciaFunc2 3 (*10) 5 == 5000
-- potenciaFunc2 4 (+10) 5 == 45
-----

```

```

potenciaFunc2 :: Int -> (a -> a) -> a -> a
potenciaFunc2 n f x = last (take (n+1) (iterate f x))

```

```

-----
-- Ejercicio 13.1. Cuentan que Alan Turing tenía una bicicleta vieja,
-- que tenía una cadena con un eslabón débil y además uno de los radios
-- de la rueda estaba doblado. Cuando el radio doblado coincidía con el
-- eslabón débil, entonces la cadena se rompía.
--
-- La bicicleta se identifica por los parámetros (i,d,n) donde
-- * i es el número del eslabón que coincide con el radio doblado al
-- empezar a andar,
-- * d es el número de eslabones que se desplaza la cadena en cada
-- vuelta de la rueda y
-- * n es el número de eslabones de la cadena (el número n es el débil).

```



```

-- Si  $i=2$  y  $d=7$  y  $n=25$ , entonces la lista con el número de eslabón que
-- toca el radio doblado en cada vuelta es
--   [2,9,16,23,5,12,19,1,8,15,22,4,11,18,0,7,14,21,3,10,17,24,6,...
-- Con lo que la cadena se rompe en la vuelta número 14.
--
-- Definir la función
--   eslabones :: Int -> Int -> Int -> [Int]
-- tal que (eslabones i d n) es la lista con los números de eslabones
-- que tocan el radio doblado en cada vuelta en una bicicleta de tipo
-- (i,d,n). Por ejemplo,
--   take 10 (eslabones 2 7 25) == [2,9,16,23,5,12,19,1,8,15]

```

```

eslabones :: Int -> Int -> Int -> [Int]
eslabones i d n = [(i+d*j) `mod` n | j <- [0..]]

```

```

-- 2ª definición (con iterate):
eslabones2 :: Int -> Int -> Int -> [Int]
eslabones2 i d n = map (\x-> mod x n) (iterate (+d) i)

```

```

-----
-- Ejercicio 13.2. Definir la función
--   numeroVueltas :: Int -> Int -> Int -> Int
-- tal que (numeroVueltas i d n) es el número de vueltas que pasarán
-- hasta que la cadena se rompa en una bicicleta de tipo (i,d,n). Por
-- ejemplo,
--   numeroVueltas 2 7 25 == 14

```

```

numeroVueltas :: Int -> Int -> Int -> Int
numeroVueltas i d n = length (takeWhile (/=0) (eslabones i d n))

```

```

-----
-- Ejercicio 14.1. [Basado en el problema 341 del proyecto Euler]. La
-- sucesión de Golomb  $\{G(n)\}$  es una sucesión auto descriptiva: es la
-- única sucesión no decreciente de números naturales tal que el número
--  $n$  aparece  $G(n)$  veces en la sucesión. Los valores de  $G(n)$  para los
-- primeros números son los siguientes:
--   n          1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
--   G(n)       1 2 2 3 3 4 4 4 5 5 5 6 6 6 6 ...

```

```

-- En los apartados de este ejercicio se definirá una función para
-- calcular los términos de la sucesión de Golomb.
--
-- Definir la función
--   golomb :: Int -> Int
-- tal que (golomb n) es el n-ésimo término de la sucesión de Golomb.
-- Por ejemplo,
--   golomb 5 == 3
--   golomb 9 == 5
-- Indicación: Se puede usar la función sucGolomb del apartado 2.
-----

```

```

golomb :: Int -> Int
golomb n = sucGolomb !! (n-1)

```

```

-----
-- Ejercicio 14.2. Definir la función
--   sucGolomb :: [Int]
-- tal que sucGolomb es la lista de los términos de la sucesión de
-- Golomb. Por ejemplo,
--   take 15 sucGolomb == [1,2,2,3,3,4,4,4,5,5,5,6,6,6,6]
-- Indicación: Se puede usar la función subSucGolomb del apartado 3.
-----

```

```

sucGolomb :: [Int]
sucGolomb = subSucGolomb 1

```

```

-----
-- Ejercicio 14.3. Definir la función
--   subSucGolomb :: Int -> [Int]
-- tal que (subSucGolomb x) es la lista de los términos de la sucesión
-- de Golomb a partir de la primera ocurrencia de x. Por ejemplo,
--   take 10 (subSucGolomb 4) == [4,4,4,5,5,5,6,6,6,6]
-- Indicación: Se puede usar la función golomb del apartado 1.
-----

```

```

subSucGolomb :: Int -> [Int]
subSucGolomb 1 = [1] ++ subSucGolomb 2
subSucGolomb 2 = [2,2] ++ subSucGolomb 3
subSucGolomb x = (replicate (golomb x) x) ++ subSucGolomb (x+1)

```

```
-- Nota: La sucesión de Golomb puede definirse de forma más compacta
-- como se muestra a continuación.
sucGolomb' :: [Int]
sucGolomb' = 1 : 2 : 2 : g 3
  where g x      = replicate (golomb x) x ++ g (x+1)
        golomb n = sucGolomb !! (n-1)
```



# Relación 18

## Tipos de datos algebraicos

```
-----  
-- Introducción -----  
-----  
  
-- En esta relación se presenta ejercicios sobre tipos de datos  
-- algebraicos. Se consideran abreviaturas y dos tipos de datos  
-- algebraicos: los números naturales (para los que se define su  
-- producto) y los árboles binarios, para los que se definen funciones  
-- para calcular:  
-- * los puntos más cercanos,  
-- * la ocurrencia de un elemento en el árbol,  
-- * el número de hojas,  
-- * el carácter balanceado de un árbol y  
-- * el árbol balanceado correspondiente a una lista.  
--  
-- Los ejercicios corresponden al tema 9 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-9.pdf  
  
-----  
-- Ejercicio 1. Los puntos del plano se pueden representar por pares de  
-- números como se indica a continuación  
-- type Punto = (Double,Double)  
-- Definir la función  
-- cercanos :: [Punto] -> [Punto] -> (Punto,Punto)  
-- tal que (cercanos ps qs) es un par de puntos, el primero de ps y el  
-- segundo de qs, que son los más cercanos (es decir, no hay otro par
```

```
-- (p',q') con p' en ps y q' en qs tales que la distancia entre p' y q'
-- sea menor que la que hay entre p y q). Por ejemplo,
--   cercanos [(2,5),(3,6)] [(4,3),(1,0),(7,9)] == ((2.0,5.0),(4.0,3.0))
-----
```

```
type Punto = (Double,Double)
```

```
cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
```

```
cercanos ps qs = (p,q)
```

```
  where (d,p,q) = minimum [(distancia p q, p, q) | p <- ps, q <-qs]
          distancia (x,y) (u,v) = sqrt ((x-u)^2+(y-v)^2)
```

```
-----
-- Ejercicio 2.1. En los siguientes ejercicios se usará el tipo
-- algebraico de datos de los números naturales definido por
--   data Nat = Cero | Suc Nat
--           deriving (Eq, Show)
-- Definir la función
--   suma :: Nat -> Nat -> Nat
-- tal que (suma m n) es la suma de los números naturales m y
-- n. Por ejemplo,
--   ghci> suma (Suc (Suc Cero)) (Suc (Suc (Suc Cero)))
--   Suc (Suc (Suc (Suc (Suc Cero))))
-----
```

```
data Nat = Cero | Suc Nat
  deriving (Eq, Show)
```

```
suma :: Nat -> Nat -> Nat
```

```
suma Cero    n = n
```

```
suma (Suc m) n = Suc (suma m n)
```

```
-----
-- Ejercicio 2.2. Definir la función
--   producto :: Nat -> Nat -> Nat
-- tal que (producto m n) es el producto de los números naturales m y
-- n. Por ejemplo,
--   ghci> producto (Suc (Suc Cero)) (Suc (Suc (Suc Cero)))
--   Suc (Suc (Suc (Suc (Suc (Suc Cero)))))
-----
```

```

producto :: Nat -> Nat -> Nat
producto Cero _ = Cero
producto (Suc m) n = suma n (producto m n)

```

-----

-- Ejercicio 3. En los apartados de este ejercicio se trabajará con  
 -- árboles binarios definidos como sigue

```

-- data Arbol = Hoja Int
--             | Nodo Arbol Int Arbol
--             deriving (Show, Eq)

```

-- Por ejemplo, el árbol

```

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

```

-- se representa por

```

--      Nodo (Nodo (Hoja 1) 3 (Hoja 4))
--           5
--           (Nodo (Hoja 6) 7 (Hoja 9))

```

-----

```

data Arbol = Hoja Int
            | Nodo Arbol Int Arbol
            deriving (Show, Eq)

```

```

ejArbol :: Arbol
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
              5
              (Nodo (Hoja 6) 7 (Hoja 9))

```

-----

-- Ejercicio 3.1. Definir la función

```

-- ocurre :: Int -> Arbol -> Bool
-- tal que (ocurre x a) se verifica si x ocurre en el árbol a como valor
-- de un nodo o de una hoja. Por ejemplo,
-- ocurre 4 ejArbol == True
-- ocurre 10 ejArbol == False

```

```

-----
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)      = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d

```

```

-----
-- Ejercicio 3.2. En el prelude está definido el tipo de datos
--   data Ordering = LT | EQ | GT
-- junto con la función
--   compare :: Ord a => a -> a -> Ordering
-- que decide si un valor en un tipo ordenado es menor (LT), igual (EQ)
-- o mayor (GT) que otro.
--
-- Usando esta función, redefinir la función
--   ocurre :: Int -> Arbol -> Bool
-- del ejercicio anterior.
-----

```

```

ocurre' :: Int -> Arbol -> Bool
ocurre' m (Hoja n)      = m == n
ocurre' m (Nodo i n d) = case compare m n of
    LT -> ocurre' m i
    EQ -> True
    GT -> ocurre' m d

```

```

-----
-- Ejercicio 4. En los apartados de este ejercicio se trabajará con
-- árboles binarios definidos como sigue
--   type ArbolB = HojaB Int
--               | NodoB ArbolB ArbolB
--               deriving Show
-- Por ejemplo, el árbol
--
--       .
--      / \
--     /   \
--    .     .
--   / \   / \
--  1 4 6 9
-- se representa por

```



```

--      NodoB (NodoB (HojaB 1) (HojaB 4))
--            (NodoB (HojaB 6) (HojaB 9))
-----

data ArbolB = HojaB Int
             | NodoB ArbolB ArbolB
             deriving Show

ejArbolB :: ArbolB
ejArbolB = NodoB (NodoB (HojaB 1) (HojaB 4))
           (NodoB (HojaB 6) (HojaB 9))

-----

-- Ejercicio 4.1. Definir la función
--   nHojas :: ArbolB -> Int
--   tal que (nHojas a) es el número de hojas del árbol a. Por ejemplo,
--   nHojas (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7))) == 3
--   nHojas ejArbolB == 4
-----

nHojas :: ArbolB -> Int
nHojas (HojaB _)      = 1
nHojas (NodoB a1 a2) = nHojas a1 + nHojas a2

-----

-- Ejercicio 4.2. Se dice que un árbol de este tipo es balanceado si es
-- una hoja o bien si para cada nodo se tiene que el número de hojas en
-- cada uno de sus subárboles difiere como máximo en uno y sus
-- subárboles son balanceados. Definir la función
--   balanceado :: ArbolB -> BoolB
--   tal que (balanceado a) se verifica si a es un árbol balanceado. Por
--   ejemplo,
--   balanceado ejArbolB
--   ==> True
--   balanceado (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7)))
--   ==> True
--   balanceado (NodoB (HojaB 5) (NodoB (HojaB 3) (NodoB (HojaB 5) (HojaB 7))))
--   ==> False
-----

```

```

balanceado :: ArbolB -> Bool
balanceado (HojaB _)      = True
balanceado (NodoB a1 a2) = abs (nHojas a1 - nHojas a2) <= 1 &&
                           balanceado a1 &&
                           balanceado a2

-----

-- Ejercicio 4.3. Definir la función
--   mitades :: [a] -> ([a],[a])
-- tal que (mitades xs) es un par de listas que se obtiene al dividir xs
-- en dos mitades cuya longitud difiere como máximo en uno. Por ejemplo,
--   mitades [2,3,5,1,4,7]    == ([2,3,5],[1,4,7])
--   mitades [2,3,5,1,4,7,9] == ([2,3,5],[1,4,7,9])
-----

mitades :: [a] -> ([a],[a])
mitades xs = splitAt (length xs `div` 2) xs

-----

-- Ejercicio 4.4. Definir la función
--   arbolBalanceado :: [Int] -> ArbolB
-- tal que (arbolBalanceado xs) es el árbol balanceado correspondiente
-- a la lista xs. Por ejemplo,
--   ghci> arbolBalanceado [2,5,3]
--   NodoB (HojaB 2) (NodoB (HojaB 5) (HojaB 3))
--   ghci> arbolBalanceado [2,5,3,7]
--   NodoB (NodoB (HojaB 2) (HojaB 5)) (NodoB (HojaB 3) (HojaB 7))
-----

arbolBalanceado :: [Int] -> ArbolB
arbolBalanceado [x] = HojaB x
arbolBalanceado xs = NodoB (arbolBalanceado ys) (arbolBalanceado zs)
  where (ys,zs) = mitades xs

```

# Relación 19

## Tipos de datos algebraicos: árboles binarios

```
-- -----  
-- Introducción                                     --  
-- -----  
  
-- En esta relación se plantean ejercicios sobre árboles binarios. En  
-- concreto, se definen funciones para calcular:  
-- * el número de hojas de un árbol,  
-- * el número de nodos de un árbol,  
-- * la profundidad de un árbol,  
-- * el recorrido preorden de un árbol,  
-- * el recorrido postorden de un árbol,  
-- * el recorrido preorden de forma iterativa,  
-- * la imagen especular de un árbol,  
-- * el subárbol de profundidad dada,  
-- * el árbol infinito generado con un elemento y  
-- * el árbol de profundidad dada cuyos nodos son iguales a un elemento.  
--  
-- Estos ejercicios corresponden al tema 9 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-9.pdf  
  
-- -----  
-- Importación de librerías auxiliares  
-- -----
```

```
import Data.List
```

```
-- -----
-- Nota. En los siguientes ejercicios se trabajará con los árboles
-- binarios definidos como sigue
```

```
-- data Arbol a = Hoja
--               | Nodo a (Arbol a) (Arbol a)
--               deriving (Show, Eq)
```

```
-- En los ejemplos se usará el siguiente árbol
```

```
-- arbol = Nodo 9
--         (Nodo 3
--          (Nodo 2 Hoja Hoja)
--          (Nodo 4 Hoja Hoja))
--         (Nodo 7 Hoja Hoja)
```

```
data Arbol a = Hoja
              | Nodo a (Arbol a) (Arbol a)
              deriving (Show, Eq)
```

```
arbol = Nodo 9
       (Nodo 3
        (Nodo 2 Hoja Hoja)
        (Nodo 4 Hoja Hoja))
       (Nodo 7 Hoja Hoja)
```

```
-- -----
-- Ejercicio 1. Definir la función
```

```
-- nHojas :: Arbol a -> Int
```

```
-- tal que (nHojas x) es el número de hojas del árbol x. Por ejemplo,
```

```
-- ghci> arbol
```

```
--      Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
```

```
--      ghci> nHojas arbol
```

```
--      6
```

```
nHojas :: Arbol a -> Int
```

```
nHojas Hoja      = 1
```

```
nHojas (Nodo x i d) = nHojas i + nHojas d
```

```

-----
-- Ejercicio 2. Definir la función
--   nNodos :: Arbol a -> Int
-- tal que (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> nNodos arbol
--   5
-----

```

```

nNodos :: Arbol a -> Int
nNodos Hoja      = 0
nNodos (Nodo x i d) = 1 + nNodos i + nNodos d

```

```

-----
-- Ejercicio 3. Definir la función
--   profundidad :: Arbol a -> Int
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> profundidad arbol
--   3
-----

```

```

profundidad :: Arbol a -> Int
profundidad Hoja = 0
profundidad (Nodo x i d) = 1 + max (profundidad i) (profundidad d)

```

```

-----
-- Ejercicio 4. Definir la función
--   preorden :: Arbol a -> [a]
-- tal que (preorden x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> preorden arbol
--   [9,3,2,4,7]
-----

```

```
preorden :: Arbol a -> [a]
preorden Hoja      = []
preorden (Nodo x i d) = x : (preorden i ++ preorden d)
```

```
-----
-- Ejercicio 5. Definir la función
--   postorden :: Arbol a -> [a]
-- tal que (postorden x) es la lista correspondiente al recorrido
-- postorden del árbol x; es decir, primero recorre el subárbol
-- izquierdo, a continuación el subárbol derecho y, finalmente, la raíz
-- del árbol. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> postorden arbol
--   [2,4,3,7,9]
-----
```

```
postorden :: Arbol a -> [a]
postorden Hoja      = []
postorden (Nodo x i d) = postorden i ++ postorden d ++ [x]
```

```
-----
-- Ejercicio 6. Definir, usando un acumulador, la función
--   preordenIt :: Arbol a -> [a]
-- tal que (preordenIt x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> preordenIt arbol
--   [9,3,2,4,7]
-- Nota: No usar (++) en la definición
-----
```

```
preordenIt :: Arbol a -> [a]
preordenIt x = preordenItAux x []
  where preordenItAux Hoja xs      = xs
        preordenItAux (Nodo x i d) xs =
```

```
x : preordenItAux i (preordenItAux d xs)
```

```
-----
-- Ejercicio 7. Definir la función
--   espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--   ghci> espejo arbol
--   Nodo 9
--       (Nodo 7 Hoja Hoja)
--       (Nodo 3
--         (Nodo 4 Hoja Hoja)
--         (Nodo 2 Hoja Hoja))
-----
```

```
espejo :: Arbol a -> Arbol a
espejo Hoja      = Hoja
espejo (Nodo x i d) = Nodo x (espejo d) (espejo i)
```

```
-----
-- Ejercicio 8. La función take está definida por
--   take :: Int -> [a] -> [a]
--   take 0          = []
--   take (n+1) []   = []
--   take (n+1) (x:xs) = x : take n xs
-- Definir la función
--   takeArbol :: Int -> Arbol a -> Arbol a
-- tal que (takeArbol n t) es el subárbol de t de profundidad n. Por
-- ejemplo,
--   ghci> takeArbol 0 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--   Hoja
--   ghci> takeArbol 1 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--   Nodo 6 Hoja Hoja
--   ghci> takeArbol 2 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--   Nodo 6 Hoja (Nodo 7 Hoja Hoja)
--   ghci> takeArbol 3 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--   Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja)
--   ghci> takeArbol 4 (Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja))
--   Nodo 6 Hoja (Nodo 7 (Nodo 5 Hoja Hoja) Hoja)
-----
```

```

takeArbol :: Int -> Arbol a -> Arbol a
takeArbol 0 _ = Hoja
takeArbol _ Hoja = Hoja
takeArbol n (Nodo x i d) =
    Nodo x (takeArbol (n-1) i) (takeArbol (n-1) d)

```

```

-----
-- Ejercicio 9. La función
--   repeat :: a -> [a]
-- está definida de forma que (repeat x) es la lista formada por
-- infinitos elementos x. Por ejemplo,
--   repeat 3 == [3,3,3,3,3,3,3,3,3,3,3,3,3,3,...]
-- La definición de repeat es
--   repeat x = xs where xs = x:xs
-- Definir la función
--   repeatArbol :: a -> Arbol a
-- tal que (repeatArbol x) es es árbol con infinitos nodos x. Por
-- ejemplo,
--   ghci> takeArbol 0 (repeatArbol 3)
--   Hoja
--   ghci> takeArbol 1 (repeatArbol 3)
--   Nodo 3 Hoja Hoja
--   ghci> takeArbol 2 (repeatArbol 3)
--   Nodo 3 (Nodo 3 Hoja Hoja) (Nodo 3 Hoja Hoja)
-----

```

```

repeatArbol :: a -> Arbol a
repeatArbol x = Nodo x t t
    where t = repeatArbol x

```

```

-----
-- Ejercicio 10. La función
--   replicate :: Int -> a -> [a]
-- está definida por
--   replicate n = take n . repeat
-- es tal que (replicate n x) es la lista de longitud n cuyos elementos
-- son x. Por ejemplo,
--   replicate 3 5 == [5,5,5]
-- Definir la función
--   replicateArbol :: Int -> a -> Arbol a

```



```
-- tal que (replicate n x) es el árbol de profundidad n cuyos nodos son
-- x. Por ejemplo,
-- ghci> replicateArbol 0 5
-- Hoja
-- ghci> replicateArbol 1 5
-- Nodo 5 Hoja Hoja
-- ghci> replicateArbol 2 5
-- Nodo 5 (Nodo 5 Hoja Hoja) (Nodo 5 Hoja Hoja)
```

```
-----
replicateArbol :: Int -> a -> Arbol a
replicateArbol n = takeArbol n . repeatArbol
```



# Relación 20

## Tipos de datos: Modelización de juego de cartas

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se estudia la modelización de un juego de cartas  
-- como aplicación de los tipos de datos algebraicos. Además, se definen  
-- los generadores correspondientes para comprobar las propiedades con  
-- QuickCheck.  
--  
-- Estos ejercicios corresponden al tema 9 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-9.pdf  
  
-- -----  
-- Importación de librerías auxiliares --  
-- -----  
  
import Test.QuickCheck  
import Data.Char  
import Data.List  
  
-- -----  
-- Ejercicio resuelto. Definir el tipo de datos Palo para representar  
-- los cuatro palos de la baraja: picas, corazones, diamantes y  
-- tréboles. Hacer que Palo sea instancia de Eq y Show.
```

-----  
*-- La definición es*

```
data Palo = Picas | Corazones | Diamantes | Treboles
           deriving (Eq, Show)
```

-----  
*-- Nota: Para que QuickCheck pueda generar elementos del tipo Palo se  
-- usa la siguiente función.*  
-----

```
instance Arbitrary Palo where
  arbitrary = elements [Picas, Corazones, Diamantes, Treboles]
```

-----  
*-- Ejercicio resuelto. Definir el tipo de dato Color para representar los  
-- colores de las cartas: rojo y negro. Hacer que Color sea instancia de  
-- Show.*  
-----

```
data Color = Rojo | Negro
           deriving Show
```

-----  
*-- Ejercicio 1. Definir la función  
-- color :: Palo -> Color  
-- tal que (color p) es el color del palo p. Por ejemplo,  
-- color Corazones ==> Rojo  
-- Nota: Los corazones y los diamantes son rojos. Las picas y los  
-- tréboles son negros.*  
-----

```
color :: Palo -> Color
color Picas      = Negro
color Corazones = Rojo
color Diamantes = Rojo
color Treboles  = Negro
```

-----  
*-- Ejercicio resuelto. Los valores de las cartas se dividen en los*

```

-- numéricos (del 2 al 10) y las figuras (sota, reina, rey y
-- as). Definir el tipo -- de datos Valor para representar los valores
-- de las cartas. Hacer que Valor sea instancia de Eq y Show.
-- Main> :type Sota
-- Sota :: Valor
-- Main> :type Reina
-- Reina :: Valor
-- Main> :type Rey
-- Rey :: Valor
-- Main> :type As
-- As :: Valor
-- Main> :type Numerico 3
-- Numerico 3 :: Valor

```

```

data Valor = Numerico Int | Sota | Reina | Rey | As
deriving (Eq, Show)

```

```

-- Nota: Para que QuickCheck pueda generar elementos del tipo Valor se
-- usa la siguiente función.

```

```

instance Arbitrary Valor where
  arbitrary =
    oneof $
      [ do return c
        | c <- [Sota, Reina, Rey, As]
        ] ++
      [ do n <- choose (2,10)
        return (Numerico n)
        ]

```

```

-- Ejercicio 2. El orden de valor de las cartas (de mayor a menor) es
-- as, rey, reina, sota y las numéricas según su valor. Definir la
-- función
-- mayor :: Valor -> Valor -> Bool
-- tal que (mayor x y) se verifica si la carta x es de mayor valor que
-- la carta y. Por ejemplo,

```

```

-- mayor Sota (Numerico 7) ==> True
-- mayor (Numerico 10) Reina ==> False
-----

mayor :: Valor -> Valor -> Bool
mayor _ As = False
mayor As _ = True
mayor _ Rey = False
mayor Rey _ = True
mayor _ Reina = False
mayor Reina _ = True
mayor _ Sota = False
mayor Sota _ = True
mayor (Numerico m) (Numerico n) = m > n

-----

-- Ejercicio 3. Comprobar con QuickCheck si dadas dos cartas, una
-- siempre tiene mayor valor que la otra. En caso de que no se verifique,
-- añadir la menor precondición para que lo haga.
-----

-- La propiedad es
prop_MayorValor1 a b =
  mayor a b || mayor b a

-- La comprobación es
-- Main> quickCheck prop_MayorValor1
-- Falsifiable, after 2 tests:
-- Sota
-- Sota
-- que indica que la propiedad es falsa porque la sota no tiene mayor
-- valor que la sota.

-- La precondición es que las cartas sean distintas:
prop_MayorValor a b =
  a /= b ==> mayor a b || mayor b a

-- La comprobación es
-- Main> quickCheck prop_MayorValor
-- OK, passed 100 tests.

```

```
-----  
-- Ejercicio resuelto. Definir el tipo de datos Carta para representar  
-- las cartas mediante un valor y un palo. Hacer que Carta sea instancia  
-- de Eq y Show. Por ejemplo,  
--   Main> :type Carta Rey Corazones  
--   Carta Rey Corazones :: Carta  
--   Main> :type Carta (Numerico 4) Corazones  
--   Carta (Numerico 4) Corazones :: Carta  
-----
```

```
data Carta = Carta Valor Palo  
      deriving (Eq, Show)
```

```
-----  
-- Ejercicio 4. Definir la función  
--   valor :: Carta -> Valor  
-- tal que (valor c) es el valor de la carta c. Por ejemplo,  
--   valor (Carta Rey Corazones) ==> Rey  
-----
```

```
valor :: Carta -> Valor  
valor (Carta v p) = v
```

```
-----  
-- Ejercicio 5. Definir la función  
--   palo :: Carta -> Valor  
-- tal que (palo c) es el palo de la carta c. Por ejemplo,  
--   palo (Carta Rey Corazones) ==> Corazones  
-----
```

```
palo :: Carta -> Palo  
palo (Carta v p) = p
```

```
-----  
-- Nota: Para que QuickCheck pueda generar elementos del tipo Carta se  
-- usa la siguiente función.  
-----
```

```
instance Arbitrary Carta where
```

```

arbitrary =
  do v <- arbitrary
     p <- arbitrary
     return (Carta v p)
-----
-- Ejercicio 6. Definir la función
--   ganaCarta :: Palo -> Carta -> Carta -> Bool
-- tal que (ganaCarta p c1 c2) se verifica si la carta c1 le gana a la
-- carta c2 cuando el palo de triunfo es p (es decir, las cartas son del
-- mismo palo y el valor de c1 es mayor que el de c2 o c1 es del palo de
-- triunfo). Por ejemplo,
--   ganaCarta Corazones (Carta Sota Picas) (Carta (Numerico 5) Picas)
--   ==> True
--   ganaCarta Corazones (Carta (Numerico 3) Picas) (Carta Sota Picas)
--   ==> False
--   ganaCarta Corazones (Carta (Numerico 3) Corazones) (Carta Sota Picas)
--   ==> True
--   ganaCarta Treboles (Carta (Numerico 3) Corazones) (Carta Sota Picas)
--   ==> False
-----

ganaCarta :: Palo -> Carta -> Carta -> Bool
ganaCarta triunfo c c'
  | palo c == palo c' = mayor (valor c) (valor c')
  | palo c == triunfo = True
  | otherwise         = False
-----

-- Ejercicio 7. Comprobar con QuickCheck si dadas dos cartas, una
-- siempre gana a la otra.
-----

-- La propiedad es
prop_GanaCarta t c1 c2 =
  ganaCarta t c1 c2 || ganaCarta t c2 c1

-- La comprobación es
--   Main> quickCheck prop_GanaCarta
--   Falsifiable, after 0 tests:

```



```

--   Diamantes
--   Carta Rey Corazones
--   Carta As Treboles
-- que indica que la propiedad no se verifica ya que cuando el triunfo
-- es diamantes, ni el rey de corazones le gana al as de tréboles ni el
-- as de tréboles le gana al rey de corazones.

-----
-- Ejercicio resuelto. Definir el tipo de datos Mano para representar
-- una mano en el juego de cartas. Una mano es vacía o se obtiene
-- agregando una carta a una mano. Hacer Mano instancia de Eq y
-- Show. Por ejemplo,
--   Main> :type Agrega (Carta Rey Corazones) Vacía
--   Agrega (Carta Rey Corazones) Vacía :: Mano
-----

data Mano = Vacía | Agrega Carta Mano
deriving (Eq, Show)

-----
-- Nota: Para que QuickCheck pueda generar elementos del tipo Mano se
-- usa la siguiente función.
-----

instance Arbitrary Mano where
  arbitrary =
    do cs <- arbitrary
    let mano [] = Vacía
        mano (c:cs) = Agrega c (mano cs)
    return (mano cs)

-----
-- Ejercicio 8. Una mano gana a una carta c si alguna carta de la mano
-- le gana a c. Definir la función
--   ganaMano :: Palo -> Mano -> Carta -> Bool
-- tal que (gana t m c) se verifica si la mano m le gana a la carta c
-- cuando el triunfo es t. Por ejemplo,
--   ganaMano Picas (Agrega (Carta Sota Picas) Vacía) (Carta Rey Corazones)
--   ==> True
--   ganaMano Picas (Agrega (Carta Sota Picas) Vacía) (Carta Rey Picas)

```

```

-- ==> False
-----

ganaMano :: Palo -> Mano -> Carta -> Bool
ganaMano triunfo Vacia      c' = False
ganaMano triunfo (Agrega c m) c' = ganaCarta triunfo c c' ||
                                   ganaMano triunfo m c'

-----

-- Ejercicio 9. Definir la función
--   eligeCarta :: Palo -> Carta -> Mano -> Carta
-- tal que (eligeCarta t c1 m) es la mejor carta de la mano m frente a
-- la carta c cuando el triunfo es t. La estrategia para elegir la mejor
-- carta es la siguiente:
-- * Si la mano sólo tiene una carta, se elige dicha carta.
-- * Si la primera carta de la mano es del palo de c1 y la mejor del
--   resto no es del palo de c1, se elige la primera de la mano,
-- * Si la primera carta de la mano no es del palo de c1 y la mejor
--   del resto es del palo de c1, se elige la mejor del resto.
-- * Si la primera carta de la mano le gana a c1 y la mejor del
--   resto no le gana a c1, se elige la primera de la mano,
-- * Si la mejor del resto le gana a c1 y la primera carta de la mano
--   no le gana a c1, se elige la mejor del resto.
-- * Si el valor de la primera carta es mayor que el de la mejor del
--   resto, se elige la mejor del resto.
-- * Si el valor de la primera carta no es mayor que el de la mejor
--   del resto, se elige la primera carta.
-----

eligeCarta :: Palo -> Carta -> Mano -> Carta
eligeCarta triunfo c1 (Agrega c Vacia) = c           -- 1
eligeCarta triunfo c1 (Agrega c resto)
  | palo c == palo c1 && palo c' /= palo c1          = c   -- 2
  | palo c /= palo c1 && palo c' == palo c1          = c'  -- 3
  | ganaCarta triunfo c c1 && not (ganaCarta triunfo c' c1) = c   -- 4
  | ganaCarta triunfo c' c1 && not (ganaCarta triunfo c c1) = c'  -- 5
  | mayor (valor c) (valor c')                       = c'  -- 6
  | otherwise                                          = c   -- 7
where
  c' = eligeCarta triunfo c1 resto

```

```
-----  
-- Ejercicio 10. Comprobar con QuickCheck que si una mano es ganadora,  
-- entonces la carta elegida es ganadora.  
-----  
  
-- La propiedad es  
prop_eligeCartaGanaSiEsPosible triunfo c m =  
  m /= Vacia ==>  
  ganaMano triunfo m c == ganaCarta triunfo (eligeCarta triunfo c m) c  
  
-- La comprobación es  
-- Main> quickCheck prop_eligeCartaGanaSiEsPosible  
-- Falsificable, after 12 tests:  
-- Corazones  
-- Carta Rey Treboles  
-- Agrega (Carta (Numerico 6) Diamantes)  
--       (Agrega (Carta Sota Picas)  
--       (Agrega (Carta Rey Corazones)  
--       (Agrega (Carta (Numerico 10) Treboles)  
--       Vacia)))  
-- La carta elegida es el 10 de tréboles (porque tiene que ser del mismo  
-- palo), aunque el mano hay una carta (el rey de corazones) que gana.
```



# Relación 21

## Resolución de problemas matemáticos

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se plantea la resolución de distintos problemas  
-- matemáticos. En concreto,  
-- * el problema de Ullman sobre la existencia de subconjunto del tamaño  
-- dado y con su suma acotada,  
-- * las descomposiciones de un número como suma de dos cuadrados,  
-- * el problema 145 del proyecto Euler,  
-- * el grafo de una función sobre los elementos que cumplen una  
-- propiedad,  
-- * los números semiperfectos,  
-- * el carácter funcional de una relación,  
-- * la identidad de Bezout.  
--  
-- Además, de los 2 primeros se presentan distintas definiciones y se  
-- compara su eficiencia.  
--  
-- Estos ejercicios corresponden a los temas 5, 6 y 7 cuyas  
-- transparencias se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-5.pdf  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-6.pdf  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-7.pdf
```

```

-----
-- Importación de librerías auxiliares
-----

import Test.QuickCheck
import Data.List

-----
-- Ejercicio 1. Definir la función
--   ullman :: (Num a, Ord a) => a -> Int -> [a] -> Bool
-- tal que (ullman t k xs) se verifica si xs tiene un subconjunto con k
-- elementos cuya suma sea menor que t. Por ejemplo,
--   ullman 9 3 [1..10] == True
--   ullman 5 3 [1..10] == False
-----

-- 1ª solución (corta y eficiente)
ullman :: (Ord a, Num a) => a -> Int -> [a] -> Bool
ullman t k xs = sum (take k (sort xs)) < t

-- 2ª solución (larga e ineficiente)
ullman2 :: (Num a, Ord a) => a -> Int -> [a] -> Bool
ullman2 t k xs =
  [ys | ys <- subconjuntos xs, length ys == k, sum ys < t] /= []

-- (subconjuntos xs) es la lista de los subconjuntos de xs. Por
-- ejemplo,
--   subconjuntos "bc" == [ "", "c", "b", "bc" ]
--   subconjuntos "abc" == [ "", "c", "b", "bc", "a", "ac", "ab", "abc" ]
subconjuntos :: [a] -> [[a]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = zss++[x:ys | ys <- zss]
  where zss = subconjuntos xs

-- Los siguientes ejemplos muestran la diferencia en la eficiencia:
--   *Main> ullman 9 3 [1..20]
--   True
--   (0.02 secs, 528380 bytes)
--   *Main> ullman2 9 3 [1..20]
--   True

```

```
-- (4.08 secs, 135267904 bytes)
-- *Main> ullman 9 3 [1..100]
-- True
-- (0.02 secs, 526360 bytes)
-- *Main> ullman2 9 3 [1..100]
-- C-c C-cInterrupted.
-- Agotado
```

-----

```
-- Ejercicio 2. Definir la función
```

```
-- sumasDe2Cuadrados :: Integer -> [(Integer, Integer)]
-- tal que (sumasDe2Cuadrados n) es la lista de los pares de números
-- tales que la suma de sus cuadrados es n y el primer elemento del par
-- es mayor o igual que el segundo. Por ejemplo,
-- sumasDe2Cuadrados 25 == [(5,0),(4,3)]
```

-----

```
-- Primera definición:
```

```
sumasDe2Cuadrados_1 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_1 n =
  [(x,y) | x <- [n,n-1..0],
           y <- [0..x],
           x*x+y*y == n]
```

```
-- Segunda definición:
```

```
sumasDe2Cuadrados_2 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_2 n =
  [(x,y) | x <- [a,a-1..0],
           y <- [0..x],
           x*x+y*y == n]
  where a = ceiling (sqrt (fromIntegral n))
```

```
-- Tercera definición:
```

```
sumasDe2Cuadrados_3 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_3 n = aux (ceiling (sqrt (fromIntegral n))) 0 where
  aux x y | x < y           = []
          | x*x + y*y < n = aux x (y+1)
          | x*x + y*y == n = (x,y) : aux (x-1) (y+1)
          | otherwise      = aux (x-1) y
```

```
-- Comparación
```

```
-- +-----+-----+-----+-----+
-- | n          | 1ª definición | 2ª definición | 3ª definición |
-- +-----+-----+-----+-----+
-- |      999 | 2.17 segs     | 0.02 segs     | 0.01 segs     |
-- | 48612265 |                | 140.38 segs   | 0.13 segs     |
-- +-----+-----+-----+-----+
```

```
-- -----
-- Ejercicio 3. (Basado en el problema 145 del Proyecto Euler). Se dice
-- que un número  $n$  es reversible si su última cifra es distinta de 0 y
-- la suma de  $n$  y el número obtenido escribiendo las cifras de  $n$  en
-- orden inverso es un número que tiene todas sus cifras impares. Por
-- ejemplo, 36 es reversible porque  $36+63=99$  tiene todas sus cifras
-- impares, 409 es reversible porque  $409+904=1313$  tiene todas sus cifras
-- impares, 243 no es reversible porque  $243+342=585$  no tiene todas sus
-- cifras impares.
```

```
-- Definir la función
```

```
-- reversiblesMenores :: Int -> Int
```

```
-- tal que (reversiblesMenores  $n$ ) es la cantidad de números reversibles
-- menores que  $n$ . Por ejemplo,
```

```
-- reversiblesMenores 10 == 0
```

```
-- reversiblesMenores 100 == 20
```

```
-- reversiblesMenores 1000 == 120
```

```
reversiblesMenores :: Int -> Int
```

```
reversiblesMenores n = length [x | x <- [1..n-1], esReversible x]
```

```
-- (esReversible  $n$ ) se verifica si  $n$  es reversible; es decir, si su
-- última cifra es distinta de 0 y la suma de  $n$  y el número obtenido
-- escribiendo las cifras de  $n$  en orden inverso es un número que tiene
-- todas sus cifras impares. Por ejemplo,
```

```
-- esReversible 36 == True
```

```
-- esReversible 409 == True
```

```
esReversible :: Int -> Bool
```

```
esReversible n = rem n 10 /= 0 && impares (cifras (n + (inverso n)))
```

```
-- (impares  $xs$ ) se verifica si  $xs$  es una lista de números impares. Por
-- ejemplo,
```



```

--      impares [3,5,1] == True
--      impares [3,4,1] == False
impares :: [Int] -> Bool
impares xs = and [odd x | x <- xs]

-- (inverso n) es el número obtenido escribiendo las cifras de n en
-- orden inverso. Por ejemplo,
--      inverso 3034 == 4303
inverso :: Int -> Int
inverso n = read (reverse (show n))

-- (cifras n) es la lista de las cifras del número n. Por ejemplo,
--      cifras 3034 == [3,0,3,4]
cifras :: Int -> [Int]
cifras n = [read [x] | x <- show n]

-----
-- Ejercicio 4. Definir, usando funciones de orden superior, la función
--      grafoReducido :: Eq a => (a -> b) -> (a -> Bool) -> [a] -> [(a,b)]
-- tal que (grafoReducido f p xs) es la lista (sin repeticiones) de los
-- pares formados por los elementos de xs que verifican el predicado p
-- y sus imágenes. Por ejemplo,
--      grafoReducido (^2) even [1..9] == [(2,4),(4,16),(6,36),(8,64)]
--      grafoReducido (+4) even (replicate 40 1) == []
--      grafoReducido (*5) even (replicate 40 2) == [(2,10)]
-----

grafoReducido :: Eq a => (a -> b) -> (a -> Bool) -> [a] -> [(a,b)]
grafoReducido f p xs = [(x,f x) | x <- nub xs, p x]

-----
-- Ejercicio 5.1. Un número natural n se denomina semiperfecto si es la
-- suma de algunos de sus divisores propios. Por ejemplo, 18 es
-- semiperfecto ya que sus divisores son 1, 2, 3, 6, 9 y se cumple que
-- 3+6+9=18.
--
-- Definir la función
--      esSemiPerfecto :: Int -> Bool
-- tal que (esSemiPerfecto n) se verifica si n es semiperfecto. Por
-- ejemplo,

```

```

--     esSemiPerfecto 18 == True
--     esSemiPerfecto 9  == False
--     esSemiPerfecto 24 == True
-----

esSemiPerfecto :: Int -> Bool
esSemiPerfecto n =
    or [sum ys == n | ys <- subconjuntos (divisores n)]

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

-----

-- Ejercicio 5.2. Definir la constante primerSemiPerfecto tal que su
-- valor es el primer número semiperfecto.
-----

primerSemiPerfecto :: Int
primerSemiPerfecto = head [n | n <- [1..], esSemiPerfecto n]

-- La evaluación es
--     *Main> primerSemiPerfecto
--     6
-----

-- Ejercicio 5.3. Definir la función
--     semiPerfecto :: Int -> Int
-- tal que (semiPerfecto n) es el n-ésimo número semiperfecto. Por
-- ejemplo,
--     semiPerfecto 1    == 6
--     semiPerfecto 4    == 20
--     semiPerfecto 100 == 414
-----

semiPerfecto :: Int -> Int
semiPerfecto n = semiPerfectos !! n

-- semiPerfectos es la lista de los números semiPerfectos. Por ejemplo,

```

```

--      take 4 semiPerfectos == [6,12,18,20]
semiPerfectos = [n | n <- [1..], esSemiPerfecto n]

-----
-- Ejercicio 6. Las relaciones finitas se pueden representar mediante
-- listas de pares. Por ejemplo,
--      r1, r2, r3 :: [(Int, Int)]
--      r1 = [(1,3), (2,6), (8,9), (2,7)]
--      r2 = [(1,3), (2,6), (8,9), (3,7)]
--      r3 = [(1,3), (2,6), (8,9), (3,6)]
-- Definir la función
--      esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
-- tal que (esFuncion r) se verifica si la relación r es una función (es
-- decir, a cada elemento del dominio de la relación r le corresponde un
-- único elemento). Por ejemplo,
--      esFuncion r1 == False
--      esFuncion r2 == True
--      esFuncion r3 == True
-----

r1, r2, r3 :: [(Int, Int)]
r1 = [(1,3), (2,6), (8,9), (2,7)]
r2 = [(1,3), (2,6), (8,9), (3,7)]
r3 = [(1,3), (2,6), (8,9), (3,6)]

esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
esFuncion [] = True
esFuncion ((x,y):r) =
    [y' | (x',y') <- r, x == x', y /= y'] == [] && esFuncion r

-----
-- Ejercicio 7.1. [La identidad de Bezout] Definir la función
--      bezout :: Integer -> Integer -> (Integer, Integer)
-- tal que (bezout a b) es un par de números x e y tal que a*x+b*y es el
-- máximo común divisor de a y b. Por ejemplo,
--      bezout 21 15 == (-2,3)
-- Indicación: Se puede usar la función quotRem tal que (quotRem x y) es
-- el par formado por el cociente y el resto de dividir x entre y.
-----

```

```

-- Ejemplo de cálculo
--   a  b  q  r
--   36 21  1 15  (1)
--   21 15  1  6  (2)
--   15  6  2  3  (3)
--   6  3  2  0
--   3  0
-- Por tanto,
--   3 = 15 - 6*2           [por (3)]
--     = 15 - (21-15*1)*2   [por (2)]
--     = 21*(-2) + 15*3
--     = 21*(-2)+ (36-21*1)*3 [por (1)]
--     = 36*3 + 21*(-5)

-- Sean p, r el cociente y el resto de a entre b, d el máximo común
-- divisor de a y b y (x,y) el valor de (bezout b r) . Entonces,
--   a = bp+r
--   d = bx+ry
-- ya que  $\text{mcd}(a,b) = \text{mcd}(b,r)$ . Por tanto,
--   d = bx + (a-bp)y
--     = ay + b(x-qy)
-- Luego,
--   bezout a b = (y,x-qy)

bezout :: Integer -> Integer -> (Integer, Integer)
bezout _ 0 = (1,0)
bezout _ 1 = (0,1)
bezout a b = (y, x-q*y)
  where (q,r) = quotRem a b
        (x,y) = bezout b r

-----
-- Ejercicio 7.2. Comprobar con QuickCheck que si  $a>0$ ,  $b>0$  y
--  $(x,y)$  es el valor de (bezout a b), entonces  $a*x+b*y$  es igual al
-- máximo común divisor de a y b.
-----

-- La propiedad es
prop_Bezout :: Integer -> Integer -> Property
prop_Bezout a b = a>0 && b>0 ==> a*x+b*y == gcd a b

```

```
where (x,y) = bezout a b
```

```
-- La comprobación es  
-- Main> quickCheck prop_Bezout  
-- OK, passed 100 tests.
```



# Relación 22

## Combinatoria

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación es estudiar la generación y el número de  
-- las principales operaciones de la combinatoria. En concreto, se  
-- estudia  
-- * Permutaciones.  
-- * Combinaciones sin repetición..  
-- * Combinaciones con repetición  
-- * Variaciones sin repetición.  
-- * Variaciones con repetición.  
-- Además, se estudia dos temas relacionados:  
-- * Reconocimiento y generación de subconjuntos y  
-- * El triángulo de Pascal  
  
-----  
-- Importación de librerías --  
-----  
  
import Test.QuickCheck  
import Data.List (genericLength)  
  
-----  
-- § Subconjuntos  
-----
```

```

-----
-- Ejercicio 1. Definir, por recursión, la función
--   subconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
--   subconjunto [1,3,2,3] [1,2,3] == True
--   subconjunto [1,3,4,3] [1,2,3] == False
-----

```

```

subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto [] _ = True
subconjunto (x:xs) ys = elem x ys && subconjunto xs ys

```

```

-----
-- Ejercicio 2. Definir, mediante all, la función
--   subconjunto' :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto' xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
--   subconjunto' [1,3,2,3] [1,2,3] == True
--   subconjunto' [1,3,4,3] [1,2,3] == False
-----

```

```

subconjunto' :: Eq a => [a] -> [a] -> Bool
subconjunto' xs ys = all ('elem' ys) xs

```

```

-----
-- Ejercicio 3. Comprobar con QuickCheck que las funciones subconjunto
-- y subconjunto' son equivalentes.
-----

```

```

-- La propiedad es
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
  subconjunto xs ys == subconjunto' xs ys

```

```

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

```



```
-- Ejercicio 4. Definir la función
-- igualConjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (igualConjunto xs ys) se verifica si las listas xs e ys,
-- vistas como conjuntos, son iguales. Por ejemplo,
-- igualConjunto [1..10] [10,9..1] == True
-- igualConjunto [1..10] [11,10..1] == False
```

```
-----
igualConjunto :: Eq a => [a] -> [a] -> Bool
igualConjunto xs ys = subconjunto xs ys && subconjunto ys xs
```

```
-----
-- Ejercicio 5. Definir la función
-- subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de los subconjuntos de la lista
-- xs. Por ejemplo,
-- ghci> subconjuntos [2,3,4]
-- [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
-- ghci> subconjuntos [1,2,3,4]
-- [[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
-- [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
```

```
-----
subconjuntos :: [a] -> [[a]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = [x:ys | ys <- sub] ++ sub
  where sub = subconjuntos xs
```

```
-- Cambiando la comprensión por map se obtiene
subconjuntos' :: [a] -> [[a]]
subconjuntos' [] = [[]]
subconjuntos' (x:xs) = sub ++ map (x:) sub
  where sub = subconjuntos' xs
```

```
-----
-- § Permutaciones
-----
```

```
-----
-- Ejercicio 6. Definir la función
```

```
--   intercala :: a -> [a] -> [[a]]
--   tal que (intercala x ys) es la lista de las listas obtenidas
--   intercalando x entre los elementos de ys. Por ejemplo,
--   intercala 1 [2,3] == [[1,2,3],[2,1,3],[2,3,1]]
-----
```

```
-- Una definición recursiva es
intercala1 :: a -> [a] -> [[a]]
intercala1 x [] = [[x]]
intercala1 x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala1 x ys]
```

```
-- Otra definición, más eficiente, es
intercala :: a -> [a] -> [[a]]
intercala y xs =
  [take n xs ++ (y : drop n xs) | n <- [0..length xs]]
-----
```

```
-- Ejercicio 7. Definir la función
--   permutaciones :: [a] -> [[a]]
--   tal que (permutaciones xs) es la lista de las permutaciones de la
--   lista xs. Por ejemplo,
--   permutaciones "bc" == ["bc","cb"]
--   permutaciones "abc" == ["abc","bac","bca","acb","cab","cba"]
-----
```

```
permutaciones :: [a] -> [[a]]
permutaciones [] = [[]]
permutaciones (x:xs) =
  concat [intercala x ys | ys <- permutaciones xs]
```

```
-- Ejercicio 8. Definir la función
--   permutacionesN :: Integer -> [[Integer]]
--   tal que (permutacionesN n) es la lista de las permutaciones de los n
--   primeros números. Por ejemplo,
--   ghci> permutacionesN 3
--   [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
-----
```

```
permutacionesN :: Integer -> [[Integer]]
```

```
permutacionesN n = permutaciones [1..n]
```

```
-----  
-- Ejercicio 9. Definir, usando permutacionesN, la función  
--   numeroPermutacionesN :: Integer -> Integer  
-- tal que (numeroPermutacionesN n) es el número de permutaciones de un  
-- conjunto con n elementos. Por ejemplo,  
--   numeroPermutacionesN 3 == 6  
--   numeroPermutacionesN 4 == 24  
-----
```

```
numeroPermutacionesN :: Integer -> Integer  
numeroPermutacionesN = genericLength . permutacionesN
```

```
-----  
-- Ejercicio 10. Definir la función  
--   fact :: Integer -> Integer  
-- tal que (fact n) es el factorial de n. Por ejemplo,  
--   fact 3 == 6  
-----
```

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

```
-----  
-- Ejercicio 11. Definir, usando fact, la función  
--   numeroPermutacionesN' :: Integer -> Integer  
-- tal que (numeroPermutacionesN' n) es el número de permutaciones de un  
-- conjunto con n elementos. Por ejemplo,  
--   numeroPermutacionesN' 3 == 6  
--   numeroPermutacionesN' 4 == 24  
-----
```

```
numeroPermutacionesN' :: Integer -> Integer  
numeroPermutacionesN' = fact
```

```
-----  
-- Ejercicio 12. Definir la función  
--   prop_numeroPermutacionesN :: Integer -> Bool  
-- tal que (prop_numeroPermutacionesN n) se verifica si las funciones
```

```

-- numeroPermutacionesN y numeroPermutacionesN' son equivalentes para
-- los n primeros números. Por ejemplo,
--   prop_numeroPermutacionesN 5 == True
-----

prop_numeroPermutacionesN :: Integer -> Bool
prop_numeroPermutacionesN n =
  and [numeroPermutacionesN x == numeroPermutacionesN' x | x <- [1..n]]
-----

-- § Combinaciones
-----

-- Ejercicio 13. Definir la función
--   combinaciones :: Integer -> [a] -> [[a]]
-- tal que (combinaciones k xs) es la lista de las combinaciones de
-- orden k de los elementos de la lista xs. Por ejemplo,
--   ghci> combinaciones 2 "bcde"
--   ["bc","bd","be","cd","ce","de"]
--   ghci> combinaciones 3 "bcde"
--   ["bcd","bce","bde","cde"]
--   ghci> combinaciones 3 "abcde"
--   ["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
-----

-- 1ª definición
combinaciones_1 :: Integer -> [a] -> [[a]]
combinaciones_1 n xs =
  [ys | ys <- subconjuntos xs, genericLength ys == n]

-- 2ª definición
combinaciones_2 :: Integer -> [a] -> [[a]]
combinaciones_2 0 _      = [[]]
combinaciones_2 _ []     = []
combinaciones_2 k (x:xs) =
  [x:ys | ys <- combinaciones_2 (k-1) xs] ++ combinaciones_2 k xs

-- La anterior definición se puede escribir usando map:
combinaciones_3 :: Integer -> [a] -> [[a]]

```

```

combinaciones_3 0 _ = [[]]
combinaciones_3 _ [] = []
combinaciones_3 (k+1) (x:xs) =
    map (x:) (combinaciones_3 k xs) ++ combinaciones_3 (k+1) xs

-- Nota. La segunda definición es más eficiente como se comprueba en la
-- siguiente sesión
-- ghci> :set +s
-- ghci> length (combinaciones_1 2 [1..15])
-- 105
-- (0.19 secs, 6373848 bytes)
-- ghci> length (combinaciones_2 2 [1..15])
-- 105
-- (0.01 secs, 525360 bytes)
-- ghci> length (combinaciones_3 2 [1..15])
-- 105
-- (0.02 secs, 528808 bytes)

-- En lo que sigue, usaremos combinaciones como combinaciones_2
combinaciones :: Integer -> [a] -> [[a]]
combinaciones = combinaciones_2

-----
-- Ejercicio 14. Definir la función
-- combinacionesN :: Integer -> Integer -> [[Int]]
-- tal que (combinacionesN n k) es la lista de las combinaciones de
-- orden k de los n primeros números. Por ejemplo,
-- ghci> combinacionesN 4 2
-- [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
-- ghci> combinacionesN 4 3
-- [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
-----

combinacionesN :: Integer -> Integer -> [[Integer]]
combinacionesN n k = combinaciones k [1..n]

-----
-- Ejercicio 15. Definir, usando combinacionesN, la función
-- numeroCombinaciones :: Integer -> Integer -> Integer
-- tal que (numeroCombinaciones n k) es el número de combinaciones de

```

```
-- orden k de un conjunto con n elementos. Por ejemplo,
--   numeroCombinaciones 4 2 == 6
--   numeroCombinaciones 4 3 == 4
```

```
-----
numeroCombinaciones :: Integer -> Integer -> Integer
numeroCombinaciones n k = genericLength (combinacionesN n k)
```

```
-- Puede definirse por composición
numeroCombinaciones_2 :: Integer -> Integer -> Integer
numeroCombinaciones_2 = (genericLength .) . combinacionesN
```

```
-- Para facilitar la escritura de las definiciones por composición de
-- funciones con dos argumentos, se puede definir
(.:) :: (c -> d) -> (a -> b -> c) -> a -> b -> d
(.:) = (.) . (.)
```

```
-- con lo que la definición anterior se simplifica a
numeroCombinaciones_3 :: Integer -> Integer -> Integer
numeroCombinaciones_3 = genericLength .: combinacionesN
```

```
-----
-- Ejercicio 16. Definir la función
--   comb :: Integer -> Integer -> Integer
-- tal que (comb n k) es el número combinatorio n sobre k; es decir, .
--   (comb n k) = n! / (k!(n-k)!).
-- Por ejemplo,
--   comb 4 2 == 6
--   comb 4 3 == 4
```

```
-----
comb :: Integer -> Integer -> Integer
comb n k = (fact n) 'div' ((fact k) * (fact (n-k)))
```

```
-----
-- Ejercicio 17. Definir, usando comb, la función
--   numeroCombinaciones' :: Integer -> Integer -> Integer
-- tal que (numeroCombinaciones' n k) es el número de combinaciones de
-- orden k de un conjunto con n elementos. Por ejemplo,
--   numeroCombinaciones' 4 2 == 6
```

```

--      numeroCombinaciones' 4 3 == 4
-----

numeroCombinaciones' :: Integer -> Integer -> Integer
numeroCombinaciones' = comb

-----

-- Ejercicio 18. Definir la función
--      prop_numeroCombinaciones :: Integer -> Bool
-- tal que (prop_numeroCombinaciones n) se verifica si las funciones
-- numeroCombinaciones y numeroCombinaciones' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--      prop_numeroCombinaciones 5 == True
-----

prop_numeroCombinaciones :: Integer -> Bool
prop_numeroCombinaciones n =
  and [numeroCombinaciones n k == numeroCombinaciones' n k | k <- [1..n]]

-----

-- § Combinaciones con repetición
-----

-----

-- Ejercicio 19. Definir la función
--      combinacionesR :: Integer -> [a] -> [[a]]
-- tal que (combinacionesR k xs) es la lista de las combinaciones orden
-- k de los elementos de xs con repeticiones. Por ejemplo,
--      ghci> combinacionesR 2 "abc"
--      ["aa","ab","ac","bb","bc","cc"]
--      ghci> combinacionesR 3 "bc"
--      ["bbb","bbc","bcc","ccc"]
--      ghci> combinacionesR 3 "abc"
--      ["aaa","aab","aac","abb","abc","acc","bbb","bbc","bcc","ccc"]
-----

combinacionesR :: Integer -> [a] -> [[a]]
combinacionesR _ [] = []
combinacionesR 0 _ = [[]]
combinacionesR k (x:xs) =

```

```
[x:ys | ys <- combinacionesR (k-1) (x:xs)] ++ combinacionesR k xs
```

```
-----
-- Ejercicio 20. Definir la función
```

```
-- combinacionesRN :: Integer -> Integer -> [[Integer]]
```

```
-- tal que (combinacionesRN n k) es la lista de las combinaciones orden
-- k de los primeros n números naturales. Por ejemplo,
```

```
-- ghci> combinacionesRN 3 2
```

```
-- [[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]
```

```
-- ghci> combinacionesRN 2 3
```

```
-- [[1,1,1],[1,1,2],[1,2,2],[2,2,2]]
-----
```

```
combinacionesRN :: Integer -> Integer -> [[Integer]]
```

```
combinacionesRN n k = combinacionesR k [1..n]
```

```
-----
-- Ejercicio 21. Definir, usando combinacionesRN, la función
```

```
-- numeroCombinacionesR :: Integer -> Integer -> Integer
```

```
-- tal que (numeroCombinacionesR n k) es el número de combinaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
```

```
-- numeroCombinacionesR 3 2 == 6
```

```
-- numeroCombinacionesR 2 3 == 4
-----
```

```
numeroCombinacionesR :: Integer -> Integer -> Integer
```

```
numeroCombinacionesR n k = genericLength (combinacionesRN n k)
```

```
-----
-- Ejercicio 22. Definir, usando comb, la función
```

```
-- numeroCombinacionesR' :: Integer -> Integer -> Integer
```

```
-- tal que (numeroCombinacionesR' n k) es el número de combinaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
```

```
-- numeroCombinacionesR' 3 2 == 6
```

```
-- numeroCombinacionesR' 2 3 == 4
-----
```

```
numeroCombinacionesR' :: Integer -> Integer -> Integer
```

```
numeroCombinacionesR' n k = comb (n+k-1) k
```



```

-----
-- Ejercicio 23. Definir la función
--   prop_numeroCombinacionesR :: Integer -> Bool
-- tal que (prop_numeroCombinacionesR n) se verifica si las funciones
-- numeroCombinacionesR y numeroCombinacionesR' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--   prop_numeroCombinacionesR 5 == True
-----

```

```

prop_numeroCombinacionesR :: Integer -> Bool
prop_numeroCombinacionesR n =
  and [numeroCombinacionesR n k == numeroCombinacionesR' n k |
       k <- [1..n]]

```

```

-----
-- § Variaciones
-----

```

```

-----
-- Ejercicio 24. Definir la función
--   variaciones :: Integer -> [a] -> [[a]]
-- tal que (variaciones n xs) es la lista de las variaciones n-arias
-- de la lista xs. Por ejemplo,
--   variaciones 2 "abc" == ["ab","ba","ac","ca","bc","cb"]
-----

```

```

variaciones :: Integer -> [a] -> [[a]]
variaciones k xs =
  concat (map permutaciones (combinaciones k xs))

```

```

-----
-- Ejercicio 25. Definir la función
--   variacionesN :: Integer -> Integer -> [[Integer]]
-- tal que (variacionesN n k) es la lista de las variaciones de orden k
-- de los n primeros números. Por ejemplo,
--   variacionesN 3 2 == [[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]]
-----

```

```

variacionesN :: Integer -> Integer -> [[Integer]]
variacionesN n k = variaciones k [1..n]

```

```

-----
-- Ejercicio 26. Definir, usando variacionesN, la función
--   numeroVariaciones :: Integer -> Integer -> Integer
-- tal que (numeroVariaciones n k) es el número de variaciones de orden
-- k de un conjunto con n elementos. Por ejemplo,
--   numeroVariaciones 4 2 == 12
--   numeroVariaciones 4 3 == 24
-----

```

```

numeroVariaciones :: Integer -> Integer -> Integer
numeroVariaciones n k = genericLength (variacionesN n k)

```

```

-----
-- Ejercicio 27. Definir, usando product, la función
--   numeroVariaciones' :: Integer -> Integer -> Integer
-- tal que (numeroVariaciones' n k) es el número de variaciones de orden
-- k de un conjunto con n elementos. Por ejemplo,
--   numeroVariaciones' 4 2 == 12
--   numeroVariaciones' 4 3 == 24
-----

```

```

numeroVariaciones' :: Integer -> Integer -> Integer
numeroVariaciones' n k = product [(n-k+1)..n]

```

```

-----
-- Ejercicio 28. Definir la función
--   prop_numeroVariaciones :: Integer -> Bool
-- tal que (prop_numeroVariaciones n) se verifica si las funciones
-- numeroVariaciones y numeroVariaciones' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
--   prop_numeroVariaciones 5 == True
-----

```

```

prop_numeroVariaciones :: Integer -> Bool
prop_numeroVariaciones n =
  and [numeroVariaciones n k == numeroVariaciones' n k | k <- [1..n]]

```

```

-----
-- § Variaciones con repetición

```

```

-----
--
-- -----
-- Ejercicio 28. Definir la función
--   variacionesR :: Integer -> [a] -> [[a]]
-- tal que (variacionesR k xs) es la lista de las variaciones de orden
-- k de los elementos de xs con repeticiones. Por ejemplo,
--   ghci> variacionesR 1 "ab"
--   ["a","b"]
--   ghci> variacionesR 2 "ab"
--   ["aa","ab","ba","bb"]
--   ghci> variacionesR 3 "ab"
--   ["aaa","aab","aba","abb","baa","bab","bba","bbb"]
-----

```

```

variacionesR :: Integer -> [a] -> [[a]]
variacionesR _ [] = [[]]
variacionesR 0 _ = [[]]
variacionesR k xs =
  [z:ys | z <- xs, ys <- variacionesR (k-1) xs]

```

```

-----
--
-- -----
-- Ejercicio 30. Definir la función
--   variacionesRN :: Integer -> Integer -> [[Integer]]
-- tal que (variacionesRN n k) es la lista de las variaciones orden
-- k de los primeros n números naturales. Por ejemplo,
--   ghci> variacionesRN 3 2
--   [[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
--   ghci> variacionesRN 2 3
--   [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
-----

```

```

variacionesRN :: Integer -> Integer -> [[Integer]]
variacionesRN n k = variacionesR k [1..n]

```

```

-----
--
-- -----
-- Ejercicio 31. Definir, usando variacionesR, la función
--   numeroVariacionesR :: Integer -> Integer -> Integer
-- tal que (numeroVariacionesR n k) es el número de variaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,

```

```
-- numeroVariacionesR 3 2 == 9
-- numeroVariacionesR 2 3 == 8
```

```
-----
numeroVariacionesR :: Integer -> Integer -> Integer
numeroVariacionesR n k = genericLength (variacionesRN n k)
```

```
-----
-- Ejercicio 32. Definir, usando (^), la función
-- numeroVariacionesR' :: Integer -> Integer -> Integer
-- tal que (numeroVariacionesR' n k) es el número de variaciones con
-- repetición de orden k de un conjunto con n elementos. Por ejemplo,
-- numeroVariacionesR' 3 2 == 9
-- numeroVariacionesR' 2 3 == 8
```

```
-----
numeroVariacionesR' :: Integer -> Integer -> Integer
numeroVariacionesR' n k = n^k
```

```
-----
-- Ejercicio 33. Definir la función
-- prop_numeroVariacionesR :: Integer -> Bool
-- tal que (prop_numeroVariacionesR n) se verifica si las funciones
-- numeroVariacionesR y numeroVariacionesR' son equivalentes para
-- los n primeros números y todo k entre 1 y n. Por ejemplo,
-- prop_numeroVariacionesR 5 == True
```

```
-----
prop_numeroVariacionesR :: Integer -> Bool
prop_numeroVariacionesR n =
  and [numeroVariacionesR n k == numeroVariacionesR' n k |
       k <- [1..n]]
```

```
-----
-- § El triángulo de Pascal
-----
```

```
-----
-- Ejercicio 34.1. El triángulo de Pascal es un triángulo de números
-- 1
```

```

--      1 1
--     1 2 1
--    1 3 3 1
--   1 4 6 4 1
--  1 5 10 10 5 1
--  .....
-- construido de la siguiente forma
-- * la primera fila está formada por el número 1;
-- * las filas siguientes se construyen sumando los números adyacentes
--   de la fila superior y añadiendo un 1 al principio y al final de la
--   fila.
--
-- Definir la función
--   pascal :: Integer -> [Integer]
-- tal que (pascal n) es la n-ésima fila del triángulo de Pascal. Por
-- ejemplo,
--   pascal 6 == [1,5,10,10,5,1]
-----

pascal :: Integer -> [Integer]
pascal 1 = [1]
pascal n = [1] ++ [x+y | (x,y) <- pares (pascal (n-1))] ++ [1]

-- (pares xs) es la lista formada por los pares de elementos adyacentes
-- de la lista xs. Por ejemplo,
--   pares [1,4,6,4,1] == [(1,4),(4,6),(6,4),(4,1)]
pares :: [a] -> [(a,a)]
pares (x:y:xs) = (x,y) : pares (y:xs)
pares _       = []

-- otra definición de pares, usando zip, es
pares' :: [a] -> [(a,a)]
pares' xs = zip xs (tail xs)

-- las definiciones son equivalentes
prop_pares :: [Integer] -> Bool
prop_pares xs =
  pares xs == pares' xs

-- La comprobación es

```

```
-- ghci> quickCheck prop_pares
-- +++ OK, passed 100 tests.
```

```
-----
-- Ejercicio 34.2. Comprobar con QuickCheck, que la fila n-ésima del
-- triángulo de Pascal tiene n elementos.
-----
```

```
-- La propiedad es
```

```
prop_Pascal :: Integer -> Property
```

```
prop_Pascal n =
```

```
  n >= 1 ==> genericLength (pascal n) == n
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_Pascal
```

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

```
-----
-- Ejercicio 34.3. Comprobar con QuickCheck, que la suma de los
-- elementos de la fila n-ésima del triángulo de Pascal es igual a
--  $2^{(n-1)}$ .
-----
```

```
-- la propiedad es
```

```
prop_sumaPascal :: Integer -> Property
```

```
prop_sumaPascal n =
```

```
  n >= 1 ==> sum (pascal n) == 2(n-1)
```

```
-- La comprobación es
```

```
-- ghci> quickCheck prop_sumaPascal
```

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

```
-----
-- Ejercicio 34.4. Comprobar con QuickCheck, que el m-ésimo elemento de
-- la fila (n+1)-ésima del triángulo de Pascal es el número combinatorio
-- (comb n m).
-----
```

```
-- La propiedad es
```

```
prop_Combinaciones :: Integer -> Property
```

```
prop_Combinaciones n =
  n >= 1 ==> pascal n == [comb (n-1) m | m <- [0..n-1]]

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





# Relación 23

## Cálculo numérico

```
-----  
-- Introducción --  
-----  
  
-- En esta relación se definen funciones para resolver los siguientes  
-- problemas de cálculo numérico:  
-- * diferenciación numérica,  
-- * cálculo de la raíz cuadrada mediante el método de Herón,  
-- * cálculo de los ceros de una función por el método de Newton y  
-- * cálculo de funciones inversas.  
  
-----  
-- Importación de librerías --  
-----  
  
import Test.QuickCheck  
  
-----  
-- Diferenciación numérica --  
-----  
  
-----  
-- Ejercicio 1.1. Definir la función  
-- derivada :: Double -> (Double -> Double) -> Double -> Double  
-- tal que (derivada a f x) es el valor de la derivada de la función f  
-- en el punto x con aproximación a. Por ejemplo,  
-- derivada 0.001 sin pi == -0.9999998333332315
```

```

--   derivada 0.001 cos pi == 4.999999583255033e-4
-----

derivada :: Double -> (Double -> Double) -> Double -> Double
derivada a f x = (f(x+a)-f(x))/a

-----

-- Ejercicio 1.2. Definir las funciones
--   derivadaBurda :: (Double -> Double) -> Double -> Double
--   derivadaFina  :: (Double -> Double) -> Double -> Double
--   derivadaSuper :: (Double -> Double) -> Double -> Double
-- tales que
--   * (derivadaBurda f x) es el valor de la derivada de la función f
--     en el punto x con aproximación 0.01,
--   * (derivadaFina f x) es el valor de la derivada de la función f
--     en el punto x con aproximación 0.0001.
--   * (derivadaSuper f x) es el valor de la derivada de la función f
--     en el punto x con aproximación 0.000001.
-- Por ejemplo,
--   derivadaBurda cos pi == 4.999958333473664e-3
--   derivadaFina  cos pi == 4.999999969612645e-5
--   derivadaSuper cos pi == 5.000444502911705e-7
-----

derivadaBurda :: (Double -> Double) -> Double -> Double
derivadaBurda = derivada 0.01

derivadaFina  :: (Double -> Double) -> Double -> Double
derivadaFina  = derivada 0.0001

derivadaSuper :: (Double -> Double) -> Double -> Double
derivadaSuper = derivada 0.000001

-----

-- Ejercicio 1.3. Definir la función
--   derivadaFinaDelSeno :: Double -> Double
-- tal que (derivadaFinaDelSeno x) es el valor de la derivada fina del
-- seno en x. Por ejemplo,
--   derivadaFinaDelSeno pi == -0.9999999983354436
-----

```

```
derivadaFinaDelSeno :: Double -> Double
```

```
derivadaFinaDelSeno = derivadaFina sin
```

```
-----
-- Cálculo de la raíz cuadrada                                     --
-----
```

```
-----
-- Ejercicio 2.1. En los siguientes apartados de este ejercicio se va a
-- calcular la raíz cuadrada de un número basándose en las siguientes
-- propiedades:
```

```
-- * Si  $y$  es una aproximación de la raíz cuadrada de  $x$ , entonces
--  $(y+x/y)/2$  es una aproximación mejor.
```

```
-- * El límite de la sucesión definida por
```

```
--  $x_0 = 1$ 
--  $x_{n+1} = (x_n + x/x_n)/2$ 
-- es la raíz cuadrada de  $x$ .
```

```
--
```

```
-- Definir, por iteración con until, la función
```

```
-- raiz :: Double -> Double
```

```
-- tal que (raiz x) es la raíz cuadrada de x calculada usando la
```

```
-- propiedad anterior con una aproximación de 0.00001 y tomando como
```

```
-- v. Por ejemplo,
```

```
-- raiz 9 == 3.000000001396984
```

```
-----
raiz :: Double -> Double
```

```
raiz x = raiz' 1
```

```
  where raiz' y | acceptable y = y
          | otherwise      = raiz' (mejora y)
        mejora y      = 0.5*(y+x/y)
        acceptable y = abs(y*y-x) < 0.00001
```

```
-----
-- Ejercicio 3.2. Definir el operador
-- (~=) :: Double -> Double -> Bool
-- tal que (x ~= y) si  $|x-y| < 0.001$ . Por ejemplo,
-- 3.05 ~= 3.07 == False
-- 3.00005 ~= 3.00007 == True
```

```

-----
infix 5 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs(x-y) < 0.001

```

```

-----
-- Ejercicio 3.3. Comprobar con QuickCheck que si x es positivo,
-- entonces
--   (raiz x)^2 ~= x
-----

```

```

-- La propiedad es
prop_raiz :: Double -> Bool
prop_raiz x =
  (raiz x')^2 ~= x'
  where x' = abs x

```

```

-- La comprobación es
--   *Main> quickCheck prop_raiz
--   OK, passed 100 tests.

```

```

-----
-- Ejercicio 3.4. Definir por recursión la función
--   until' :: (a -> Bool) -> (a -> a) -> a -> a
-- tal que (until' p f x) es el resultado de aplicar la función f a x el
-- menor número posible de veces, hasta alcanzar un valor que satisface
-- el predicado p. Por ejemplo,
--   until' (>1000) (2*) 1 == 1024
-- Nota: until' es equivalente a la predefinida until.
-----

```

```

until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f x | p x      = x
              | otherwise = until' p f (f x)

```

```

-----
-- Ejercicio 3.5. Definir, por iteración con until, la función
--   raizI :: Double -> Double
-- tal que (raizI x) es la raíz cuadrada de x calculada usando la

```

```

-- propiedad anterior. Por ejemplo,
--   raizI 9 == 3.000000001396984
-----

raizI :: Double -> Double
raizI x = until aceptable mejora 1
      where mejora y    = 0.5*(y+x/y)
            aceptable y = abs(y*y-x) < 0.00001
-----

-- Ejercicio 3.6. Comprobar con QuickCheck que si x es positivo,
-- entonces
--   (raizI x)^2 ~= x
-----

-- La propiedad es
prop_raizI :: Double -> Bool
prop_raizI x =
  (raizI x')^2 ~= x'
  where x' = abs x

-- La comprobación es
--   *Main> quickCheck prop_raizI
--   OK, passed 100 tests.
-----

-- Ceros de una función
-----

-- Ejercicio 4. Los ceros de una función pueden calcularse mediante el
-- método de Newton basándose en las siguientes propiedades:
-- * Si b es una aproximación para el punto cero de f, entonces
--    $b - f(b)/f'(b)$  es una mejor aproximación.
-- * el límite de la sucesión  $x_n$  definida por
--    $x_0 = 1$ 
--    $x_{n+1} = x_n - f(x_n)/f'(x_n)$ 
--   es un cero de f.
-----

```

```

-----
-- Ejercicio 4.1. Definir por recursión la función
-- puntoCero :: (Double -> Double) -> Double
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
-- puntoCero cos == 1.5707963267949576
-----

puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCero' f 1
  where puntoCero' f x | acceptable x = x
                    | otherwise     = puntoCero' f (mejora x)
        mejora b      = b - f b / derivadaFina f b
        acceptable b  = abs (f b) < 0.00001

-----
-- Ejercicio 4.2. Definir, por iteración con until, la función
-- puntoCeroI :: (Double -> Double) -> Double
-- tal que (puntoCeroI f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
-- puntoCeroI cos == 1.5707963267949576
-----

puntoCeroI :: (Double -> Double) -> Double
puntoCeroI f = until acceptable mejora 1
  where mejora b      = b - f b / derivadaFina f b
        acceptable b  = abs (f b) < 0.00001

-----
-- Funciones inversas
-----

-----
-- Ejercicio 5. En este ejercicio se usará la función puntoCero para
-- definir la inversa de distintas funciones.
-----

-----
-- Ejercicio 5.1. Definir, usando puntoCero, la función
-- raizCuadrada :: Double -> Double

```

```
-- tal que (raizCuadrada x) es la raíz cuadrada de x. Por ejemplo,  
--   raizCuadrada 9 == 3.000000002941184  
-----
```

```
raizCuadrada :: Double -> Double  
raizCuadrada a = puntoCero f  
  where f x = x*x-a  
-----
```

```
-- Ejercicio 5.2. Comprobar con QuickCheck que si x es positivo,  
-- entonces  
--   (raizCuadrada x)^2 ~= x  
-----
```

```
-- La propiedad es  
prop_raizCuadrada :: Double -> Bool  
prop_raizCuadrada x =  
  (raizCuadrada x')^2 ~= x'  
  where x' = abs x  
-----
```

```
-- La comprobación es  
--   *Main> quickCheck prop_raizCuadrada  
--   OK, passed 100 tests.  
-----
```

```
-- Ejercicio 5.3. Definir, usando puntoCero, la función  
--   raizCubica :: Double -> Double  
-- tal que (raizCubica x) es la raíz cuadrada de x. Por ejemplo,  
--   raizCubica 27 == 3.000000000196048  
-----
```

```
raizCubica :: Double -> Double  
raizCubica a = puntoCero f  
  where f x = x*x*x-a  
-----
```

```
-- Ejercicio 5.4. Comprobar con QuickCheck que si x es positivo,  
-- entonces  
--   (raizCubica x)^3 ~= x  
-----
```

```
-- La propiedad es
prop_raizCubica :: Double -> Bool
prop_raizCubica x =
  (raizCubica x)^3 == x
  where x' = abs x
```

```
-- La comprobación es
-- *Main> quickCheck prop_raizCubica
-- OK, passed 100 tests.
```

```
-----
-- Ejercicio 5.5. Definir, usando puntoCero, la función
--   arcoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--   arcoseno 1 == 1.5665489428306574
-----
```

```
arcoseno :: Double -> Double
arcoseno a = puntoCero f
  where f x = sin x - a
```

```
-----
-- Ejercicio 5.6. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--   sin (arcoseno x) == x
-----
```

```
-- La propiedad es
prop_arcoseno :: Double -> Bool
prop_arcoseno x =
  sin (arcoseno x) == x
  where x' = abs (x - fromIntegral (truncate x))
```

```
-- La comprobación es
-- *Main> quickCheck prop_arcoseno
-- OK, passed 100 tests.
```

```
-----
-- Ejercicio 5.7. Definir, usando puntoCero, la función
```



```

--   arcocoseno :: Double -> Double
--   tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--   arcocoseno 0 == 1.5707963267949576
-----

arcocoseno :: Double -> Double
arcocoseno a = puntoCero f
  where f x = cos x - a

-----

-- Ejercicio 5.8. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--   cos (arcocoseno x) ~= x
-----

-- La propiedad es
prop_arcocoseno :: Double -> Bool
prop_arcocoseno x =
  cos (arcocoseno x) ~= x
  where x' = abs (x - fromIntegral (truncate x))

-- La comprobación es
--   *Main> quickCheck prop_arcocoseno
--   OK, passed 100 tests.

-----

-- Ejercicio 5.7. Definir, usando puntoCero, la función
--   inversa :: (Double -> Double) -> Double -> Double
-- tal que (inversa g x) es el valor de la inversa de g en x. Por
-- ejemplo,
--   inversa (^2) 9 == 3.0000000002941184
-----

inversa :: (Double -> Double) -> Double -> Double
inversa g a = puntoCero f
  where f x = g x - a

-----

-- Ejercicio 5.8. Redefinir, usando inversa, las funciones raizCuadrada,
-- raizCubica, arcoseno y arcocoseno.

```

```
raizCuadrada' = inversa (^2)
raizCubica'   = inversa (^3)
arcoseno'     = inversa sin
arcocoseno'   = inversa cos
```

# Relación 24

## Operaciones con el TAD de polinomios

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación es ampliar el conjunto de operaciones  
-- sobre polinomios definidas utilizando las implementaciones del TAD de  
-- polinomio estudiadas en el tema 21 que se pueden descargar desde  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/codigos.zip  
-- Además, en algunos ejemplos de usan polinomios con coeficientes  
-- racionales. En Haskell, el número racional  $x/y$  se representa por  
--  $x\%y$ . El TAD de los números racionales está definido en el módulo  
-- Data.Ratio.  
--  
-- Las transparencias del tema 21 se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-21.pdf  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
import PolOperaciones  
import Test.QuickCheck  
import Data.Ratio  
  
-- -----
```

```
-- Ejercicio 1. Definir la función
--   creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
--   tal que (creaPolDispersa xs) es el polinomio cuya representación
--   dispersa es xs. Por ejemplo,
--   creaPolDispersa [7,0,0,4,0,3] == 7*x^5 + 4*x^2 + 3
```

```
creaPolDispersa :: (Num a, Eq a) => [a] -> Polinomio a
creaPolDispersa [] = polCero
creaPolDispersa (x:xs) = consPol (length xs) x (creaPolDispersa xs)
```

```
-- Ejercicio 2. Definir la función
--   creaPolDensa :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
--   tal que (creaPolDensa xs) es el polinomio cuya representación
--   densa es xs. Por ejemplo,
--   creaPolDensa [(5,7),(4,2),(3,0)] == 7*x^5 + 2*x^4
```

```
creaPolDensa :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
creaPolDensa [] = polCero
creaPolDensa ((n,a):ps) = consPol n a (creaPolDensa ps)
```

```
-- Nota. En el resto de la sucesión se usará en los ejemplos los
-- los polinomios que se definen a continuación.
```

```
pol1, pol2, pol3 :: (Num a, Eq a) => Polinomio a
pol1 = creaPolDensa [(5,1),(2,5),(1,4)]
pol2 = creaPolDispersa [2,3]
pol3 = creaPolDensa [(7,2),(4,5),(2,5)]
```

```
pol4, pol5, pol6 :: Polinomio Rational
pol4 = creaPolDensa [(4,3),(2,5),(0,3)]
pol5 = creaPolDensa [(2,6),(1,2)]
pol6 = creaPolDensa [(2,8),(1,14),(0,3)]
```

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

```

-- densa :: Num a => Polinomio a -> [(Int,a)]
-- tal que (densa p) es la representación densa del polinomio p. Por
-- ejemplo,
-- pol1      == x^5 + 5*x^2 + 4*x
-- densa pol1 == [(5,1),(2,5),(1,4)]
-----

```

```

densa :: Num a => Polinomio a -> [(Int,a)]
densa p | esPolCero p = []
        | otherwise   = (grado p, coefLider p) : densa (restoPol p)
-----

```

```

-- Ejercicio 4. Definir la función
-- densaAdispersa :: Num a => [(Int,a)] -> [a]
-- tal que (densaAdispersa ps) es la representación dispersa del
-- polinomio cuya representación densa es ps. Por ejemplo,
-- densaAdispersa [(5,1),(2,5),(1,4)] == [1,0,0,5,4,0]
-----

```

```

densaAdispersa :: Num a => [(Int,a)] -> [a]
densaAdispersa [] = []
densaAdispersa [(n,a)] = a : replicate n 0
densaAdispersa ((n,a):(m,b):ps) =
  a : (replicate (n-m-1) 0) ++ densaAdispersa ((m,b):ps)
-----

```

```

-- Ejercicio 5. Definir la función
-- dispersa :: Num a => Polinomio a -> [a]
-- tal que (dispersa p) es la representación dispersa del polinomio
-- p. Por ejemplo,
-- pol1      == x^5 + 5*x^2 + 4*x
-- dispersa pol1 == [1,0,0,5,4,0]
-----

```

```

dispersa :: Num a => Polinomio a -> [a]
dispersa = densaAdispersa . densa
-----

```

```

-- Ejercicio 6. Definir la función
-- coeficiente :: Num a => Int -> Polinomio a -> a
-----

```

```
-- tal que (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--   poll                == x^5 + 5*x^2 + 4*x
--   coeficiente 2 poll == 5
--   coeficiente 3 poll == 0
```

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

-- Otra definición equivalente es

```
coeficiente' :: Num a => Int -> Polinomio a -> a
coeficiente' k p = busca k (densa p)
  where busca k ps = head ([a | (n,a) <- ps, n == k] ++ [0])
```

-----

-- Ejercicio 7. Definir la función

```
--   coeficientes :: Num a => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de los coeficientes del
-- polinomio p. Por ejemplo,
--   poll                == x^5 + 5*x^2 + 4*x
--   coeficientes poll == [1,0,0,5,4,0]
```

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

-- Una definición equivalente es

```
coeficientes' :: Num a => Polinomio a -> [a]
coeficientes' = dispersa
```

-----

-- Ejercicio 8. Definir la función

```
--   potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potencia p n) es la potencia n-ésima del polinomio p. Por
-- ejemplo,
```

```
--      pol2                == 2*x + 3
--      potencia pol2 2    == 4*x^2 + 12*x + 9
--      potencia pol2 3    == 8*x^3 + 36*x^2 + 54*x + 27
```

```
-----
potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potencia p 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))
```

```
-----
-- Ejercicio 9. Mejorar la definición de potencia definiendo la función
-- potenciaM :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potenciaM p n) es la potencia n-ésima del polinomio p,
-- utilizando las siguientes propiedades:
```

```
-- * Si n es par, entonces  $x^n = (x^2)^{(n/2)}$ 
-- * Si n es impar, entonces  $x^n = x * (x^2)^{((n-1)/2)}$ 
```

```
-- Por ejemplo,
```

```
--      pol2                == 2*x + 3
--      potenciaM pol2 2    == 4*x^2 + 12*x + 9
--      potenciaM pol2 3    == 8*x^3 + 36*x^2 + 54*x + 27
```

```
-----
potenciaM :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potenciaM p 0 = polUnidad
potenciaM p n
  | even n      = potenciaM (multPol p p) (n `div` 2)
  | otherwise   = multPol p (potenciaM (multPol p p) ((n-1) `div` 2))
```

```
-----
-- Ejercicio 10. Definir la función
```

```
-- integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
-- tal que (integral p) es la integral del polinomio p cuyos coeficientes
-- son números racionales. Por ejemplo,
```

```
-- ghci> pol3
--      2*x^7 + 5*x^4 + 5*x^2
-- ghci> integral pol3
--      0.25*x^8 + x^5 + 1.6666666666666667*x^3
-- ghci> integral pol3 :: Polinomio Rational
--      1 % 4*x^8 + x^5 + 5 % 3*x^3
```

```

integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
integral p
  | esPolCero p = polCero
  | otherwise   = consPol (n+1) (b / (fromIntegral (n+1))) (integral r)
  where n = grado p
        b = coefLider p
        r = restoPol p

```

```

-----
-- Ejercicio 11. Definir la función
--   integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
-- tal que (integralDef p a b) es la integral definida del polinomio p
-- cuyos coeficientes son números racionales. Por ejemplo,
--   ghci> integralDef pol3 0 1
--   2.916666666666667
--   ghci> integralDef pol3 0 1 :: Rational
--   35 % 12
-----

```

```

integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
integralDef p a b = (valor q b) - (valor q a)
  where q = integral p

```

```

-----
-- Ejercicio 12. Definir la función
--   multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
-- tal que (multEscalar c p) es el polinomio obtenido multiplicando el
-- número c por el polinomio p. Por ejemplo,
--   pol2                == 2*x + 3
--   multEscalar 4 pol2  == 8*x + 12
--   multEscalar (1%4) pol2 == 1 % 2*x + 3 % 4
-----

```

```

multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
multEscalar c p
  | esPolCero p = polCero
  | otherwise   = consPol n (c*b) (multEscalar c r)
  where n = grado p
        b = coefLider p

```



```
r = restoPol p
```

```
-----
-- Ejercicio 13. Definir la función
--   cociente:: (Fractional a, Eq a) =>
--               Polinomio a -> Polinomio a -> Polinomio a
-- tal que (cociente p q) es el cociente de la división de p entre
-- q. Por ejemplo,
--   pol4 == 3 % 1*x^4 + 5 % 1*x^2 + 3 % 1
--   pol5 == 6 % 1*x^2 + 2 % 1*x
--   cociente pol4 pol5 == 1 % 2*x^2 + (-1) % 6*x + 8 % 9
-----
```

```
cociente:: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
cociente p q
  | n2 == 0    = multEscalar (1/a2) p
  | n1 < n2    = polCero
  | otherwise  = consPol n' a' (cociente p' q)
where n1 = grado p
      a1 = coefLider p
      n2 = grado q
      a2 = coefLider q
      n' = n1-n2
      a' = a1/a2
      p' = restaPol p (multPorTerm (creaTermino n' a') q)
```

```
-----
-- Ejercicio 14. Definir la función
--   resto:: (Fractional a, Eq a) =>
--           Polinomio a -> Polinomio a -> Polinomio a
-- tal que (resto p q) es el resto de la división de p entre q. Por
-- ejemplo,
--   pol4 == 3 % 1*x^4 + 5 % 1*x^2 + 3 % 1
--   pol5 == 6 % 1*x^2 + 2 % 1*x
--   resto pol4 pol5 == (-16) % 9*x + 3 % 1
-----
```

```
resto :: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
resto p q = restaPol p (multPol (cociente p q) q)
```

```

-----
-- Ejercicio 15. Definir la función
--   divisiblePol :: (Fractional a, Eq a) =>
--                 Polinomio a -> Polinomio a -> Bool
-- tal que (divisiblePol p q) se verifica si el polinomio p es divisible
-- por el polinomio q. Por ejemplo,
--   pol6 == 8 % 1*x^2 + 14 % 1*x + 3 % 1
--   pol2 == 2*x + 3
--   pol5 == 6 % 1*x^2 + 2 % 1*x
--   divisiblePol pol6 pol2 == True
--   divisiblePol pol6 pol5 == False
-----

```

```

divisiblePol :: (Fractional a, Eq a) => Polinomio a -> Polinomio a -> Bool
divisiblePol p q = esPolCero (resto p q)

```

```

-----
-- Ejercicio 16. El método de Horner para calcular el valor de un
-- polinomio se basa en representarlo de una forma forma alernativa. Por
-- ejemplo, para calcular el valor de
--   a*x^5 + b*x^4 + c*x^3 + d*x^2 + e*x + f
-- se representa como
--   (((a * x + b) * x + c) * x + d) * x + e) * x + f
-- y se evalúa de dentro hacia afuera.
--
-- Definir la función
--   horner :: Num a => Polinomio a -> a -> a
-- tal que (horner p x) es el valor del polinomio p al sustituir su
-- variable por el número x. Por ejemplo,
--   horner pol1 0      == 0
--   horner pol1 1      == 10
--   horner pol1 1.5    == 24.84375
--   horner pol1 (3%2) == 795 % 32
-----

```

```

horner :: Num a => Polinomio a -> a -> a
horner p x = hornerAux (coeficientes p) 0
  where hornerAux [] v      = v
        hornerAux (a:as) v = hornerAux as (a+v*x)

```

```
-- Una definición equivalente por plegado es
horner' :: Num a => Polinomio a -> a -> a
horner' p x = (foldr (\a b -> a + b*x) 0) (coeficientes p)
```



# Relación 25

## División y factorización de polinomios mediante la regla de Ruffini

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es implementar la regla de  
-- Ruffini y sus aplicaciones utilizando las implementaciones del TAD de  
-- polinomio estudiadas en el tema 21 que se pueden descargar desde  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/codigos.zip  
--  
-- Las transparencias del tema 21 se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-11/temas/tema-21.pdf  
  
-----  
-- Importación de librerías --  
-----  
  
import PolOperaciones  
import Test.QuickCheck  
  
-----  
-- Ejemplos --  
-----
```

-- Además de los ejemplos de polinomios (ejPol1, ejPol2 y ejPol3) que se encuentran en PolOperaciones, usaremos el siguiente ejemplo.

```
ejPol4 :: Polinomio Int
```

```
ejPol4 = consPol 3 1
          (consPol 2 2
            (consPol 1 (-1)
              (consPol 0 (-2) polCero)))
```

-----  
-- Ejercicio 1. Definir la función

```
-- divisores :: Int -> [Int]
-- tal que (divisores n) es la lista de todos los divisores enteros de
-- n. Por ejemplo,
-- divisores 4    == [1,-1,2,-2,4,-4]
-- divisores (-6) == [1,-1,2,-2,3,-3,6,-6]
```

```
divisores :: Int -> [Int]
```

```
divisores n = concat [[x,-x] | x <- [1..abs n], rem n x == 0]
```

-----  
-- Ejercicio 2. Definir la función

```
-- coeficiente :: Num a => Int -> Polinomio a -> a
-- tal que (coeficiente k p) es el coeficiente del término de grado k en
-- p. Por ejemplo:
-- coeficiente 4 ejPol1 == 3
-- coeficiente 3 ejPol1 == 0
-- coeficiente 2 ejPol1 == -5
-- coeficiente 5 ejPol1 == 0
```

```
coeficiente :: Num a => Int -> Polinomio a -> a
```

```
coeficiente k p | k == gp      = coefLider p
                | k > grado rp = 0
                | otherwise   = coeficiente k rp
  where gp = grado p
        rp = restoPol p
```

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

```
-- terminoIndep :: Num a => Polinomio a -> a
-- tal que (terminoIndep p) es el término independiente del polinomio
-- p. Por ejemplo,
-- terminoIndep ejPol1 == 3
-- terminoIndep ejPol2 == 0
-- terminoIndep ejPol4 == -2
```

```
terminoIndep :: Num a => Polinomio a -> a
terminoIndep p = coeficiente 0 p
```

```
-- Ejercicio 4. Definir la función
-- coeficientes :: Num a => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de coeficientes de p, ordenada
-- según el grado. Por ejemplo,
-- coeficientes ejPol1 == [3,0,-5,0,3]
-- coeficientes ejPol4 == [1,2,-1,-2]
-- coeficientes ejPol2 == [1,0,0,5,4,0]
```

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

```
-- Ejercicio 5. Definir la función
-- creaPol :: (Num a, Eq a) => [a] -> Polinomio a
-- tal que (creaPol cs) es el polinomio cuya lista de coeficientes es
-- cs. Por ejemplo,
-- creaPol [1,0,0,5,4,0] == x^5 + 5*x^2 + 4*x
-- creaPol [1,2,0,3,0] == x^4 + 2*x^3 + 3*x
```

```
creaPol :: (Num a, Eq a) => [a] -> Polinomio a
creaPol [] = polCero
creaPol (a:as) = consPol n a (creaPol as)
  where n = length as
```

```
-- Ejercicio 6. Comprobar con QuickCheck que, dado un polinomio p, el
-- polinomio obtenido mediante creaPol a partir de la lista de
-- coeficientes de p coincide con p.
```

```
-----
-- La propiedad es
prop_coef :: Polinomio Int -> Bool
prop_coef p =
    creaPol (coeficientes p) == p
```

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

```
-----
-- Ejercicio 7. Definir una función
-- pRuffini :: Int -> [Int] -> [Int]
-- tal que (pRuffini r cs) es la lista que resulta de aplicar un paso
-- del regla de Ruffini al número entero r y a la lista de coeficientes
-- cs. Por ejemplo,
-- pRuffini 2 [1,2,-1,-2] == [1,4,7,12]
-- pRuffini 1 [1,2,-1,-2] == [1,3,2,0]
-- ya que
```

```
--   | 1  2  -1  -2           | 1  2  -1  -2
-- 2 |   2  8  14           1 |   1  3  2
-- --+-----                --+-----
--   | 1  4  7  12           | 1  3  2  0
```

```
-----
pRuffini :: Int -> [Int] -> [Int]
pRuffini r p@(c:cs) =
    c : [x+r*y | (x,y) <- zip cs (pRuffini r p)]
```

```
-- Otra forma:
pRuffini' :: Int -> [Int] -> [Int]
pRuffini' r = scanl1 (\s x -> s * r + x)
```

```
-----
-- Ejercicio 8. Definir la función
-- cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
```



```
-- tal que (cocienteRuffini r p) es el cociente de dividir el polinomio
-- p por el polinomio x-r. Por ejemplo:
--   cocienteRuffini 2 ejPol4    == x^2 + 4*x + 7
--   cocienteRuffini (-2) ejPol4 == x^2 + -1
--   cocienteRuffini 3 ejPol4    == x^2 + 5*x + 14
-----
```

```
cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = creaPol (init (pRuffini r (coeficientes p)))
```

```
-- Ejercicio 9. Definir la función
--   restoRuffini:: Int -> Polinomio Int -> Int
-- tal que (restoRuffini r p) es el resto de dividir el polinomio p por
-- el polinomio x-r. Por ejemplo,
--   restoRuffini 2 ejPol4    == 12
--   restoRuffini (-2) ejPol4 == 0
--   restoRuffini 3 ejPol4    == 40
-----
```

```
restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (pRuffini r (coeficientes p))
```

```
-- Ejercicio 10. Comprobar con QuickCheck que, dado un polinomio p y un
-- número entero r, las funciones anteriores verifican la propiedad de
-- la división euclídea.
-----
```

```
-- La propiedad es
prop_diviEuclidea:: Int -> Polinomio Int -> Bool
prop_diviEuclidea r p =
  p == sumaPol (multPol coc div) res
  where coc = cocienteRuffini r p
         div = creaPol [1,-r]
         res = creaTermino 0 (restoRuffini r p)
```

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

```

-----
-- Ejercicio 11. Definir la función
--   esRaizRuffini :: Int -> Polinomio Int -> Bool
-- tal que (esRaizRuffini r p) se verifica si r es una raíz de p, usando
-- para ello el regla de Ruffini. Por ejemplo,
--   esRaizRuffini 0 ejPol3 == True
--   esRaizRuffini 1 ejPol3 == False
-----

```

```

esRaizRuffini :: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0

```

```

-----
-- Ejercicio 12. Definir la función
--   raicesRuffini :: Polinomio Int -> [Int]
-- tal que (raicesRuffini p) es la lista de las raíces enteras de p,
-- calculadas usando el regla de Ruffini. Por ejemplo,
--   raicesRuffini ejPol1 == []
--   raicesRuffini ejPol2 == [0, -1]
--   raicesRuffini ejPol3 == [0]
--   raicesRuffini ejPol4 == [1, -1, -2]
--   raicesRuffini polCero == []
-----

```

```

raicesRuffini :: Polinomio Int -> [Int]
raicesRuffini p
  | esPolCero p = []
  | otherwise  = aux (0 : divisores (terminoIndep p))
  where
    aux [] = []
    aux (r:rs)
      | esRaizRuffini r p = r : raicesRuffini (cocienteRuffini r p)
      | otherwise        = aux rs

```

```

-----
-- Ejercicio 13. Definir la función
--   factorizacion :: Polinomio Int -> [Polinomio Int]
-- tal que (factorizacion p) es la lista de la descomposición del
-- polinomio p en factores obtenida mediante el regla de Ruffini. Por

```

```

-- ejemplo,
-- ejPol2 == x^5 + 5*x^2 + 4*x
-- factorizacion ejPol2 == [1*x,1*x+1,x^3+-1*x^2+1*x+4]
-- ejPol4 == x^3 + 2*x^2 + -1*x + -2
-- factorizacion ejPol4 == [1*x + -1,1*x + 1,1*x + 2,1]
-- factorizacion (creaPol [1,0,0,0,-1]) == [1*x + -1,1*x + 1,x^2 + 1]
-- -----

```

```

factorizacion :: Polinomio Int -> [Polinomio Int]

```

```

factorizacion p

```

```

| esPolCero p = [p]

```

```

| otherwise = aux (0 : divisores (terminoIndep p))

```

```

where

```

```

aux [] = [p]

```

```

aux (r:rs)

```

```

| esRaizRuffini r p =

```

```

    (creaPol [1,-r]) : factorizacion (cocienteRuffini r p)

```

```

| otherwise = aux rs

```



# Relación 26

## Vectores y matrices

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación es hacer ejercicios sobre vectores y  
-- matrices con el tipo de tablas de las tablas, definido en el módulo  
-- Data.Array y explicado en el tema 18 cuyas transparencias se  
-- encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-18t.pdf  
-- Además, en algunos ejemplos de usan matrices con números racionales.  
-- En Haskell, el número racional  $x/y$  se representa por  $x\%y$ . El TAD de  
-- los números racionales está definido en el módulo Data.Ratio.  
  
-----  
-- Importación de librerías --  
-----  
  
import Data.Array  
import Data.Ratio  
  
-----  
-- Tipos de los vectores y de las matrices --  
-----  
  
-- Los vectores son tablas cuyos índices son números naturales.  
type Vector a = Array Int a
```

```

-- Las matrices son tablas cuyos índices son pares de números
-- naturales.
type Matriz a = Array (Int,Int) a

-----
-- Operaciones básicas con matrices
-----

-----
-- Ejercicio 1. Definir la función
--   listaVector :: Num a => [a] -> Vector a
-- tal que (listaVector xs) es el vector correspondiente a la lista
-- xs. Por ejemplo,
--   ghci> listaVector [3,2,5]
--   array (1,3) [(1,3),(2,2),(3,5)]
-----

listaVector :: Num a => [a] -> Vector a
listaVector xs = listArray (1,n) xs
  where n = length xs

-----
-- Ejercicio 2. Definir la función
--   listaMatriz :: Num a => [[a]] -> Matriz a
-- tal que (listaMatriz xss) es la matriz cuyas filas son los elementos
-- de xss. Por ejemplo,
--   ghci> listaMatriz [[1,3,5],[2,4,7]]
--   array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),5),
--                        ((2,1),2),((2,2),4),((2,3),7)]
-----

listaMatriz :: Num a => [[a]] -> Matriz a
listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
  where m = length xss
        n = length (head xss)

-----
-- Ejercicio 3. Definir la función
--   numFilas :: Num a => Matriz a -> Int
-- tal que (numFilas m) es el número de filas de la matriz m. Por

```

```
-- ejemplo,  
-- numFilas (listaMatriz [[1,3,5],[2,4,7]]) == 2  
-----
```

```
numFilas :: Num a => Matriz a -> Int  
numFilas = fst . snd . bounds
```

```
-----  
-- Ejercicio 4. Definir la función  
-- numColumnas :: Num a => Matriz a -> Int  
-- tal que (numColumnas m) es el número de columnas de la matriz  
-- m. Por ejemplo,  
-- numColumnas (listaMatriz [[1,3,5],[2,4,7]]) == 3  
-----
```

```
numColumnas :: Num a => Matriz a -> Int  
numColumnas = snd . snd . bounds
```

```
-----  
-- Ejercicio 5. Definir la función  
-- dimension :: Num a => Matriz a -> (Int,Int)  
-- tal que (dimension m) es el número de columnas de la matriz m. Por  
-- ejemplo,  
-- dimension (listaMatriz [[1,3,5],[2,4,7]]) == (2,3)  
-----
```

```
dimension :: Num a => Matriz a -> (Int,Int)  
dimension p = (numFilas p, numColumnas p)
```

```
-----  
-- Ejercicio 6. Definir la función  
-- separa :: Int -> [a] -> [[a]]  
-- tal que (separa n xs) es la lista obtenida separando los elementos de  
-- xs en grupos de n elementos (salvo el último que puede tener menos de  
-- n elementos). Por ejemplo,  
-- separa 3 [1..11] == [[1,2,3],[4,5,6],[7,8,9],[10,11]]  
-----
```

```
separa :: Int -> [a] -> [[a]]  
separa _ [] = []
```

```
separa n xs = take n xs : separa n (drop n xs)
```

```
-----
-- Ejercicio 7. Definir la función
--   matrizLista :: Num a => Matriz a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
--   ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> m
--   array ((1,1),(2,3)) [((1,1),5),((1,2),1),((1,3),0),
--                        ((2,1),3),((2,2),2),((2,3),6)]
--   ghci> matrizLista m
--   [[5,1,0],[3,2,6]]
-----
```

```
matrizLista :: Num a => Matriz a -> [[a]]
matrizLista p = separa (numColumnas p) (elems p)
```

```
-----
-- Ejercicio 8. Definir la función
--   vectorLista :: Num a => Vector a -> [a]
-- tal que (vectorLista x) es la lista de los elementos del vector
-- v. Por ejemplo,
--   ghci> let v = listaVector [3,2,5]
--   ghci> v
--   array (1,3) [(1,3),(2,2),(3,5)]
--   ghci> vectorLista v
--   [3,2,5]
-----
```

```
vectorLista :: Num a => Vector a -> [a]
vectorLista = elems
```

```
-----
-- Suma de matrices
-----
```

```
-----
-- Ejercicio 9. Definir la función
--   sumaMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
-----
```



```
-- tal que (sumaMatrices x y) es la suma de las matrices x e y. Por
-- ejemplo,
-- ghci> let m1 = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> let m2 = listaMatriz [[4,6,3],[1,5,2]]
-- ghci> matrizLista (sumaMatrices m1 m2)
-- [[9,7,3],[4,7,8]]
```

```
-----
sumaMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
sumaMatrices p q =
  array ((1,1),(m,n)) [((i,j),p!(i,j)+q!(i,j)) |
                       i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
```

```
-----
-- Ejercicio 10. Definir la función
-- filaMat :: Num a => Int -> Matriz a -> Vector a
-- tal que (filaMat i p) es el vector correspondiente a la fila i-ésima
-- de la matriz p. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
-- ghci> filaMat 2 p
-- array (1,3) [(1,3),(2,2),(3,6)]
-- ghci> vectorLista (filaMat 2 p)
-- [3,2,6]
```

```
-----
filaMat :: Num a => Int -> Matriz a -> Vector a
filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
  where n = numColumnas p
```

```
-----
-- Ejercicio 11. Definir la función
-- columnaMat :: Num a => Int -> Matriz a -> Vector a
-- tal que (columnaMat j p) es el vector correspondiente a la columna
-- j-ésima de la matriz p. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
-- ghci> columnaMat 2 p
-- array (1,3) [(1,1),(2,2),(3,5)]
-- ghci> vectorLista (columnaMat 2 p)
-- [1,2,5]
```

```

-----
columnaMat :: Num a => Int -> Matriz a -> Vector a
columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]
  where m = numFilas p

```

```

-----
-- Producto de matrices
-----

```

```

-----
-- Ejercicio 12. Definir la función
--   prodEscalar :: Num a => Vector a -> Vector a -> a
-- tal que (prodEscalar v1 v2) es el producto escalar de los vectores v1
-- y v2. Por ejemplo,
--   ghci> let v = listaVector [3,1,10]
--   ghci> prodEscalar v v
--   110
-----

```

```

prodEscalar :: Num a => Vector a -> Vector a -> a
prodEscalar v1 v2 =
  sum [i*j | (i,j) <- zip (elems v1) (elems v2)]

```

```

-----
-- Ejercicio 13. Definir la función
--   prodMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
-- tal que (prodMatrices p q) es el producto de las matrices p y q. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[3,1],[2,4]]
--   ghci> prodMatrices p p
--   array ((1,1),(2,2)) [((1,1),11),((1,2),7),((2,1),14),((2,2),18)]
--   ghci> matrizLista (prodMatrices p p)
--   [[11,7],[14,18]]
--   ghci> let q = listaMatriz [[7],[5]]
--   ghci> prodMatrices p q
--   array ((1,1),(2,1)) [((1,1),26),((2,1),34)]
--   ghci> matrizLista (prodMatrices p q)
--   [[26],[34]]
-----

```

```

prodMatrices :: Num a => Matriz a -> Matriz a -> Matriz a
prodMatrices p q =
  array ((1,1),(m,n))
    [((i,j), prodEscalar (filaMat i p) (columnaMat j q)) |
      i <- [1..m], j <- [1..n]]
  where m = numFilas p
        n = numColumnas q

```

```

-----
-- Traspuestas y simétricas                                     --
-----

```

```

-----
-- Ejercicio 14. Definir la función
--   traspuesta :: Num a => Matriz a -> Matriz a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> traspuesta p
--   array ((1,1),(3,2)) [((1,1),5),((1,2),3),
--                          ((2,1),1),((2,2),2),
--                          ((3,1),0),((3,2),6)]
--   ghci> matrizLista (traspuesta p)
--   [[5,3],[1,2],[0,6]]
-----

```

```

traspuesta :: Num a => Matriz a -> Matriz a
traspuesta p =
  array ((1,1),(n,m))
    [((i,j), p!(j,i)) | i <- [1..n], j <- [1..m]]
  where (m,n) = dimension p

```

```

-----
-- Ejercicio 15. Definir la función
--   esCuadrada :: Num a => Matriz a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> esCuadrada p
--   False

```

```
-- ghci> let q = listaMatriz [[5,1],[3,2]]
-- ghci> esCuadrada q
-- True
```

```
-----
esCuadrada :: Num a => Matriz a -> Bool
esCuadrada x = numFilas x == numColumnas x
```

```
-----
-- Ejercicio 16. Definir la función
-- esSimetrica :: (Num a, Eq a) => Matriz a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
-- ghci> esSimetrica p
-- True
-- ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
-- ghci> esSimetrica q
-- False
```

```
-----
esSimetrica :: (Num a, Eq a) => Matriz a -> Bool
esSimetrica x = x == traspuesta x
```

```
-----
-- Diagonales de una matriz -----
```

```
-----
-- Ejercicio 17. Definir la función
-- diagonalPral :: Num a => Matriz a -> Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
-- ghci> diagonalPral p
-- array (1,2) [(1,5),(2,2)]
-- ghci> vectorLista (diagonalPral p)
-- [5,2]
```

```

diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
  where n = min (numFilas p) (numColumnas p)

```

```

-----
-- Ejercicio 18. Definir la función
--   diagonalSec :: Num a => Matriz a -> Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--   ghci> diagonalSec p
--   array (1,2) [(1,1),(2,3)]
--   ghci> vectorLista (diagonalSec p)
--   [1,3]
--   ghci> let q = traspuesta p
--   ghci> matrizLista q
--   [[5,3],[1,2],[0,6]]
--   ghci> vectorLista (diagonalSec q)
--   [3,1]
-----

```

```

diagonalSec :: Num a => Matriz a -> Vector a
diagonalSec p = array (1,n) [(i,p!(i,n+1-i)) | i <- [1..n]]
  where n = min (numFilas p) (numColumnas p)

```

```

-----
-- Submatrices
-----

```

```

-----
-- Ejercicio 19. Definir la función
--   submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> submatriz 2 3 p
--   array ((1,1),(2,2)) [((1,1),5),((1,2),1),((2,1),4),((2,2),6)]
--   ghci> matrizLista (submatriz 2 3 p)
--   [[5,1],[4,6]]
-----

```

```

submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
  array ((1,1), (m-1,n -1))
    [((k,l), p ! f k l) | k <- [1..m-1], l <- [1.. n-1]]
  where (m,n) = dimension p
         f k l | k < i  && l < j  = (k,l)
               | k >= i && l < j  = (k+1,l)
               | k < i  && l >= j = (k,l+1)
               | otherwise      = (k+1,l+1)

-----
-- Transformaciones elementales
-----

-----
-- Ejercicio 20. Definir la función
--   intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> intercambiaFilas 1 3 p
--   array ((1,1),(3,3)) [((1,1),4),((1,2),6),((1,3),9),
--                        ((2,1),3),((2,2),2),((2,3),6),
--                        ((3,1),5),((3,2),1),((3,3),0)]
--   ghci> matrizLista (intercambiaFilas 1 3 p)
--   [[4,6,9],[3,2,6],[5,1,0]]
-----

intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaFilas k l p =
  array ((1,1), (m,n))
    [((i,j), p! f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
         f i j | i == k    = (l,j)
               | i == l    = (k,j)
               | otherwise = (i,j)

-----
-- Ejercicio 21. Definir la función

```

```
--   intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
--   tal que (intercambiaColumnas k l p) es la matriz obtenida
--   intercambiando las columnas k y l de la matriz p. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> matrizLista (intercambiaColumnas 1 3 p)
--   [[0,1,5],[6,2,3],[9,6,4]]
```

```
intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaColumnas k l p =
  array ((1,1), (m,n))
    [((i,j), p ! f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | j == k    = (i,l)
              | j == l    = (i,k)
              | otherwise = (i,j)
```

```
-- Ejercicio 22. Definir la función
--   multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
--   tal que (multFilaPor k x p) es a matriz obtenida multiplicando la
--   fila k de la matriz p por el número x. Por ejemplo,
--   ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--   ghci> matrizLista (multFilaPor 2 3 p)
--   [[5,1,0],[9,6,18],[4,6,9]]
```

```
multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
multFilaPor k x p =
  array ((1,1), (m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = x*(p!(i,j))
              | otherwise = p!(i,j)
```

```
-- Ejercicio 23. Definir la función
--   sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
--   tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
--   a la fila k d la matriz p. Por ejemplo,
```

```
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> matrizLista (sumaFilaFila 2 3 p)
-- [[5,1,0],[7,8,15],[4,6,9]]
```

```
sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
```

```
sumaFilaFila k l p =
  array ((1,1), (m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = p!(i,j) + p!(l,j)
              | otherwise = p!(i,j)
```

```
-- Ejercicio 24. Definir la función
```

```
-- sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> matrizLista (sumaFilaPor 2 3 10 p)
-- [[5,1,0],[43,62,96],[4,6,9]]
```

```
sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
```

```
sumaFilaPor k l x p =
  array ((1,1), (m,n))
    [((i,j), f i j) | i <- [1..m], j <- [1..n]]
  where (m,n) = dimension p
        f i j | i == k    = p!(i,j) + x*p!(l,j)
              | otherwise = p!(i,j)
```

```
-- Triangularización de matrices
```

```
-- Ejercicio 25. Definir la función
```

```
-- buscaIndiceDesde :: (Num a, Eq a) =>
--                   Matriz a -> Int -> Int -> Maybe Int
-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
```



```
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> buscaIndiceDesde p 3 2
-- Just 2
-- ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
-- ghci> buscaIndiceDesde q 3 2
-- Nothing
```

```
-----
buscaIndiceDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
  | null xs    = Nothing
  | otherwise  = Just (head xs)
  where xs = [k | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]
```

```
-----
-- Ejercicio 26. Definir la función
-- buscaPivoteDesde :: (Num a, Eq a) =>
--                   Matriz a -> Int -> Int -> Maybe a
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> buscaPivoteDesde p 3 2
-- Just 6
-- ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
-- ghci> buscaPivoteDesde q 3 2
-- Nothing
```

```
-----
buscaPivoteDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
  | null xs    = Nothing
  | otherwise  = Just (head xs)
  where xs = [y | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]
```

```
-----
-- Ejercicio 27. Definir la función
-- anuladaColumnaDesde :: (Num a, Eq a) =>
--                       Int -> Int -> Matriz a -> Bool
```

```
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
-- ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
-- ghci> anuladaColumnaDesde q 3 2
-- True
-- ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
-- ghci> anuladaColumnaDesde p 3 2
-- False
```

```
-----
anuladaColumnaDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
  buscaIndiceDesde p j (i+1) == Nothing
```

```
-----
-- Ejercicio 28. Definir la función
-- anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
-- Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
-- ghci> let p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matriz Double
-- ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
-- [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]
```

```
-----
anulaEltoColumnaDesde :: (Fractional a, Eq a) =>
  Matriz a -> Int -> Int -> Matriz a
anulaEltoColumnaDesde p j i =
  sumaFilaPor l i (-(p!(l,j)/a)) p
  where Just l = buscaIndiceDesde p j (i+1)
        a      = p!(i,j)
```

```
-----
-- Ejercicio 29. Definir la función
-- anulaColumnaDesde :: (Fractional a, Eq a) =>
-- Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando
-- todos los elementos de la columna j de la matriz p por debajo del la
```

```
-- posición (i,j) (se supone que el elemnto p_(i,j) es no nulo). Por
-- ejemplo,
-- ghci> let p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matriz Double
-- ghci> matrizLista (anulaColumnaDesde p 2 1)
-- [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
-- ghci> let p = listaMatriz [[4,5],[2,7%2],[6,10]]
-- ghci> matrizLista (anulaColumnaDesde p 1 1)
-- [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
-----
```

```
anulaColumnaDesde :: (Fractional a, Eq a) =>
Matriz a -> Int -> Int -> Matriz a
```

```
anulaColumnaDesde p j i
  | anuladaColumnaDesde p j i = p
  | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i
-----
```

```
-- Algoritmo de Gauss para triangularizar matrices -----
```

```
-- Ejercicio 30. Definir la función
-- elementosNoNulosColDesde :: (Num a, Eq a) =>
--                               Matriz a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
-- ghci> let p = listaMatriz [[3,2],[5,1],[0,4]]
-- ghci> elementosNoNulosColDesde p 1 2
-- [5]
-----
```

```
elementosNoNulosColDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> [a]
elementosNoNulosColDesde p j i =
  [x | ((k,j'),x) <- assocs p, x /= 0, j' == j, k >= i]
-----
```

```
-- Ejercicio 31. Definir la función
-- existeColNoNulaDesde :: (Num a, Eq a) =>
--                               Matriz a -> Int -> Int -> Bool
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-----
```

```
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la j; es decir, si la submatriz de p obtenida eliminando
-- las i-1 primeras filas y las j-1 primeras columnas es no nula. Por
-- ejemplo,
-- ghci> let p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
-- ghci> existeColNoNulaDesde p 2 2
-- False
-- ghci> let q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
-- ghci> existeColNoNulaDesde q 2 2
```

```
-----
existeColNoNulaDesde :: (Num a, Eq a) => Matriz a -> Int -> Int -> Bool
existeColNoNulaDesde p j i =
  or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
  where n = numColumnas p
```

```
-----
-- Ejercicio 32. Definir la función
-- menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
--                               Matriz a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
-- ghci> let p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
-- ghci> menorIndiceColNoNulaDesde p 2 2
-- Just 2
-- ghci> let q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
-- ghci> menorIndiceColNoNulaDesde q 2 2
-- Just 3
-- ghci> let r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
-- ghci> menorIndiceColNoNulaDesde r 2 2
-- Nothing
```

```
-----
menorIndiceColNoNulaDesde :: (Num a, Eq a) =>
                               Matriz a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
  | null js    = Nothing
  | otherwise = Just (head js)
  where n     = numColumnas p
```

```
js = [j' | j' <- [j..n],
      not (null (elementosNoNulosColDesde p j' i))]
```

```
-----
-- Ejercicio 33. Definir la función
--   gaussAux :: (Fractional a, Eq a) =>
--             Matriz a -> Int -> Int -> Matriz a
-- tal que (gauss p) es la matriz que en el que las i-1 primeras filas y
-- las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
--   1. Si la dimensión de p es (i,j), entonces p.
--   2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--      primeras columnas es nulas, entonces p.
--   3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
--   3.1. j' la primera columna a partir de la j donde p tiene
--        algún elemento no nulo a partir de la fila i,
--   3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--        de p,
--   3.3. i' la primera fila a partir de la i donde la columna j de
--        p1 tiene un elemento no nulo,
--   3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--        la matriz p1 y
--   3.5. p' la matriz obtenida anulando todos los elementos de la
--        columna j de p2 por debajo de la fila i.
-- Por ejemplo,
--   ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
--   ghci> matrizLista (gaussAux p 2 2)
--   [[1.0,2.0,3.0],[1.0,2.0,4.0],[2.0,0.0,1.0]]
-----
```

```
gaussAux :: (Fractional a, Eq a) => Matriz a -> Int -> Int -> Matriz a
gaussAux p i j
  | dimension p == (i,j)           = p -- 1
  | not (existeColNoNulaDesde p j i) = p -- 2
  | otherwise                       = gaussAux p' (i+1) (j+1) -- 3
  where Just j' = menorIndiceColNoNulaDesde p j i -- 3.1
        p1     = intercambiaColumnas j j' p -- 3.2
        Just i' = buscaIndiceDesde p1 j i -- 3.3
        p2     = intercambiaFilas i i' p1 -- 3.4
        p'     = anulaColumnaDesde p2 j i -- 3.5
```

```

-----
-- Ejercicio 34. Definir la función
--   gauss :: (Fractional a, Eq a) => Matriz a -> Matriz a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
--   ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--   ghci> gauss p
--   array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                          ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                          ((3,1),0.0),((3,2),0.0),((3,3),0.0)]
--   ghci> matrizLista (gauss p)
--   [[1.0,3.0,2.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
--   ghci> let p = listaMatriz [[3.0,2,3],[1,2,4],[1,2,5]]
--   ghci> matrizLista (gauss p)
--   [[3.0,2.0,3.0],[0.0,1.3333333333333335,3.0],[0.0,0.0,1.0]]
--   ghci> let p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
--   ghci> matrizLista (gauss p)
--   [[3 % 1,2 % 1,3 % 1],[0 % 1,4 % 3,3 % 1],[0 % 1,0 % 1,1 % 1]]
--   ghci> let p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
--   ghci> matrizLista (gauss p)
--   [[1.0,3.0,0.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
-----

```

```

gauss :: (Fractional a, Eq a) => Matriz a -> Matriz a
gauss p = gaussAux p 1 1

```

```

-----
-- Determinante
-----

```

```

-----
-- Ejercicio 35. Definir la función
--   determinante :: (Fractional a, Eq a) => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--   determinante (listaMatriz [[1,2,3],[1,2,4],[1,2,5]]) == 0.0
--   determinante (listaMatriz [[1,2,3],[1,3,4],[1,2,5]]) == 2.0
--   determinante (listaMatriz [[0,1],[1,0]]) == -1.0
-----

```







## Relación 27

# Aplicaciones de la programación funcional con listas infinitas

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se estudia distintas aplicaciones de la programación  
-- funcional que usan listas infinitas  
-- * definición alternativa de la sucesión de Hamming estudiada en el  
-- tema 11,  
-- * propiedades de la sucesión de Hamming,  
-- * problemas 10 y 12 del proyecto Euler y  
-- * numero de pares de naturales en un círculo.  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
import Test.QuickCheck  
import Data.List  
  
-- -----  
-- Ejercicio 1.1. Definir la función  
-- divisoresEn :: Integer -> [Integer] -> Bool  
-- tal que (divisoresEn x ys) se verifica si x puede expresarse como un  
-- producto de potencias de elementos de ys. Por ejemplo,  
-- divisoresEn 12 [2,3,5] == True
```

```
-- divisoresEn 14 [2,3,5] == False
-----

divisoresEn :: Integer -> [Integer] -> Bool
divisoresEn 1 _ = True
divisoresEn x [] = False
divisoresEn x (y:ys) | mod x y == 0 = divisoresEn (div x y) (y:ys)
                    | otherwise = divisoresEn x ys
```

```
-- Ejercicio 1.2. Los números de Hamming forman una sucesión
-- estrictamente creciente de números que cumplen las siguientes
-- condiciones:
-- 1. El número 1 está en la sucesión.
-- 2. Si x está en la sucesión, entonces 2x, 3x y 5x también están.
-- 3. Ningún otro número está en la sucesión.
-- Definir, usando divisoresEn, la constante
-- hamming :: [Integer]
-- tal que hamming es la sucesión de Hamming. Por ejemplo,
-- take 12 hamming == [1,2,3,4,5,6,8,9,10,12,15,16]
-----
```

```
hamming :: [Integer]
hamming = [x | x <- [1..], divisoresEn x [2,3,5]]
```

```
-- Ejercicio 1.3. Definir la función
-- cantidadHammingMenores :: Integer -> Int
-- tal que (cantidadHammingMenores x) es la cantidad de números de
-- Hamming menores que x. Por ejemplo,
-- cantidadHammingMenores 6 == 5
-- cantidadHammingMenores 7 == 6
-- cantidadHammingMenores 8 == 6
-----
```

```
cantidadHammingMenores :: Integer -> Int
cantidadHammingMenores x = length (takeWhile (<x) hamming)
```

```
-- Ejercicio 1.4. Definir la función
```

```

-- siguienteHamming :: Integer -> Integer
-- tal que (siguienteHamming x) es el menor número de la sucesión de
-- Hamming mayor que x. Por ejemplo,
-- siguienteHamming 6 == 8
-- siguienteHamming 21 == 24

```

```

siguienteHamming :: Integer -> Integer
siguienteHamming x = head (dropWhile (<=x) hamming)

```

```

-----
-- Ejercicio 1.5. Definir la función
-- huecoHamming :: Integer -> [(Integer,Integer)]
-- tal que (huecoHamming n) es la lista de pares de números consecutivos
-- en la sucesión de Hamming cuya distancia es mayor que n. Por ejemplo,
-- take 4 (huecoHamming 2) == [(12,15),(20,24),(27,30),(32,36)]
-- take 3 (huecoHamming 2) == [(12,15),(20,24),(27,30)]
-- take 2 (huecoHamming 3) == [(20,24),(32,36)]
-- head (huecoHamming 10) == (108,120)
-- head (huecoHamming 1000) == (34992,36000)

```

```

huecoHamming :: Integer -> [(Integer,Integer)]
huecoHamming n = [(x,y) | x <- hamming,
                        let y = siguienteHamming x,
                            y-x > n]

```

```

-----
-- Ejercicio 1.6. Comprobar con QuickCheck que para todo n, existen
-- pares de números consecutivos en la sucesión de Hamming cuya
-- distancia es mayor o igual que n.

```

```

-- La propiedad es
prop_Hamming :: Integer -> Bool
prop_Hamming n = huecoHamming n' /= []
                where n' = abs n

```

```

-- La comprobación es
-- *Main> quickCheck prop_Hamming

```

```

--      OK, passed 100 tests.

-----

-- Ejercicio 2. (Problema 10 del Proyecto Euler)
-- Definir la función
--      sumaPrimoMenores :: Integer -> Integer
-- tal que (sumaPrimoMenores n) es la suma de los primos menores que
-- n. Por ejemplo,
--      sumaPrimoMenores 10 == 17
--      sumaPrimoMenores 7  == 10
-----

-- La definición es
sumaPrimoMenores :: Integer -> Integer
sumaPrimoMenores n = sumaMenores n primos 0
  where sumaMenores n (x:xs) a | n <= x    = a
                                | otherwise = sumaMenores n xs (a+x)

-- primos es la lista de los número primos obtenida mediante la criba de
-- Eratóstenes. Por ejemplo,
--      primos => [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,...]
primos :: [Integer]
primos = criba [2..]
  where criba (p:ps) = p : criba [n | n<-ps, mod n p /= 0]

-----

-- Ejercicio 3. (Problema 12 del Proyecto Euler)
-- La sucesión de los números triangulares se obtiene sumando los
-- números naturales. Así, el 7º número triangular es
--      1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.
-- Los primeros 10 números triangulares son
--      1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
-- Los divisores de los primeros 7 números triangulares son:
--      1: 1
--      3: 1,3
--      6: 1,2,3,6
--     10: 1,2,5,10
--     15: 1,3,5,15
--     21: 1,3,7,21
--     28: 1,2,4,7,14,28

```

```
-- Como se puede observar, 28 es el menor número triangular con más de 5
-- divisores.
--
-- Definir la función
--   euler12 :: Int -> Integer
-- tal que (euler12 n) es el menor número triangular con más de n
-- divisores. Por ejemplo,
--   euler12 5 == 28
-----

euler12 :: Int -> Integer
euler12 n = head [x | x <- triangulares, nDivisores x > n]

-- triangulares es la lista de los números triangulares
--   take 10 triangulares => [1,3,6,10,15,21,28,36,45,55]
triangulares :: [Integer]
triangulares = 1:[x+y | (x,y) <- zip [2..] triangulares]

-- Otra definición de triangulares es
triangulares' :: [Integer]
triangulares' = scanl (+) 1 [2..]

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--   divisores 28 == [1,2,4,7,14,28]
divisores :: Integer -> [Integer]
divisores x = [y | y <- [1..x], mod x y == 0]

-- (nDivisores n) es el número de los divisores de n. Por ejemplo,
--   nDivisores 28 == 6
nDivisores :: Integer -> Int
nDivisores x = length (divisores x)
-----

-- Ejercicio 4. Definir la función
--   circulo :: Int -> Int
-- tal que (circulo n) es el la cantidad de pares de números naturales
-- (x,y) que se encuentran dentro del círculo de radio n. Por ejemplo,
--   circulo 3 == 9
--   circulo 4 == 15
--   circulo 5 == 22
```

---

```
circulo :: Int -> Int
circulo n = length [(x,y) | x <- [0..n], y <- [0..n], x^2+y^2 < n^2]

-- La eficiencia puede mejorarse con
circulo' :: Int -> Int
circulo' n = length [(x,y) | x <- [0..m], y <- [0..m], x^2+y^2 < n^2]
    where m = raizCuadradaEntera n

-- (raizCuadradaEntera n) es la parte entera de la raíz cuadrada de
-- n. Por ejemplo,
--     raizCuadradaEntera 17 == 4
raizCuadradaEntera :: Int -> Int
raizCuadradaEntera n = truncate (sqrt (fromIntegral n))
```

# Relación 28

## Ejercicios complementarios

```
-----  
-- Introducción --  
-----  
  
-- En esta relación se recogen ejercicios variados de programación  
-- funcional que complementan las relaciones anteriores.  
--  
-- Los problemas incluidos son:  
-- * el de los cuadrados mágicos,  
-- * el de la codificación de mensajes,  
-- * el de pertenencia al rango de una función creciente,  
-- * el de los puntos cercanos,  
-- * el de la evaluación de expresiones aritméticas y  
-- * el del menor número con todos los dígitos en la factorización de su  
-- factorial.  
  
-----  
-- Importación de librerías --  
-----  
  
import Data.List  
import Data.Char  
  
-----  
-- Ejercicio 1. Una matriz cuadrada representa un cuadrado mágico de  
-- orden n el conjunto de sus elementos es {1,2,...,n^2}, las sumas de  
-- cada una de sus filas, columnas y dos diagonales principales
```

```

-- coinciden. Por ejemplo,
--   / 2  9  4  \
--   | 7  5  3  |
--   \ 6  1  8  /
-- es un cuadrado mágico de orden 3, ya que el conjunto de sus
-- elementos es {1,2,...,9}, todas sus filas, columnas y diagonales
-- principales suman 15.
--
-- Representaremos una matriz numérica como una lista cuyos elementos son
-- las filas del cuadrado, en forma de listas. Por ejemplo, el cuadrado
-- anterior vendría representado por la siguiente lista:
--   [[2, 9, 4], [7, 5, 3], [6, 1, 8]]
-- En los distintos apartados de este ejercicio se definirán funciones
-- cuyo objetivo es decidir si una matriz representa un cuadrado
-- mágico y construirlos.
-- -----
--
-- Ejercicio 1.1. Definir la función
--   traspuesta :: [[a]] -> [[a]]
-- tal que (traspuesta m) es la traspuesta de la matriz m. Por ejemplo,
--   traspuesta [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]
--   traspuesta [[1,4],[2,5],[3,6]] == [[1,2,3],[4,5,6]]
-- -----

traspuesta :: [[a]] -> [[a]]
traspuesta []           = []
traspuesta ([:xss])    = traspuesta xss
traspuesta ((x:xs):xss) =
  (x:[h | (h:_) <- xss]) : traspuesta (xs : [t | (_:t) <- xss])

-- Una definición equivalente es
traspuesta' :: [[a]] -> [[a]]
traspuesta' = transpose

-- -----
-- Ejercicio 1.2. Definir la función
--   sumas_de_filas :: Num a => [[a]] -> [a]
-- tal que (sumas_de_filas xss) es la lista de las sumas de las filas de
-- la matriz xss. Por ejemplo,

```



```
--      sumas_de_filas [[2,4,0],[7,1,3],[6,1,8]] == [6,11,15]
```

```
sumas_de_filas :: Num a => [[a]] -> [a]
sumas_de_filas = map sum
```

```
-- Ejercicio 1.3. Definir la función
```

```
--      sumas_de_columnas :: Num a => [[a]] -> [a]
-- tal que (sumas_de_columnas xss) es la lista de las sumas de las
-- columnas de la matriz xss. Por ejemplo,
--      sumas_de_filas [[2,4,0],[7,1,3],[6,1,8]] == [6,11,15]
```

```
sumas_de_columnas :: Num a => [[a]] -> [a]
sumas_de_columnas = sumas_de_filas . traspuesta
```

```
-- Ejercicio 1.4. Definir la función
```

```
--      diagonal_pral :: [[a]] -> [a]
-- tal que (diagonal_pral m) es la diagonal principal de la matriz m. Por
-- ejemplo,
--      diagonal_pral [[3,5,2],[4,7,1],[6,9,0]] == [3,7,0]
--      diagonal_pral [[3,5,2],[4,7,1]]         == [3,7]
```

```
diagonal_pral :: [[a]] -> [a]
diagonal_pral ((x1:_):xs) = x1 : diagonal_pral [tail x | x <- xs]
diagonal_pral _           = []
```

```
-- Ejercicio 1.5. Definir la función
```

```
--      diagonal_sec :: [[a]] -> [a]
-- tal que (diagonal_sec m) es la diagonal secundaria de la matriz m
-- Por ejemplo,
--      diagonal_sec [[3,5,2],[4,7,1],[6,9,0]] == [6,7,2]
--      diagonal_sec [[3,5,2],[4,7,1]]         == [4,5]
```

```
diagonal_sec :: [[a]] -> [a]
```

```
diagonal_sec = diagonal_pral . reverse
```

```
-----
-- Ejercicio 1.6. Definir la función
--   todos_iguales :: Eq a => [a] -> Bool
-- tal que (todos_iguales xs) se verifica si todos los elementos de xs
-- son iguales. Por ejemplo,
--   todos_iguales [2,2,2] == True
--   todos_iguales [2,3,2] == False
-----
```

```
todos_iguales :: Eq a => [a] -> Bool
todos_iguales (x:y:ys) = x == y && todos_iguales (y:ys)
todos_iguales _ = True
```

```
-----
-- Ejercicio 1.7. Definir la función
--   matrizCuadrada :: [[Int]] -> Bool
-- tal que (matrizCuadrada xss) se verifica si xss es una matriz
-- cuadrada; es decir, xss es una lista de n elementos y cada elemento
-- de xss es una lista de n elementos. Por ejemplo,
--   matrizCuadrada [[7,3],[1,5]] == True
--   matrizCuadrada [[7,3,1],[1,5,2]] == False
-----
```

```
matrizCuadrada :: [[Int]] -> Bool
matrizCuadrada xss =
  and [length xs == n | xs <- xss]
  where n = length xss
```

```
-----
-- Ejercicio 1.9. Definir la función
--   elementos :: [[a]] -> [a]
-- tal que (elementos xss) es la lista de los elementos de xss. Por
-- ejemplo,
--   elementos [[7,3],[1,5],[3,5]] == [7,3,1,5,3,5]
-----
```

```
elementos :: [[a]] -> [a]
elementos = concat
```

```

-----
-- Ejercicio 1.10. Definir por recursión la función
--   borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando la primera
-- ocurrencia de x en la lista xs. Por ejemplo,
--   borra 1 [1,2,1] == [2,1]
--   borra 3 [1,2,1] == [1,2,1]
-----

borra :: Eq a => a -> [a] -> [a]
borra x [] = []
borra x (y:ys) | x == y = ys
                | otherwise = y : borra x ys
-----

-- Ejercicio 1.11. Definir por recursión la función
--   esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutación de
-- ys. Por ejemplo,
--   esPermutacion [1,2,1] [2,1,1] == True
--   esPermutacion [1,2,1] [1,2,2] == False
-----

esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion [] [] = True
esPermutacion [] (y:ys) = False
esPermutacion (x:xs) ys = elem x ys && esPermutacion xs (borra x ys)
-----

-- Ejercicio 1.12. Definir la función
--   cuadradoMagico :: Num a => [[a]] -> Bool
-- tal que (cuadradoMagico xss) se verifica si xss es un cuadrado
-- mágico. Por ejemplo,
--   ghci> cuadradoMagico [[2,9,4],[7,5,3],[6,1,8]]
--   True
--   ghci> cuadradoMagico [[1,2,3],[4,5,6],[7,8,9]]
--   False
--   ghci> cuadradoMagico [[1,1],[1,1]]
--   False

```

```
-- ghci> cuadradoMagico [[5,8,12,9],[16,13,1,4],[2,10,7,15],[11,3,14,6]]
-- False
```

```
-----
cuadradoMagico xss =
  matrizCuadrada xss &&
  esPermutacion (elementos xss) [1..(length xss)^2] &&
  todos_iguales ((sumas_de_filas xss) ++
                (sumas_de_columnas xss) ++
                [(sum (diagonal_pral xss)),
                 (sum (diagonal_sec xss))])
```

```
-----
-- Ejercicio 1.13. Definir la función
--   matriz :: Int -> [a] -> [[a]]
-- tal que (matriz n xs) es la matriz cuadrada de orden nxn cuyos
-- elementos son xs (se supone que la longitud de xs es n^2). Por
-- ejemplo,
--   matriz 3 [1..9] == [[1,2,3],[4,5,6],[7,8,9]]
```

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

```
-----
-- Ejercicio 1.14. Definir la función
--   cuadradosMagicos :: Int -> [[[Int]]]
-- tal que (cuadradosMagicos n) es la lista de los cuadrados mágicos de
-- orden nxn. Por ejemplo,
--   ghci> take 2 (cuadradosMagicos 3)
--   [[[2,9,4],[7,5,3],[6,1,8]], [[2,7,6],[9,5,1],[4,3,8]]]
```

```
-----
cuadradosMagicos :: Int -> [[[Int]]]
cuadradosMagicos n =
  [m | xs <- permutations [1..n^2],
      let m = matriz n xs,
          cuadradoMagico m]
```

```

-----
-- Ejercicio 1.15. Los cuadrados mágicos de orden 3 tienen la forma
--   +---+---+---+
--   | a | b | c |
--   +---+---+---+
--   | d | e | f |
--   +---+---+---+
--   | g | h | i |
--   +---+---+---+
-- y se pueden construir como sigue:
--   * a es un elemento de [1..9],
--   * b es un elemento de los restantes (es decir, de [1..9] \\ [a]),
--   * c es un elemento de los restantes,
--   * a+b+c tiene que ser igual a 15,
--   * d es un elemento de los restantes,
--   * g es un elemento de los restantes,
--   * a+d+g tiene que ser igual a 15,
-- y así sucesivamente.
--
-- Definir la función
--   cuadradosMagicos_3 :: [[[Int]]]
-- tal que cuadradosMagicos_3 es la lista de los cuadrados mágicos de
-- orden 3 contruidos usando el proceso anterior. Por ejemplo,
--   ghci> take 2 cuadradosMagicos_3
--   [[ [2,7,6], [9,5,1], [4,3,8] ], [ [2,9,4], [7,5,3], [6,1,8] ] ]
-----

```

```

cuadradosMagicos_3 :: [[[Int]]]
cuadradosMagicos_3 =
  [[ [a,b,c],[d,e,f],[g,h,i] |
    a <- [1..9],
    b <- [1..9] \\ [a],
    c <- [1..9] \\ [a,b],
    a+b+c == 15,
    d <- [1..9] \\ [a,b,c],
    g <- [1..9] \\ [a,b,c,d],
    a+d+g == 15,
    e <- [1..9] \\ [a,b,c,d,g],
    c+e+g == 15,
    i <- [1..9] \\ [a,b,c,d,g,e],

```

```

a+e+i == 15,
f <- [1..9] \\ [a,b,c,d,g,e,i],
h <- [1..9] \\ [a,b,c,d,g,e,i,f],
c+f+i == 15,
d+e+f == 15]

-----
-- Ejercicio 1.16. Comprobar que cuadradosMagicos_3 es el mismo conjunto
-- que (cuadradosMagicos 3).
-----

-- La comprobación es
--   ghci> esPermutacion cuadradosMagicos_3 (cuadradosMagicos 3)
--   True

-----
-- Ejercicio 1.17. Comparar los tiempos utilizados en calcular
-- cuadradosMagicos_3 y (cuadradosMagicos 3).
-----

-- La comparación es
--   ghci> :set +s
--   ghci> cuadradosMagicos_3
--   [[[2,7,6],[9,5,1],[4,3,8]],[[2,9,4],[7,5,3],[6,1,8]]], ...
--   (0.02 secs, 532348 bytes)
--   ghci> (cuadradosMagicos 3)
--   [[[2,9,4],[7,5,3],[6,1,8]],[[2,7,6],[9,5,1],[4,3,8]]], ...
--   (50.32 secs, 2616351124 bytes)

-----
-- Ejercicio 2. Se desea definir una función que codifique mensajes
-- tales como
--   "eres lo que piensas"
-- del siguiente modo:
-- (a) se separa la cadena en la lista de sus palabras:
--   ["eres","lo","que","piensas"]
-- (b) se cuenta las letras de cada palabra:
--   [4,2,3,7]
-- (c) se une todas las palabras:
--   "eresloquepiensas"

```

```

-- (d) se reagrupa las letras de 4 en 4, dejando el último grupo con el
--     resto:
--     ["eres","loqu","epie","nsas"]
-- (e) se invierte cada palabra:
--     ["sere","uqol","eipe","sasn"]
-- (f) se une todas las palabras:
--     "sereuqoleipesasn"
-- (g) se reagrupan tal como indica la inversa de la lista del apartado
--     (b):
--     ["sereuqo","lei","pe","sasn"]
-- (h) se crea una frase con las palabras anteriores separadas por un
--     espacio en blanco
--     "sereuqo lei pe sasn"
--     obteniendo así el mensaje codificado.
--
-- En los distintos apartados de este ejercicio se definirá el anterior
-- proceso de codificación.

```

```

-----
-- Ejercicio 2.1. Definir la función
--     divide :: (a -> Bool) -> [a] -> ([a], [a])
-- tal que (divide p xs) es el par (ys,zs) donde ys es el mayor prefijo
-- de xs cuyos elementos cumplen p y zs es la lista de los restantes
-- elementos de xs. Por ejemplo,
--     divide (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
--     divide (< 9) [1,2,3]           == ([1,2,3],[ ])
--     divide (< 0) [1,2,3]           == ([],[1,2,3])

```

```

divide :: (a -> Bool) -> [a] -> ([a], [a])
divide p xs = (takeWhile p xs, dropWhile p xs)

```

```

-- Es equivalente a la predefinida span
divide' :: (a -> Bool) -> [a] -> ([a], [a])
divide' = span

```

```

-----
-- Ejercicio 2.2. Definir la función
--     palabras :: String -> [String]

```

```
-- tal que (palabras cs) es la lista de las palabras de la cadena cs.
-- Por ejemplo,
-- palabras "eres lo que piensas" == ["eres","lo","que","piensas"]
-----
```

```
palabras :: String -> [String]
palabras [] = []
palabras cs = cs1 : palabras cs2
  where cs' = dropWhile (==' ') cs
        (cs1,cs2) = divide (/=' ') cs'
```

```
-- Es equivalente a la predefinida words
```

```
palabras' :: String -> [String]
palabras' = words
-----
```

```
-- Ejercicio 2.3. Definir la función
-- longitudes :: [[a]] -> [Int]
-- tal que (longitudes xss) es la lista de las longitudes de los
-- elementos xss. Por ejemplo,
-- longitudes ["eres","lo","que","piensas"] == [4,2,3,7]
-----
```

```
longitudes :: [[a]] -> [Int]
longitudes = map length
-----
```

```
-- Ejercicio 2.4. Definir la función
-- une :: [[a]] -> [a]
-- tal que (une xss) es la lista obtenida uniendo los elementos de
-- xss. Por ejemplo,
-- une ["eres","lo","que","piensas"] == "eresloquepiensas"
-----
```

```
une :: [[a]] -> [a]
une = concat
-----
```

```
-- Ejercicio 2.5. Definir la función
-- reagrupa :: [a] -> [[a]]
-----
```



```
-- tal que (reagrupa xs) es la lista obtenida agrupando los elementos de
-- xs de 4 en 4. Por ejemplo,
--   reagrupa "eresloquepiensas" == ["eres","loqu","epie","nsas"]
--   reagrupa "erestu"           == ["eres","tu"]
-----
```

```
reagrupa :: [a] -> [[a]]
reagrupa [] = []
reagrupa xs = take 4 xs : reagrupa (drop 4 xs)
```

```
-- Ejercicio 2.6. Definir la función
--   inversas :: [[a]] -> [[a]]
-- tal que (inversas xss) es la lista obtenida invirtiendo los elementos
-- de xss. Por ejemplo,
--   ghci> inversas ["eres","loqu","epie","nsas"]
--   ["sere","uqol","eipe","sasn"]
--   ghci> une (inversas ["eres","loqu","epie","nsas"])
--   "sereuqoleipesasn"
-----
```

```
inversas :: [[a]] -> [[a]]
inversas = map reverse
```

```
-- Ejercicio 2.7. Definir la función
--   agrupa :: [a] -> [Int] -> [[a]]
-- tal que (agrupa xs ns) es la lista obtenida agrupando los elementos
-- de xs según las longitudes indicadas en ns. Por ejemplo,
--   ghci> agrupa "sereuqoleipesasn" [7,3,2,4]
--   ["sereuqo","lei","pe","sasn"]
-----
```

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

```
-- Ejercicio 2.8. Definir la función
--   frase :: [String] -> String
```

```
-- tal que (frase xs) es la frase obtenida las palabras de xs dejando un
-- espacio en blanco entre ellas. Por ejemplo,
--   frase ["sereuqo","lei","pe","sasn"] == "sereuqo lei pe sasn"
```

```
frase :: [String] -> String
frase [x]      = x
frase (x:xs)  = x ++ " " ++ frase xs
frase []      = []
```

```
-- La función frase es equivalente a unwords.
```

```
frase' :: [String] -> String
frase' = unwords
```

```
-- Ejercicio 2.9. Definir la función
```

```
--   clave :: String -> String
-- que realice el proceso completo. Por ejemplo,
--   clave "eres lo que piensas" == "sereuqo lei pe sasn"
```

```
clave :: String -> String
clave xss = frase (agrupa (une (inversas (reagrupa (une ps))))
                  (reverse (longitudes ps)))
  where ps = palabras xss
```

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

```
--   perteneceRango :: Int -> (Int -> Int) -> Bool
-- tal que (perteneceRango x f) se verifica si x pertenece al rango de
-- la función f, suponiendo que f es una función creciente cuyo dominio
-- es el conjunto de los números naturales. Por ejemplo,
--   perteneceRango 5 (\x -> 2*x+1) == True
--   perteneceRango 1234 (\x -> 2*x+1) == False
```

```
perteneceRango :: Int -> (Int -> Int) -> Bool
perteneceRango y f = elem y (takeWhile (<=y) (imagenes f))
  where imagenes f = [f x | x <- [0..]]
```

```

-----
-- Ejercicio 4. Los puntos del plano se pueden representar por pares de
-- números como se indica a continuación
--   type Punto = (Double,Double)
-- Definir la función
--   cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
-- tal que (cercanos ps qs) es un par de puntos, el primero de ps y el
-- segundo de qs, que son los más cercanos (es decir, no hay otro par
-- (p',q') con p' en ps y q' en qs tales que la distancia entre p' y q'
-- sea menor que la que hay entre p y q). Por ejemplo,
--   cercanos [(2,5),(3,6)] [(4,3),(1,0),(7,9)] == ((2.0,5.0),(4.0,3.0))
-----

```

```

type Punto = (Double,Double)

```

```

cercanos :: [Punto] -> [Punto] -> (Punto,Punto)
cercanos ps qs = (p,q)
  where (d,p,q) = minimum [(distancia p q, p, q) | p <- ps, q <-qs]
          distancia (x,y) (u,v) = sqrt ((x-u)^2+(y-v)^2)

```

```

-----
-- Ejercicio 5. Las expresiones aritméticas pueden representarse usando
-- el siguiente tipo de datos
--   data Expr = N Int | V Char | S Expr Expr | P Expr Expr
--             deriving Show
-- Por ejemplo, la expresión 2*(a+5) se representa por
--   P (N 2) (S (V 'a') (N 5))
-- Definir la función
--   valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- decir, el valor de la expresión donde las variables de x se sustituyen
-- por los valores según se indican en el entorno e). Por ejemplo,
--   ghci> valor (P (N 2) (S (V 'a') (V 'b')))) [('a',2),('b',5)]
--   14
-----

```

```

data Expr = N Int | V Char | S Expr Expr | P Expr Expr
deriving Show

```

```

valor :: Expr -> [(Char,Int)] -> Int

```

```

valor (N x) e = x
valor (V x) e = head [y | (z,y) <- e, z == x]
valor (S x y) e = (valor x e) + (valor y e)
valor (P x y) e = (valor x e) * (valor y e)

-----
-- Ejercicio 6. El enunciado del problema 652 de "Números y algo más" es
-- el siguiente:
-- Si factorizamos los factoriales de un número en función de sus
-- divisores primos y sus potencias, ¿cuál es el menor número N tal
-- que entre los factores primos y los exponentes de la factorización
-- de N! están todos los dígitos del cero al nueve?
-- Por ejemplo
--  $6! = 2^4 * 3^2 * 5^1$ , le faltan los dígitos 0,6,7,8 y 9
--  $12! = 2^{10} * 3^5 * 5^2 * 7^1 * 11^1$ , le faltan los dígitos 4,6,8 y 9
-- Definir la función
-- digitosDeFactorizacion :: Integer -> [Integer]
-- tal que (digitosDeFactorizacion n) es el conjunto de los dígitos que
-- aparecen en la factorización de n. Por ejemplo,
-- digitosDeFactorizacion (factorial 6) == [1,2,3,4,5]
-- digitosDeFactorizacion (factorial 12) == [0,1,2,3,5,7]
-- Usando la función anterior, calcular la solución del problema.
-----

digitosDeFactorizacion :: Integer -> [Integer]
digitosDeFactorizacion n =
  sort (nub (concat [digitos x | x <- numerosDeFactorizacion n]))

-- (digitos n) es la lista de los dígitos del número n. Por ejemplo,
-- digitos 320274 == [3,2,0,2,7,4]
digitos :: Integer -> [Integer]
digitos n = [read [x] | x <- show n]

-- (numerosDeFactorizacion n) es el conjunto de los números en la
-- factorización de n. Por ejemplo,
-- numerosDeFactorizacion 60 == [1,2,3,5]
numerosDeFactorizacion :: Integer -> [Integer]
numerosDeFactorizacion n =
  sort (nub (aux (factorizacion n)))
  where aux [] = []

```

```

    aux ((x,y):zs) = x : y : aux zs

-- (factorización n) es la factorización de n. Por ejemplo,
--   factorizacion 300 == [(2,2),(3,1),(5,2)]
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n =
    [(head xs, fromIntegral (length xs)) | xs <- group (factorizacion' n)]

-- (factorizacion' n) es la lista de todos los factores primos de n; es
-- decir, es una lista de números primos cuyo producto es n. Por ejemplo,
--   factorizacion 300 == [2,2,3,5,5]
factorizacion' :: Integer -> [Integer]
factorizacion' n | n == 1    = []
                  | otherwise = x : factorizacion' (div n x)
                  where x = menorFactor n

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

-- (factorial n) es el factorial de n. Por ejemplo,
--   factorial 5 == 120
factorial :: Integer -> Integer
factorial n = product [1..n]

-- Para calcular la solución, se define la constante
solucion =
    head [n | n <- [1..], digitosDeFactorizacion (factorial n) == [0..9]]

-- El cálculo de la solución es
--   ghci> solucion
--   49

```



# Relación 29

## Implementación del TAD de los grafos mediante listas

```
-----  
-- Introducción  
-----  
  
-- El objetivo de esta relación es implementar el TAD de los grafos  
-- mediante listas, de manera análoga a las implementaciones estudiadas  
-- en el tema 22 que se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-22.pdf  
-- y usando la mismas signatura.  
  
-----  
-- Signatura  
-----  
  
module Rel_29_sol  
  (Orientacion (...),  
   Grafo,  
   creaGrafo, -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->  
               -- Grafo v p  
   dirigido, -- (Ix v,Num p) => (Grafo v p) -> Bool  
   adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]  
   nodos, -- (Ix v,Num p) => (Grafo v p) -> [v]  
   aristas, -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]  
   aristaEn, -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool  
   peso -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
```

```

) where

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

import Data.Array
import Data.List

-----
-- Representación de los grafos mediante listas           --
-----

-- Orientación es D (dirigida) ó ND (no dirigida).
data Orientacion = D | ND
    deriving (Eq, Show)

-- (Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p.
data Grafo v p = G Orientacion ([v],[((v,v),p)])
    deriving (Eq, Show)

-----
-- Ejercicios                                             --
-----

-----
-- Ejercicio 1. Definir la función
--   creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
-- tal que (creaGrafo d cs as) es un grafo (dirigido o no, según el
-- valor de o), con el par de cotas cs y listas de aristas as (cada
-- arista es un tríó formado por los dos vértices y su peso). Por
-- ejemplo,
--   ghci> creaGrafo ND (1,3) [(1,2,12),(1,3,34)]
--   G ND ([1,2,3],[((1,2),12),((1,3),34),((2,1),12),((3,1),34)])
--   ghci> creaGrafo D (1,3) [(1,2,12),(1,3,34)]
--   G D ([1,2,3],[((1,2),12),((1,3),34)])
--   ghci> creaGrafo D (1,4) [(1,2,12),(1,3,34)]
--   G D ([1,2,3,4],[((1,2),12),((1,3),34)])
-----

```



```

creaGrafo :: (Ix v, Num p) =>
    Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs as =
    G o (range cs, [(x1,x2),w] | (x1,x2,w) <- as] ++
        if o == D then []
        else [(x2,x1),w] | (x1,x2,w) <- as, x1 /= x2])

```

```

-----
-- Ejercicio 2. Definir, con creaGrafo, la constante
-- ejGrafoND :: Grafo Int Int
-- para representar el siguiente grafo no dirigido

```

```

--
--      12
--      1 ----- 2
--      | \78    /|
--      |  \ 32/  |
--      |   \ /   |
--      34|    5   |55
--      |   /  \  |
--      |  /44  \ |
--      | /    93\|
--      3 ----- 4
--
--      61
-- ghci> ejGrafoND
-- G ND ([1,2,3,4,5],
--      [(1,2),12],[(1,3),34],[(1,5),78],[(2,4),55],[(2,5),32],
--      [(3,4),61],[(3,5),44],[(4,5),93],[(2,1),12],[(3,1),34],
--      [(5,1),78],[(4,2),55],[(5,2),32],[(4,3),61],[(5,3),44],
--      [(5,4),93]])

```

```

-----
ejGrafoND :: Grafo Int Int
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
    (2,4,55),(2,5,32),
    (3,4,61),(3,5,44),
    (4,5,93)]

```

```

-----
-- Ejercicio 3. Definir, con creaGrafo, la constante
-- ejGrafoD :: Grafo Int Int
-- para representar el grafo anterior donde se considera que las aristas

```

```
-- son los pares (x,y) con x < y. Por ejemplo,
-- ghci> ejGrafoD
-- G D ([1,2,3,4,5],
--      [((1,2),12),((1,3),34),((1,5),78),((2,4),55),((2,5),32),
--      ((3,4),61),((3,5),44),((4,5),93))])
-----
```

```
ejGrafoD :: Grafo Int Int
```

```
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                             (2,4,55),(2,5,32),
                             (3,4,61),(3,5,44),
                             (4,5,93)]
```

```
-- Ejercicio 4. Definir la función
-- dirigido :: (Ix v, Num p) => (Grafo v p) -> Bool
-- tal que (dirigido g) se verifica si g es dirigido. Por ejemplo,
-- dirigido ejGrafoD == True
-- dirigido ejGrafoND == False
-----
```

```
dirigido :: (Ix v, Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

```
-- Ejercicio 5. Definir la función
-- nodos :: (Ix v, Num p) => (Grafo v p) -> [v]
-- tal que (nodos g) es la lista de todos los nodos del grafo g. Por
-- ejemplo,
-- nodos ejGrafoND == [1,2,3,4,5]
-- nodos ejGrafoD == [1,2,3,4,5]
-----
```

```
nodos :: (Ix v, Num p) => (Grafo v p) -> [v]
nodos (G _ (ns, _)) = ns
```

```
-- Ejercicio 6. Definir la función
-- adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
-- tal que (adyacentes g v) es la lista de los vértices adyacentes al
```

```
-- nodo v en el grafo g. Por ejemplo,
--   adjacentes ejGrafoND 4 == [5,2,3]
--   adjacentes ejGrafoD  4 == [5]
```

```
-----
adjacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adjacentes (G _ (_,e)) v = nub [u | ((w,u),_) <- e, w == v]
```

```
-----
-- Ejercicio 7. Definir la función
--   aristaEn :: (Ix v, Num p) => Grafo v p -> (v,v) -> Bool
-- (aristaEn g a) se verifica si a es una arista del grafo g. Por
-- ejemplo,
--   aristaEn ejGrafoND (5,1) == True
--   aristaEn ejGrafoND (4,1) == False
--   aristaEn ejGrafoD  (5,1) == False
--   aristaEn ejGrafoD  (1,5) == True
```

```
-----
aristaEn :: (Ix v, Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adjacentes g x
```

```
-----
-- Ejercicio 8. Definir la función
--   peso :: (Ix v, Num p) => v -> v -> Grafo v p -> p
-- tal que (peso v1 v2 g) es el peso de la arista que une los vértices
-- v1 y v2 en el grafo g. Por ejemplo,
--   peso 1 5 ejGrafoND == 78
--   peso 1 5 ejGrafoD  == 78
```

```
-----
peso :: (Ix v, Num p) => v -> v -> Grafo v p -> p
peso x y (G _ (_,gs)) = head [c | ((x',y'),c) <- gs, x==x', y==y']
```

```
-----
-- Ejercicio 9. Definir la función
--   aristas :: (Ix v, Num p) => Grafo v p -> [(v,v,p)]
-- (aristasD g) es la lista de las aristas del grafo g. Por ejemplo,
--   ghci> aristas ejGrafoD
--   [(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
```

```
-- (3,5,44), (4,5,93)]
-- ghci> aristas ejGrafoND
-- [(1,2,12), (1,3,34), (1,5,78), (2,1,12), (2,4,55), (2,5,32),
-- (3,1,34), (3,4,61), (3,5,44), (4,2,55), (4,3,61), (4,5,93),
-- (5,1,78), (5,2,32), (5,3,44), (5,4,93)]
```

```
-----

aristas :: (Ix v, Num p) => Grafo v p -> [(v,v,p)]
aristas (G _ (_,g)) = [(v1,v2,p) | ((v1,v2),p) <- g]
```

# Relación 30

## Problemas básicos con el TAD de los grafos

```
-- -----  
-- Introducción --  
-- -----  
  
-- El objetivo de esta relación de ejercicios es definir funciones sobre  
-- el TAD de los grafos, utilizando las implementaciones estudiadas  
-- en el tema 22 que se pueden descargar desde  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/codigos.zip  
--  
-- Las transparencias del tema 22 se encuentran en  
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-22.pdf  
  
-- -----  
-- Importación de librerías --  
-- -----  
  
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}  
  
import Data.Array  
import Data.List (nub)  
import Test.QuickCheck  
  
-- Hay que seleccionar una implementación del TAD de los grafos  
import GrafoConVectorDeAdyacencia  
-- import GrafoConMatrizDeAdyacencia
```

```

-- import Rel_29_sol

-- -----
-- Ejemplos
-- -----

-- Para los ejemplos se usarán los siguientes grafos.
g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11 :: Grafo Int Int
g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                        (2,4,55),(2,5,32),
                        (3,4,61),(3,5,44),
                        (4,5,93)]
g2 = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                       (2,4,55),(2,5,32),
                       (4,3,61),(4,5,93)]
g3 = creaGrafo D (1,3) [(1,2,0),(2,2,0),(3,1,0),(3,2,0)]
g4 = creaGrafo D (1,4) [(1,2,3),(2,1,5)]
g5 = creaGrafo D (1,1) [(1,1,0)]
g6 = creaGrafo D (1,4) [(1,3,0),(3,1,0),(3,3,0),(4,2,0)]
g7 = creaGrafo ND (1,4) [(1,3,0)]
g8 = creaGrafo D (1,5) [(1,1,0),(1,2,0),(1,3,0),(2,4,0),(3,1,0),
                       (4,1,0),(4,2,0),(4,4,0),(4,5,0)]
g9 = creaGrafo D (1,5) [(4,1,1),(4,3,2),(5,1,0)]
g10 = creaGrafo ND (1,3) [(1,2,1),(1,3,1),(2,3,1),(3,3,1)]
g11 = creaGrafo D (1,3) [(1,2,1),(1,3,1),(2,3,1),(3,3,1)]

-- -----
-- Ejercicio 1. El grafo completo de orden  $n$ ,  $K(n)$ , es un grafo no
-- dirigido cuyos conjunto de vértices es  $\{1,..n\}$  y tiene una arista
-- entre par de vértices distintos. Definir la función,
--   completo :: Int -> Grafo Int Int
-- tal que (completo  $n$ ) es el grafo completo de orden  $n$ . Por ejemplo,
--   ghci> completo 4
--   G ND (array (1,4) [(1,[(2,0),(3,0),(4,0)]),
--                    (2,[(1,0),(3,0),(4,0)]),
--                    (3,[(1,0),(2,0),(4,0)]),
--                    (4,[(1,0),(2,0),(3,0)])])
-- -----

completo :: Int -> Grafo Int Int

```

```
completo n = creaGrafo ND (1,n) xs
  where xs = [(x,y,0) | x <- [1..n], y <- [1..n], x < y]
```

```
completo' :: Int -> Grafo Int Int
```

```
completo' n = creaGrafo ND (1,n) [(a,b,0) | a<-[1..n],b<-[1..a-1]]
```

```
-----
-- Ejercicio 2. El ciclo de orden n, C(n), es un grafo no dirigido
-- cuyo conjunto de vértices es {1,...,n} y las aristas son
-- (1,2), (2,3), ..., (n-1,n), (n,1)
-- Definir la función
-- grafoCiclo :: Int -> Grafo Int Int
-- tal que (grafoCiclo n) es el grafo ciclo de orden n. Por ejemplo,
-- ghci> grafoCiclo 3
-- G ND (array (1,3) [(1,[(3,0),(2,0)]),(2,[(1,0),(3,0)]),(3,[(2,0),(1,0)])])
-----
```

```
grafoCiclo :: Int -> Grafo Int Int
```

```
grafoCiclo n = creaGrafo ND (1,n) xs
  where xs = [(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)]
```

```
-----
-- Ejercicio 3. Definir la función
-- nVertices :: (Ix v, Num p) => Grafo v p -> Int
-- tal que (nVertices g) es el número de vértices del grafo g. Por
-- ejemplo,
-- nVertices (completo 4) == 4
-- nVertices (completo 5) == 5
-----
```

```
nVertices :: (Ix v, Num p) => Grafo v p -> Int
```

```
nVertices = length . nodos
```

```
-----
-- Ejercicio 4. Definir la función
-- noDirigido :: (Ix v, Num p) => Grafo v p -> Bool
-- tal que (noDirigido g) se verifica si el grafo g es no dirigido. Por
-- ejemplo,
-- noDirigido g1 == True
-- noDirigido g2 == False
```

```

--      noDirigido (completo 4) == True
-----

noDirigido :: (Ix v, Num p) => Grafo v p -> Bool
noDirigido = not . dirigido

-----

-- Ejercicio 5. En un un grafo g, los incidentes de un vértice v es el
-- conjuntos de vértices x de g para los que hay un arco (o una arista)
-- de x a v; es decir, que v es adyacente a x. Definir la función
-- incidentes :: (Ix v, Num p) => (Grafo v p) -> v -> [v]
-- tal que (incidentes g v) es la lista de los vértices incidentes en el
-- vértice v. Por ejemplo,
-- incidentes g2 5 == [1,2,4]
-- adyacentes g2 5 == []
-- incidentes g1 5 == [1,2,3,4]
-- adyacentes g1 5 == [1,2,3,4]
-----

incidentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

-----

-- Ejercicio 6. En un un grafo g, los contiguos de un vértice v es el
-- conjuntos de vértices x de g tales que x es adyacente o incidente con
-- v. Definir la función
-- contiguos :: (Ix v, Num p) => Grafo v p -> v -> [v]
-- tal que (contiguos g v) es el conjunto de los vértices de g contiguos
-- con el vértice v. Por ejemplo,
-- contiguos g2 5 == [1,2,4]
-- contiguos g1 5 == [1,2,3,4]
-----

contiguos :: (Ix v, Num p) => Grafo v p -> v -> [v]
contiguos g v = nub (adyacentes g v ++ incidentes g v)

-----

-- Ejercicio 7. Definir la función
-- lazos :: (Ix v, Num p) => Grafo v p -> [(v,v)]
-- tal que (lazos g) es el conjunto de los lazos (es decir, aristas

```



```
-- cuyos extremos son iguales) del grafo g. Por ejemplo,
-- ghci> lazos g3
-- [(2,2)]
-- ghci> lazos g2
-- []
```

```
lazos :: (Ix v, Num p) => Grafo v p -> [(v,v)]
lazos g = [(x,x) | x <- nodos g, aristaEn g (x,x)]
```

```
-- -----
-- Ejercicio 8. Definir la función
-- nLazos :: (Ix v, Num p) => Grafo v p -> Int
-- tal que (nLazos g) es el número de lazos del grafo g. Por
-- ejemplo,
-- nLazos g3 == 1
-- nLazos g2 == 0
```

```
nLazos :: (Ix v, Num p) => Grafo v p -> Int
nLazos = length . lazos
```

```
-- -----
-- Ejercicio 9. Definir la función
-- nAristas :: (Ix v, Num p) => Grafo v p -> Int
-- tal que (nAristas g) es el número de aristas del grafo g. Si g es no
-- dirigido, las aristas de v1 a v2 y de v2 a v1 sólo se cuentan una
-- vez y los lazos se cuentan dos veces. Por ejemplo,
-- nAristas g1 == 8
-- nAristas g2 == 7
-- nAristas g10 == 4
-- nAristas (completo 4) == 6
-- nAristas (completo 5) == 10
```

```
nAristas :: (Ix v, Num p) => Grafo v p -> Int
nAristas g
  | dirigido g = length (aristas g)
  | otherwise = (length (aristas g) `div` 2) + nLazos g
```

```

-----
-- Ejercicio 10. Definir la función
--   prop_nAristasCompleto :: Int -> Bool
-- tal que (prop_nAristasCompleto n) se verifica si el número de aristas
-- del grafo completo de orden n es  $n*(n-1)/2$  y, usando la función,
-- comprobar que la propiedad se cumple para n de 1 a 20.
-----

```

```

prop_nAristasCompleto :: Int -> Bool
prop_nAristasCompleto n =
  nAristas (completo n) == n*(n-1) `div` 2

```

```

-- La comprobación es
--   ghci> and [prop_nAristasCompleto n | n <- [1..20]]
--   True

```

```

-----
-- Ejercicio 11. El grado positivo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g adyacentes con v. Definir la función
--   gradoPos :: (Ix v, Num p) => Grafo v p -> v -> Int
-- tal que (gradoPos g v) es el grado positivo del vértice v en el grafo
-- g. Por ejemplo,
--   gradoPos g1 5 == 4
--   gradoPos g2 5 == 0
--   gradoPos g2 1 == 3
-----

```

```

gradoPos :: (Ix v, Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)

```

```

-----
-- Ejercicio 12. El grado negativo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g incidentes con v. Definir la función
--   gradoNeg :: (Ix v, Num p) => Grafo v p -> v -> Int
-- tal que (gradoNeg g v) es el grado negativo del vértice v en el grafo
-- g. Por ejemplo,
--   gradoNeg g1 5 == 4
--   gradoNeg g2 5 == 3
--   gradoNeg g2 1 == 0
-----

```

```
gradoNeg :: (Ix v, Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)
```

```
-----
-- Ejercicio 13. El grado de un vértice v de un grafo dirigido g, es el
-- número de aristas de g que contiene a v. Si g es no dirigido, el
-- grado de un vértice v es el número de aristas incidentes en v, teniendo
-- en cuenta que los lazos se cuentan dos veces. Definir la función
-- grado :: (Ix v, Num p) => Grafo v p -> v -> Int
-- tal que (grado g v) es el grado del vértice v en el grafo g. Por
-- ejemplo,
-- grado g1 5 == 4
-- grado g2 5 == 3
-- grado g2 1 == 3
-- grado g3 2 == 4
-- grado g3 1 == 2
-- grado g3 3 == 2
-- grado g5 1 == 3
-- grado g10 3 == 4
-- grado g11 3 == 4
-----
```

```
grado :: (Ix v, Num p) => Grafo v p -> v -> Int
grado g v | dirigido g           = gradoNeg g v + gradoPos g v
          | (v,v) 'elem' lazos g = length (incidentes g v) + 1
          | otherwise            = length (incidentes g v)
```

```
-----
-- Ejercicio 14. Comprobar con QuickCheck que para cualquier grafo g, la
-- suma de los grados positivos de los vértices de g es igual que la
-- suma de los grados negativos de los vértices de g.
-----
```

```
-- La propiedad es
```

```
prop_sumaGrados :: Grafo Int Int -> Bool
prop_sumaGrados g =
  sum [gradoPos g v | v <- vs] == sum [gradoNeg g v | v <- vs]
  where vs = nodos g
```

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

```
-----
-- Ejercicio 15. En la teoría de grafos, se conoce como "Lema del
-- apretón de manos" la siguiente propiedad: la suma de los grados de
-- los vértices de  $g$  es el doble del número de aristas de  $g$ .
-- Comprobar con QuickCheck que para cualquier grafo  $g$ , se verifica
-- dicha propiedad.
-----
```

```
prop_apretonManos :: Grafo Int Int -> Bool
prop_apretonManos g =
  sum [grado g v | v <- nodos g] == 2 * nAristas g
```

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

```
-----
-- Ejercicio 16. Comprobar con QuickCheck que en todo grafo, el número
-- de nodos de grado impar es par.
-----
```

```
prop_numNodosGradoImpar :: Grafo Int Int -> Bool
prop_numNodosGradoImpar g = even m
  where vs = nodos g
        m = length [v | v <- vs, odd(grado g v)]
```

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

```
-----
-- Ejercicio 17. Definir la propiedad
-- prop_GradoCompleto :: Int -> Bool
-- tal que (prop_GradoCompleto  $n$ ) se verifica si todos los vértices del
-- grafo completo  $K(n)$  tienen grado  $n-1$ . Usarla para comprobar que dicha
-- propiedad se verifica para los grafos completos de grados 1 hasta 30.
-----
```

```

-----
prop_GradoCompleto :: Int -> Bool
prop_GradoCompleto n =
    and [grado g v == (n-1) | v <- nodos g]
        where g = completo n

-- La comprobación es
-- ghci> and [prop_GradoCompleto n | n <- [1..30]]
-- True

```

```

-----
-- Ejercicio 18. Un grafo es regular si todos sus vértices tienen el
-- mismo grado. Definir la función
-- regular :: (Ix v, Num p) => Grafo v p -> Bool
-- tal que (regular g) se verifica si todos los nodos de g tienen el
-- mismo grado.
-- regular g1 == False
-- regular g2 == False
-- regular (completo 4) == True

```

```

-----
regular :: (Ix v, Num p) => Grafo v p -> Bool
regular g = and [grado g v == k | v <- vs]
    where vs = nodos g
          k = grado g (head vs)

```

```

-----
-- Ejercicio 19. Definir la propiedad
-- prop_CompletoRegular :: Int -> Int -> Bool
-- tal que (prop_CompletoRegular m n) se verifica si todos los grafos
-- completos desde el de orden m hasta el de orden n son regulares y
-- usarla para comprobar que todos los grafos completo desde el de orden
-- 1 hasta el de orden 30 son regulares.

```

```

prop_CompletoRegular :: Int -> Int -> Bool
prop_CompletoRegular m n = and [regular (completo x) | x <- [m..n]]

```

```

-- La comprobación es

```

```
-- ghci> prop_CompletoRegular 1 30
-- True
```

```
-----
-- Ejercicio 20. Un grafo es k-regular si todos sus vértices son de
-- grado k. Definir la función
```

```
regularidad :: (Ix v, Num p) => Grafo v p -> Maybe Int
-- tal que (regularidad g) es la regularidad de g. Por ejemplo,
regularidad g1 == Nothing
regularidad (completo 4) == Just 3
regularidad (completo 5) == Just 4
regularidad (grafoCiclo 4) == Just 2
regularidad (grafoCiclo 5) == Just 2
-----
```

```
regularidad :: (Ix v, Num p) => Grafo v p -> Maybe Int
regularidad g | regular g = Just (grado g (head (nodos g)))
              | otherwise = Nothing
```

```
-----
-- Ejercicio 21. Definir la propiedad
```

```
prop_completoRegular :: Int -> Bool
-- tal que (prop_completoRegular n) se verifica si el grafo completo de
-- orden n es (n-1)-regular. Por ejemplo,
prop_completoRegular 5 == True
-- y usarla para comprobar que la cumplen todos los grafos completos
-- desde orden 1 hasta 20.
```

```
prop_completoRegular :: Int -> Bool
prop_completoRegular n =
  regularidad (completo n) == Just (n-1)
```

```
-- La comprobación es
```

```
ghci> and [prop_completoRegular n | n <- [1..20]]
True
```

```
-----
-- Ejercicio 22. Definir la propiedad
```

```
prop_cicloRegular :: Int -> Bool
```

```

-- tal que (prop_cicloRegular n) se verifica si el grafo ciclo de orden
-- n es 2-regular. Por ejemplo,
--   prop_cicloRegular 2 == True
-- y usarla para comprobar que la cumplen todos los grafos ciclos
-- desde orden 3 hasta 20.
-----

prop_cicloRegular :: Int -> Bool
prop_cicloRegular n =
  regularidad (grafoCiclo n) == Just 2

-- La comprobación es
--   ghci> and [prop_cicloRegular n | n <- [3..20]]
--   True
-----

-- § Generador de grafos
-----

-- (generaGND n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--   ghci> generaGND 3 [4,2,5]
--   (ND,array (1,3) [(1,[(2,4),(3,2)]),
--                    (2,[(1,4),(3,5)]),
--                    3,[(1,2),(2,5)]])
--   ghci> generaGND 3 [4,-2,5]
--   (ND,array (1,3) [(1,[(2,4)]), (2,[(1,4),(3,5)]), (3,[(2,5)])])
generaGND :: Int -> [Int] -> Grafo Int Int
generaGND n ps = creaGrafo ND (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n], x < y]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- (generaGD n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--   ghci> generaGD 3 [4,2,5]
--   (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                  (2,[]),
--                  (3,[])])
--   ghci> generaGD 3 [4,2,5,3,7,9,8,6]

```

```

--      (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                      (2,[(1,3),(2,7),(3,9)]),
--                      (3,[(1,8),(2,6])])])
generaGD :: Int -> [Int] -> Grafo Int Int
generaGD n ps = creaGrafo D (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n]]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- genGD es un generador de grafos dirigidos. Por ejemplo,
-- ghci> sample genGD
--      (D,array (1,4) [(1,[(1,1)]), (2,[(3,1)]), (3,[(2,1),(4,1)]), (4,[(4,1)])])
--      (D,array (1,2) [(1,[(1,6)]), (2,[])])
--      ...
genGD :: Gen (Grafo Int Int)
genGD = do n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          return (generaGD n xs)

-- genGND es un generador de grafos dirigidos. Por ejemplo,
-- ghci> sample genGND
--      (ND,array (1,1) [(1,[])])
--      (ND,array (1,3) [(1,[(2,3),(3,13)]), (2,[(1,3)]), (3,[(1,13)])])
--      ...
genGND :: Gen (Grafo Int Int)
genGND = do n <- choose (1,10)
           xs <- vectorOf (n*n) arbitrary
           return (generaGND n xs)

-- genG es un generador de grafos. Por ejemplo,
-- ghci> sample genG
--      (D,array (1,3) [(1,[(2,1)]), (2,[(1,1),(2,1)]), (3,[(3,1)])])
--      (ND,array (1,3) [(1,[(2,2)]), (2,[(1,2)]), (3,[])])
--      ...
genG :: Gen (Grafo Int Int)
genG = do d <- choose (True,False)
          n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          if d then return (generaGD n xs)
             else return (generaGND n xs)

```



---

```
-- Los grafos está contenido en la clase de los objetos generables
-- aleatoriamente.
instance Arbitrary (Grafo Int Int) where
    arbitrary = genG
```



# Relación 31

## Demostración de propiedades por inducción

```
-- -----  
-- Introducción --  
-- -----  
  
-- En esta relación se plantean ejercicios de demostración por inducción  
-- de propiedades de programas. En concreto,  
-- * la suma de los n primeros impares es  $n^2$ ,  
-- *  $1 + 2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{(n+1)}$ ,  
-- * todos los elementos de (copia n x) son iguales a x,  
-- * la equivalencia de las definiciones de factorial con y sin  
-- acumulador,  
-- * amplia xs y = xs ++ [y].  
-- * numeroDeListasConSuma n =  $2^{(n-1)}$ ,  
-- * fibItAux n (fib k) (fib (k+1)) = fib (k+n),  
-- * potencia x n ==  $x^n$ ,  
-- * reverse (xs ++ ys) == reverse ys ++ reverse xs,  
-- * reverse (reverse xs) = xs,  
-- y por inducción sobre árboles binarios  
-- * espejo (espejo x) = x,  
-- * postorden (espejo x) = reverse (preorden x),  
-- * reverse (preorden (espejo x)) = postorden x,  
-- * nNodos (espejo x) == nNodos x,  
-- * length (preorden x) == nNodos x,  
-- * nNodos x <=  $2^{(profundidad x) - 1}$ ,  
-- * nHojas x = nNodos x + 1,
```

```

-- * preordenItAux x ys = preorden x ++ ys
--
-- Estos ejercicios corresponden al tema 8 cuyas transparencias se
-- encuentran en
-- http://www.cs.us.es/~jalonso/cursos/ilm-12/temas/tema-8.pdf
--
-- -----
-- Importación de librerías
-- -----

import Test.QuickCheck
import Control.Monad

-- -----
-- Ejercicio 1.1. Definir por recursión la función
-- sumaImpares :: Int -> Int
-- tal que (sumaImpares n) es la suma de los n primeros números
-- impares. Por ejemplo,
-- sumaImpares 5 == 25
-- -----

sumaImpares :: Int -> Int
sumaImpares 0 = 0
sumaImpares n = sumaImpares (n-1) + (2*n-1)

sumaImpares2 :: Int -> Int
sumaImpares2 0 = 0
sumaImpares2 n = 2*n+1 + sumaImpares2 (n-1)

-- -----
-- Ejercicio 1.2. Definir, sin usar recursión, la función
-- sumaImpares' :: Int -> Int
-- tal que (sumaImpares' n) es la suma de los n primeros números
-- impares. Por ejemplo,
-- ghci> sumaImpares' 5 == 25
-- -----

sumaImpares' :: Int -> Int
sumaImpares' n = sum [1,3..(2*n-1)]

```

```
-----  
-- Ejercicio 1.3. Definir la función  
-- sumaImparesIguales :: Int -> Int -> Bool  
-- tal que (sumaImparesIguales m n) se verifica si para todo x entre m y  
-- n se tiene que (sumaImpares x) y (sumaImpares' x) son iguales.  
--  
-- Comprobar que (sumaImpares x) y (sumaImpares' x) son iguales para  
-- todos los números x entre 1 y 100.  
-----  
  
-- La definición es  
sumaImparesIguales :: Int -> Int -> Bool  
sumaImparesIguales m n =  
    and [sumaImpares x == sumaImpares' x | x <- [m..n]]  
  
-- La comprobación es  
-- ghci> sumaImparesIguales 1 100  
-- True  
  
-----  
-- Ejercicio 1.4. Definir la función  
-- grafoSumaImpares :: Int -> Int -> [(Int,Int)]  
-- tal que (grafoSumaImpares m n) es la lista formada por los números x  
-- entre m y n y los valores de (sumaImpares x).  
--  
-- Calcular (grafoSumaImpares 1 9).  
-----  
  
-- La definición es  
grafoSumaImpares :: Int -> Int -> [(Int,Int)]  
grafoSumaImpares m n =  
    [(x,sumaImpares x) | x <- [m..n]]  
  
-- El cálculo es  
-- ghci> grafoSumaImpares 1 9  
-- [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64),(9,81)]  
  
-----  
-- Ejercicio 1.5. Demostrar por inducción que para todo n,
```

```
-- (sumaImpares n) es igual a n^2.
```

```
-----
```

```
{-
```

```
  Caso base: Hay que demostrar que
```

```
    sumaImpares 0 = 0^2
```

```
  En efecto,
```

```
    sumaImpares 0 [por hipótesis]
```

```
  = 0 [por sumaImpares.1]
```

```
  = 0^2 [por aritmética]
```

```
  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
```

```
    sumaImpares n = n^2
```

```
  Hay que demostrar que
```

```
    sumaImpares (n+1) = (n+1)^2
```

```
  En efecto,
```

```
    sumaImpares (n+1) =
```

```
  = (sumaImpares n) + (2*n+1) [por sumaImpares.2]
```

```
  = n^2 + (2*n+1) [por H.I.]
```

```
  = (n+1)^2 [por álgebra]
```

```
-}
```

```
-----
```

```
-- Ejercicio 2.1. Definir por recursión la función
```

```
-- sumaPotenciasDeDosMasUno :: Int -> Int
```

```
-- tal que
```

```
-- sumaPotenciasDeDosMasUno n = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
```

```
-- Por ejemplo,
```

```
-- sumaPotenciasDeDosMasUno 3 == 16
```

```
-----
```

```
sumaPotenciasDeDosMasUno :: Int -> Int
```

```
sumaPotenciasDeDosMasUno 0 = 2
```

```
sumaPotenciasDeDosMasUno n = sumaPotenciasDeDosMasUno (n-1) + 2^n
```

```
-----
```

```
-- Ejercicio 2.2. Definir por comprensión la función
```

```
-- sumaPotenciasDeDosMasUno' :: Int -> Int
```

```
-- tal que
```

```
-- (sumaPotenciasDeDosMasUno' n) = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
```

```
-- Por ejemplo,
-- sumaPotenciasDeDosMasUno' 3 == 16
```

```
-----
sumaPotenciasDeDosMasUno' :: Int -> Int
sumaPotenciasDeDosMasUno' n = 1 + sum [2^x | x <- [0..n]]
```

```
-----
-- Ejercicio 2.3. Demostrar por inducción que
-- sumaPotenciasDeDosMasUno n = 2^(n+1)
-----
```

```
{-
  Caso base: Hay que demostrar que
    sumaPotenciasDeDosMasUno 0 = 2^(0+1)
  En efecto,
    sumaPotenciasDeDosMasUno 0
  = 2                                [por sumaPotenciasDeDosMasUno.1]
  = 2^(0+1)                          [por aritmética]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
    sumaPotenciasDeDosMasUno n = 2^(n+1)
  Hay que demostrar que
    sumaPotenciasDeDosMasUno (n+1) = 2^((n+1)+1)
  En efecto,
    sumaPotenciasDeDosMasUno (n+1)
  = (sumaPotenciasDeDosMasUno n) + 2^(n+1) [por sumaPotenciasDeDosMasUno.2]
  = 2^(n+1) + 2^(n+1)                       [por H.I.]
  = 2^((n+1)+1)                             [por aritmética]
-}
```

```
-----
-- Ejercicio 3.1. Definir por recursión la función
-- copia :: Int -> a -> [a]
-- tal que (copia n x) es la lista formado por n copias del elemento
-- x. Por ejemplo,
-- copia 3 2 == [2,2,2]
-----
```

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

```

copia 0 _ = []           -- copia.1
copia n x = x : copia (n-1) x  -- copia.2

-----

-- Ejercicio 3.2. Definir por recursión la función
--   todos :: (a -> Bool) -> [a] -> Bool
-- tal que (todos p xs) se verifica si todos los elementos de xs cumplen
-- la propiedad p. Por ejemplo,
--   todos even [2,6,4] == True
--   todos even [2,5,4] == False
-----

todos :: (a -> Bool) -> [a] -> Bool
todos p []      = True           -- todos.1
todos p (x : xs) = p x && todos p xs  -- todos.2

-----

-- Ejercicio 3.3. Comprobar con QuickCheck que todos los elementos de
-- (copia n x) son iguales a x.
-----

-- La propiedad es
prop_copia :: Eq a => Int -> a -> Bool
prop_copia n x =
  todos (==x) (copia n' x)
  where n' = abs n

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

-----

-- Ejercicio 3.4. Demostrar, por inducción en n, que todos los elementos
-- de (copia n x) son iguales a x.
-----

{-
  Hay que demostrar que para todo n y todo x,
  todos (==x) (copia n x)

```



*Caso base: Hay que demostrar que*  
*todos (==x) (copia 0 x) = True*

*En efecto,*

```

    todos (== x) (copia 0 x)
  = todos (== x) []           [por copia.1]
  = True                      [por todos.1]

```

*Caso inductivo: Se supone la hipótesis de inducción (H.I.)*

*todos (==x) (copia n x) = True*

*Hay que demostrar que*

*todos (==x) (copia (n+1) x) = True*

*En efecto,*

```

    todos (==x) (copia (n+1) x)
  = todos (==x) (x : copia n x )           [por copia.2]
  = x == x && todos (==x) (copia n x )     [por todos.2]
  = True && todos (==x) (copia n x )      [por def. de ==]
  = todos (==x) (copia n x )             [por def. de &&]
  = True                                   [por H.I.]

```

-}

```

-----
-- Ejercicio 4.1. Definir por recursión la función
-- factR :: Integer -> Integer
-- tal que (factR n) es el factorial de n. Por ejemplo,
-- factR 4 == 24
-----

```

```

factR :: Integer -> Integer
factR 0 = 1
factR n = n * factR (n-1)

```

```

-----
-- Ejercicio 4.2. Definir por comprensión la función
-- factC :: Integer -> Integer
-- tal que (factR n) es el factorial de n. Por ejemplo,
-- factC 4 == 24
-----

```

```

factC :: Integer -> Integer
factC n = product [1..n]

```

```

-----
-- Ejercicio 4.3. Comprobar con QuickCheck que las funciones factR y
-- factC son equivalentes sobre los números naturales.
-----

```

```

-- La propiedad es
prop_factR_factC :: Integer -> Bool
prop_factR_factC n =
  factR n' == factC n'
  where n' = abs n

```

```

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

```

```

-----
-- Ejercicio 4.4. Comprobar con QuickCheck si las funciones factR y
-- factC son equivalentes sobre los números enteros.
-----

```

```

-- La propiedad es
prop_factR_factC_Int :: Integer -> Bool
prop_factR_factC_Int n =
  factR n == factC n

```

```

-- La comprobación es
-- ghci> quickCheck prop_factR_factC_Int
-- *** Exception: Non-exhaustive patterns in function factR

```

```

-- No son iguales ya que factR no está definida para los números
-- negativos y factC de cualquier número negativo es 0.

```

```

-----
-- Ejercicio 4.5. Se considera la siguiente definición iterativa de la
-- función factorial

```

```

-- factI :: Integer -> Integer
-- factI n = factI' n 1
--
-- factI' :: Integer -> Integer -> Integer

```

```

--      factI' 0 x = x                -- factI'.1
--      factI' n x = factI' (n-1) n*x -- factI'.2
-- Comprobar con QuickCheck que factI y factR son equivalentes sobre los
-- números naturales.
-----

factI :: Integer -> Integer
factI n = factI' n 1

factI' :: Integer -> Integer -> Integer
factI' 0 x = x
factI' n x = factI' (n-1) n*x

-- La propiedad es
prop_factI_factR n =
  factI n' == factR n'
  where n' = abs n

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

-- Ejercicio 4.6. Comprobar con QuickCheck que para todo número natural
-- n, (factI' n x) es igual al producto de x y (factR n).
-----

-- La propiedad es
prop_factI' :: Integer -> Integer -> Bool
prop_factI' n x =
  factI' n' x == x * factR n'
  where n' = abs n

-- La comprobación es
-- ghci> quickCheck prop_factI'
-- OK, passed 100 tests.
-----

-- Ejercicio 4.7. Demostrar por inducción que para todo número natural
-- n, (factI' n x) es igual x*n!

```

-----

{-

*Demostración (por inducción en n)*

*Caso base: Hay que demostrar que  $\text{factI}' 0 x = x*0!$*

*En efecto,*

```
factI' 0 x
= x           [por factI'.1]
= x*0!       [por álgebra]
```

*Caso inductivo: Se supone la hipótesis de inducción: para todo x,*

*$\text{factI}' n x = x*n!$*

*hay que demostrar que para todo x*

*$\text{factI}' (n+1) x = x*(n+1)!$*

*En efecto,*

```
factI' (n+1) x
= factI' n (n+1)*x [por factI'.2]
= (n+1)*x*n!      [por hipótesis de inducción]
= x*(n+1)!       [por álgebra]
```

-}

-----

-- *Ejercicio 5.1. Definir, recursivamente y sin usar (++) la función*

-- *`amplia :: [a] -> a -> [a]`*

-- *tal que `(amplia xs y)` es la lista obtenida añadiendo el elemento y al*

-- *final de la lista xs. Por ejemplo,*

-- *`amplia [2,5] 3 == [2,5,3]`*

-----

```
amplia :: [a] -> a -> [a]
```

```
amplia [] y = [y] -- amplia.1
```

```
amplia (x:xs) y = x : amplia xs y -- amplia.2
```

-----

-- *Ejercicio 5.2. Definir, mediante plegado, la función*

-- *`ampliaF :: [a] -> a -> [a]`*

-- *tal que `(ampliaF xs y)` es la lista obtenida añadiendo el elemento y al*

-- *final de la lista xs. Por ejemplo,*

-- *`ampliaF [2,5] 3 == [2,5,3]`*

```
-----  
ampliaF :: [a] -> a -> [a]  
ampliaF xs y = foldr (:) [y] xs
```

```
-----  
-- Ejercicio 5.3. Comprobar con QuickCheck que amplia y ampliaF son  
-- equivalentes.  
-----
```

```
-- La propiedad es  
prop_amplia_ampliaF :: Eq a => [a] -> a -> Bool  
prop_amplia_ampliaF xs y =  
    amplia xs y == ampliaF xs y
```

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

```
-----  
-- Ejercicio 5.4. Comprobar con QuickCheck que  
-- amplia xs y = xs ++ [y]  
-----
```

```
-- La propiedad es  
prop_amplia :: Eq a => [a] -> a -> Bool  
prop_amplia xs y =  
    amplia xs y == xs ++ [y]
```

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

```
-----  
-- Ejercicio 5.5. Demostrar por inducción que  
-- amplia xs y = xs ++ [y]  
-----
```

```
{-  
    Demostración: Por inducción en xs.
```

Caso base: Hay que demostrar que

```
amplia [] y = [] ++ [y]
```

En efecto,

```
amplia [] y
= [y]           [por amplia.1]
= [] ++ [y]    [por (++).1]
```

Caso inductivo: Se supone la hipótesis de inducción

```
amplia xs y = xs ++ [y]
```

Hay que demostrar que

```
amplia (x:xs) y = (x:xs) ++ [y]
```

En efecto,

```
amplia (x:xs) y
= x : amplia xs y    [por amplia.2]
= x : (xs ++ [y])    [por hipótesis de inducción]
= (x:xs) ++ [y]     [por (++).2]
```

```
-}
```

```
-----
-- Ejercicio 6.1. Definir la función
-- listaConSuma :: Int -> [[Int]]
-- que, dado un número natural n, devuelve todas las listas de enteros
-- positivos (esto es, enteros mayores o iguales que 1) cuya suma sea
-- n. Por ejemplo,
-- Main> listaConSuma 4
-- [[1,1,1,1],[1,1,2],[1,2,1],[1,3],[2,1,1],[2,2],[3,1],[4]]
-----
```

```
listaConSuma :: Int -> [[Int]]
listaConSuma 0 = [[]]
listaConSuma n = [x:xs | x <- [1..n], xs <- listaConSuma (n-x)]
```

```
-----
-- Ejercicio 6.2. Definir la función
-- numeroDeListasConSuma :: Int -> Int
-- tal que (numeroDeListasConSuma n) es el número de elementos de
-- (listaConSuma n). Por ejemplo,
-- numeroDeListasConSuma 10 = 512
-----
```

```

numeroDeListasConSuma :: Int -> Int
numeroDeListasConSuma = length . listaConSuma

-----
-- Ejercicio 6.2. Definir la constante
--   numerosDeListasConSuma :: [(Int,Int)]
-- tal que numerosDeListasConSuma es la lista de los pares formado por un
-- número natural n mayor que 0 y el número de elementos de
-- (listaConSuma n).
--
-- Calcular el valor de
--   take 10 numerosDeListasConSuma
-----

-- La constante es
numeroDeListasConSuma :: [(Int,Int)]
numeroDeListasConSuma = [(n,numeroDeListasConSuma n) | n <- [1..]]

-- El cálculo es
--   ghci> take 10 numerosDeListasConSuma
--   [(1,1),(2,2),(3,4),(4,8),(5,16),(6,32),(7,64),(8,128),(9,256),(10,512)]

-----
-- Ejercicio 6.4. A partir del ejercicio anterior, encontrar una fórmula
-- para calcular el valor de (numeroDeListasConSuma n) para los
-- números n mayores que 0.
--
-- Demostrar dicha fórmula por inducción fuerte.
-----

{-
La fórmula es
  numeroDeListasConSuma n = 2^(n-1)
La demostración, por inducción fuerte en n, es la siguiente:

Caso base (n=1):
  numeroDeListasConSuma 1
= length (listaConSuma 1)
  [por numeroDeListasConSuma]

```

```

= length [[x:xs | x <- [1..1], xs <- listaConSuma [[]]]
      [por listaConSuma.2]
= length [[]]
      [por def. de listas de comprensión]
= 1
      [por def. de length]
= 2^(1-1)
      [por aritmética]

```

*Paso de inducción: Se supone que  
para todo  $x$  en  $[1..n-1]$ ,  
 $\text{numeroDeListasConSuma } x = 2^{(x-1)}$*

*Hay que demostrar que  
 $\text{numeroDeListasConSuma } n = 2^{(n-1)}$*

```

En efecto,
numeroDeListasConSuma n
= length (listaConSuma n)
      [por numeroDeListasConSuma]
= length [x:xs | x <- [1..n], xs <- listaConSuma (n-x)]
      [por listaConSuma.2]
= sum [numeroDeListasConSuma (n-x) | x <- [1..n]]
      [por length y listas de comprensión]
= sum [2^(n-x-1) | x <- [1..n-1]] + 1
      [por hip. de inducción y numeroDeListasConSuma]
= 2^(n-2) + 2^(n-3) + ... + 2^1 + 2^0 + 1
= 2^(n-1)
      [por el ejercicio 2c de la relación 15]
-}

```

```

-----
-- Ejercicio 6.4. A partir del ejercicio anterior, definir de manera más
-- eficiente la función numeroDeListasConSuma.
-----

```

```

numeroDeListasConSuma' :: Int -> Int
numeroDeListasConSuma' 0 = 1
numeroDeListasConSuma' n = 2^(n-1)

```

```

-----
-- Ejercicio 6.5. Comparar la eficiencia de las dos definiciones

```



```
-- comparando el tiempo y el espacio usado para calcular
-- (numeroDeListasConSuma 20) y (numeroDeListasConSuma' 20).
```

```
-----
-- La comparación es
-- ghci> :set +s
-- ghci> numeroDeListasConSuma 20
-- 524288
-- (9.99 secs, 519419824 bytes)
-- ghci> numeroDeListasConSuma' 20
-- 524288
-- (0.01 secs, 0 bytes)
```

```
-----
-- Ejercicio 7.0. La sucesión de Fibonacci
-- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
-- puede definirse por recursión como
-- fib :: Int -> Int
-- fib 0 = 0 -- fib.1
-- fib 1 = 1 -- fib.2
-- fib n = (fib (n-1)) + fib (n-2) -- fib.3
-- También puede definirse por recursión iterativa como
-- fibIt :: Int -> Int
-- fibIt n = fibItAux n 0 1
-- donde la función auxiliar se define por
-- fibItAux :: Int -> Int -> Int -> Int
-- fibItAux 0 a b = a -- fibItAux.1
-- fibItAux n a b = fibItAux (n-1) b (a+b) -- fibItAux.2
```

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

```
fibIt :: Int -> Int
fibIt n = fibItAux n 0 1
```

```
fibItAux :: Int -> Int -> Int -> Int
fibItAux 0 a b = a
```

```
fibItAux n a b = fibItAux (n-1) b (a+b)
```

```
-----
-- Ejercicio 7.1. Comprobar con QuickCheck que para todo número natural
-- n tal que n <= 20, se tiene que
--   fib n = fibIt n
-----
```

```
-- La propiedad es
```

```
prop_fib :: Int -> Property
```

```
prop_fib n =
```

```
  n >= 0 && n <= 20 ==> fib n == fibIt n
```

```
-- La comprobación es
```

```
--   ghci> quickCheck prop_fib
```

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

```
-----
-- Ejercicio 7.2. Sea f la función definida por
```

```
--   f :: Int -> Int -> Int
```

```
--   f n k = fibItAux n (fib k) (fib (k+1))
```

```
-- Definir la función
```

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

```
-- tal que (grafoDeF n)
```

```
--   ghci> take 7 (grafoDeF 3)
```

```
--   [(1,3), (2,5), (3,8), (4,13), (5,21), (6,34), (7,55)]
```

```
--   ghci> take 7 (grafoDeF 5)
```

```
--   [(1,8), (2,13), (3,21), (4,34), (5,55), (6,89), (7,144)]
```

```
f :: Int -> Int -> Int
```

```
f n k = fibItAux n (fib k) (fib (k+1))
```

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

```
grafoDeF n = [(k, f n k) | k <- [1..]]
```

```
-----
-- Ejercicio 7.3. Comprobar con QuickCheck que para todo par de números
```

```
-- naturales n, k tales que n+k <= 20, se tiene que
```

```
--   fibItAux n (fib k) (fib (k+1)) = fib (k+n)
```

```

-----
-- La propiedad es
prop_fibItAux :: Int -> Int -> Property
prop_fibItAux n k =
  n >= 0 && k >= 0 && n+k <= 20 ==>
  fibItAux n (fib k) (fib (k+1)) == fib (k+n)

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

-----
-- Ejercicio 7.4. Demostrar por inducción que para todo n y todo k,
--   fibItAux n (fib k) (fib (k+1)) = fib (k+n)
-----

{-
  Demostración: Por inducción en n se prueba que
    para todo k, fibItAux n (fib k) (fib (k+1)) = fib (k+n)

  Caso base (n=0): Hay que demostrar que
    para todo k, fibItAux 0 (fib k) (fib (k+1)) = fib k
  En efecto, sea k un número natural. Se tiene
    fibItAux 0 (fib k) (fib (k+1))
  = fib k                                [por fibItAux.1]

  Paso de inducción: Se supone la hipótesis de inducción
    para todo k, fibItAux n (fib k) (fib (k+1)) = fib (k+n)
  Hay que demostrar que
    para todo k, fibItAux (n+1) (fib k) (fib (k+1)) = fib (k+n+1)
  En efecto. Sea k un número natural,
    fibItAux (n+1) (fib k) (fib (k+1))
  = fibItAux n (fib (k+1)) ((fib k) + (fib (k+1)))
    [por fibItAux.2]
  = fibItAux n (fib (k+1)) (fib (k+2))
    [por fib.3]
  = fib (n+k+1)
    [por hipótesis de inducción]
-}

```

```

-----
-- Ejercicio 7.5. Demostrar que para todo n,
--   fibIt n = fib n
-----

{-
  Demostración
  fibIt n
  = fibItAux n 0 1           [por fibIt]
  = fibItAux n (fib 0) (fib 1) [por fib.1 y fib.2]
  = fib n                   [por ejercicio 5.4]
-}

-----

-- Ejercicio 8.1. La función potencia puede definirse por
--   potencia :: Int -> Int -> Int
--   potencia x 0 = 1
--   potencia x n | even n    = potencia (x*x) (div n 2)
--                 | otherwise = x * potencia (x*x) (div n 2)
-- Comprobar con QuickCheck que para todo número natural n y todo
-- número entero x, (potencia x n) es x^n.
-----

potencia :: Integer -> Integer -> Integer
potencia x 0 = 1
potencia x n | even n    = potencia (x*x) (div n 2)
             | otherwise = x * potencia (x*x) (div n 2)

-- La propiedad es
prop_potencia :: Integer -> Integer -> Property
prop_potencia x n =
  n >= 0 ==> potencia x n == x^n

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

-- Ejercicio 8.2. Demostrar por inducción que para todo número

```

-- natural  $n$  y todo número entero  $x$ , (potencia  $x$   $n$ ) es  $x^n$

-----

{-

*Demostración: Por inducción en  $n$ .*

*Caso base: Hay que demostrar que  
para todo  $x$ , potencia  $x$   $0 = 2^0$*

*Sea  $x$  un número entero, entonces*

$$\begin{aligned} \text{potencia } x \ 0 & \\ &= 1 && \text{[por potencia.1]} \\ &= 2^0 && \text{[por aritmética]} \end{aligned}$$

*Paso de inducción: Se supone que  $n > 0$  y la hipótesis de inducción:  
para todo  $m < n$  y para todo  $x$ , potencia  $x$   $(n-1) = x^{(n-1)}$*

*Tenemos que demostrar que*

$$\text{para todo } x, \text{ potencia } x \ n = x^n$$

*Lo haremos distinguiendo casos según la paridad de  $n$ .*

*Caso 1: Supongamos que  $n$  es par. Entonces, existe un  $k$  tal que  
 $n = 2 \cdot k$ . (1)*

*Por tanto,*

$$\begin{aligned} \text{potencia } n & \\ &= \text{potencia } (x \cdot x) \ (\text{div } n \ 2) && \text{[por potencia.2]} \\ &= \text{potencia } (x \cdot x) \ k && \text{[por (1)]} \\ &= (x \cdot x)^k && \text{[por hip. de inducción]} \\ &= x^{(2 \cdot k)} && \text{[por aritmética]} \\ &= x^n && \text{[por (1)]} \end{aligned}$$

*Caso 2: Supongamos que  $n$  es impar. Entonces, existe un  $k$  tal que  
 $n = 2 \cdot k + 1$ . (2)*

*Por tanto,*

$$\begin{aligned} \text{potencia } n & \\ &= x \cdot \text{potencia } (x \cdot x) \ (\text{div } n \ 2) && \text{[por potencia.3]} \\ &= x \cdot \text{potencia } (x \cdot x) \ k && \text{[por (1)]} \\ &= x \cdot (x \cdot x)^k && \text{[por hip. de inducción]} \\ &= x^{(2 \cdot k + 1)} && \text{[por aritmética]} \\ &= x^n && \text{[por (1)]} \end{aligned}$$

-}

```

-----
-- Ejercicio 9.1. Comprobar con QuickCheck que para todo par de listas
-- xs, ys se tiene que
--   reverse (xs ++ ys) == reverse ys ++ reverse xs
-----

```

```

-- La propiedad es
prop_reverse_conc :: [Int] -> [Int] -> Bool
prop_reverse_conc xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs

```

```

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

```

```

-----
-- Ejercicio 9.2. Demostrar por inducción que para todo par de listas
-- xs, ys se tiene que
--   reverse (xs ++ ys) == reverse ys ++ reverse xs
--
-- Las definiciones de reverse y (++) son
--   reverse [] = []           -- reverse.1
--   reverse (x:xs) = reverse xs ++ [x]  -- reverse.2
--
--   [] ++ ys      = ys           -- ++.1
--   (x:xs) ++ ys = x : (xs ++ ys) -- ++.2
-----

```

```

{-
  Demostración por inducción en xs.

```

Caso base: Hay que demostrar que para toda ys,

$$\text{reverse } ([] ++ \text{ys}) == \text{reverse ys} ++ \text{reverse } []$$

En efecto,

```

reverse ([] ++ ys)
= reverse ys           [por ++.1]
= reverse ys ++ []    [por propiedad de ++]
= reverse ys ++ reverse [] [por reverse.1]

```

Paso de inducción: Se supone que para todo ys,

```

    reverse (xs ++ ys) == reverse ys ++ reverse xs
Hay que demostrar que para todo ys,
    reverse ((x:xs) ++ ys) == reverse ys ++ reverse (x:xs)
En efecto,
    reverse ((x:xs) ++ ys)
= reverse (x:(xs ++ ys))           [por ++.2]
= reverse (xs ++ ys) ++ [x]       [por reverse.2]
= (reverse ys ++ reverse xs) ++ [x] [por hip. de inducción]
= reverse ys ++ (reverse xs ++ [x]) [por asociativa de ++]
= reverse ys ++ reverse (x:xs)     [por reverse.2]
-}

-----
-- Ejercicio 9.3. Demostrar por inducción que para toda lista xs,
--    reverse (reverse xs) = xs
-----

{-
Demostración por inducción en xs.

Caso Base: Hay que demostrar que
    reverse (reverse []) = []
En efecto,
    reverse (reverse [])
= reverse []           [por reverse.1]
= []                   [por reverse.1]

Paso de inducción: Se supone que
    reverse (reverse xs) = xs
Hay que demostrar que
    reverse (reverse (x:xs)) = x:xs
En efecto,
    reverse (reverse (x:xs))
= reverse (reverse xs ++ [x])           [por reverse.2]
= reverse [x] ++ reverse (reverse xs)   [por ejercicio 7.2]
= [x] ++ reverse (reverse xs)          [por reverse]
= [x] ++ xs                             [por hip. de inducción]
= x:xs                                   [por ++.2]
-}

```

```

-----
-- Ejercicio 10.0. En los siguientes ejercicios se demostrarán
-- propiedades de los árboles binarios definidos como sigue
--   data Arbol a = Hoja
--                 | Nodo a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- En los ejemplos se usará el siguiente árbol
--   arbol = Nodo 9
--           (Nodo 3
--            (Nodo 2 Hoja Hoja)
--            (Nodo 4 Hoja Hoja))
--           (Nodo 7 Hoja Hoja)
-----

```

```

data Arbol a = Hoja
  | Nodo a (Arbol a) (Arbol a)
  deriving (Show, Eq)

```

```

arbol = Nodo 9
  (Nodo 3
   (Nodo 2 Hoja Hoja)
   (Nodo 4 Hoja Hoja))
  (Nodo 7 Hoja Hoja)

```

```

-----
-- Nota. Para comprobar propiedades de árboles con QuickCheck se
-- utilizará el siguiente generador.
-----

```

```

instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbol
  where
    arbol 0      = return Hoja
    arbol n | n>0 = oneof [return Hoja,
                          liftM3 Nodo arbitrary subarbol subarbol]
                      where subarbol = arbol (div n 2)
  -- coarbitrary = undefined

```

```

-----
-- Ejercicio 10.1. Definir la función

```



```
--     espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--     ghci> espejo arbol
--     Nodo 9
--           (Nodo 7 Hoja Hoja)
--           (Nodo 3
--            (Nodo 4 Hoja Hoja)
--            (Nodo 2 Hoja Hoja))
```

```
-----
espejo :: Arbol a -> Arbol a
espejo Hoja      = Hoja
espejo (Nodo x i d) = Nodo x (espejo d) (espejo i)
```

```
-----
-- Ejercicio 10.2. Comprobar con QuickCheck que para todo árbol x,
--     espejo (espejo x) = x
-----
```

```
prop_espejo :: Arbol Int -> Bool
prop_espejo x =
  espejo (espejo x) == x
```

```
-----
-- Ejercicio 10.3. Demostrar por inducción que para todo árbol x,
--     espejo (espejo x) = x
-----
```

```
{-
  Demostración por inducción en x
```

Caso base: Hay que demostrar que  
 $\text{espejo (espejo Hoja)} = \text{Hoja}$

En efecto,

```
espejo (espejo Hoja)
= espejo Hoja      [por espejo.1]
= Hoja             [por espejo.1]
```

Paso de inducción: Se supone la hipótesis de inducción  
 $\text{espejo (espejo i)} = i$

```

    espejo (espejo d) = d
Hay que demostrar que
    espejo (espejo (Nodo x i d)) = Nodo x i d
En efecto,
    espejo (espejo (Nodo x i d))
= espejo (Nodo x (espejo d) (espejo i))           [por espejo.2]
= Nodo x (espejo (espejo i)) (espejo (espejo d)) [por espejo.2]
= Nodo x i d                                       [por hip. inducción]
-}

-----
-- Ejercicio 10.4. Definir la función
--   preorden :: Arbol a -> [a]
-- tal que (preorden x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> preorden arbol
--   [9,3,2,4,7]
-----

preorden :: Arbol a -> [a]
preorden Hoja      = []
preorden (Nodo x i d) = x : (preorden i ++ preorden d)

-----
-- Ejercicio 10.5. Definir la función
--   postorden :: Arbol a -> [a]
-- tal que (postorden x) es la lista correspondiente al recorrido
-- postorden del árbol x; es decir, primero recorre el subárbol
-- izquierdo, a continuación el subárbol derecho y, finalmente, la raíz
-- del árbol. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> postorden arbol
--   [2,4,3,7,9]
-----

```

```

postorden :: Arbol a -> [a]
postorden Hoja      = []
postorden (Nodo x i d) = postorden i ++ postorden d ++ [x]

```

```

-----
-- Ejercicio 10.6. Comprobar con QuickCheck que para todo árbol x,
--   postorden (espejo x) = reverse (preorden x)
-----

```

```

-- La propiedad es
prop_recorrido :: Arbol Int -> Bool
prop_recorrido x =
  postorden (espejo x) == reverse (preorden x)

```

```

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

```

```

-----
-- Ejercicio 10.7. Demostrar por inducción que para todo árbol x,
--   postorden (espejo x) = reverse (preorden x)
-----

```

```

{-
  Demostración por inducción en x.

```

Caso base: Hay que demostrar que

$$\text{postorden (espejo Hoja)} = \text{reverse (preorden Hoja)}$$

En efecto,

$$\begin{aligned}
\text{postorden (espejo Hoja)} &= \text{postorden Hoja} && \text{[por espejo.1]} \\
&= [] && \text{[por postorden.1]} \\
&= \text{reverse } [] && \text{[por reverse.1]} \\
&= \text{reverse (preorden Hoja)} && \text{[por preorden.1]}
\end{aligned}$$

Paso de inducción: Se supone la hipótesis de inducción

$$\begin{aligned}
\text{postorden (espejo } i) &= \text{reverse (preorden } i) \\
\text{postorden (espejo } d) &= \text{reverse (preorden } d)
\end{aligned}$$

Hay que demostrar que

$$\text{postorden (espejo (Nodo } x \ i \ d))} = \text{reverse (preorden (Nodo } x \ i \ d))}$$

En efecto,

```

postorden (espejo (Nodo x i d))
= postorden (Nodo x (espejo d) (espejo i)) [por espejo.2]
= postorden (espejo d) ++ postorden (espejo i) ++ [x]
                                           [por postorden.2]
= reverse (preorden d) ++ reverse (preorden i) ++ [x]
                                           [por hip. inducción]
= reverse ([x] ++ preorden (espejo i) ++ preorden (espejo d))
                                           [por ejercicio 7.1]
= reverse (preorden (Nodo x i d)) [por preorden.1]
-}

```

```

-----
-- Ejercicio 10.8. Comprobar con QuickCheck que para todo árbol binario
-- x, se tiene que
--   reverse (preorden (espejo x)) = postorden x
-----

```

```

-- La propiedad es
prop_reverse_preorden_espejo :: Arbol Int -> Bool
prop_reverse_preorden_espejo x =
  reverse (preorden (espejo x)) == postorden x

```

```

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

```

```

-----
-- Ejercicio 10.9. Demostrar que para todo árbol binario x, se tiene que
--   reverse (preorden (espejo x)) = preorden x
-----

```

```

{-
Demostración:
  reverse (preorden (espejo x))
= postorden (espejo (espejo x)) [por ejercicio 8.7]
= postorden x [por ejercicio 8.3]
-}

```

```
-- Ejercicio 10.10. Definir la función
--   nNodos :: Arbol a -> Int
-- tal que (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> nNodos arbol
--   5
```

```
-----
nNodos :: Arbol a -> Int
nNodos Hoja          = 0
nNodos (Nodo x i d) = 1 + nNodos i + nNodos d
```

```
-----
-- Ejercicio 10.11. Comprobar con QuickCheck que el número de nodos de la
-- imagen especular de un árbol es el mismo que el número de nodos del
-- árbol.
```

```
-----
-- La propiedad es
prop_nNodos_espejo :: Arbol Int -> Bool
prop_nNodos_espejo x =
  nNodos (espejo x) == nNodos x
```

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

```
-----
-- Ejercicio 10.12. Demostrar por inducción que el número de nodos de la
-- imagen especular de un árbol es el mismo que el número de nodos del
-- árbol.
```

```
{-
  Demostración: Hay que demostrar, por inducción en x, que
    nNodos (espejo x) == nNodos x
```

```
  Caso base: Hay que demostrar que
    nNodos (espejo Hoja) == nNodos Hoja
```

```

En efecto,
  nNodos (espejo Hoja)
    = nNodos Hoja           [por espejo.1]

```

Paso de inducción: Se supone la hipótesis de inducción

```

nNodos (espejo i) == nNodos i
nNodos (espejo d) == nNodos d

```

Hay que demostrar que

```

nNodos (espejo (Nodo x i d)) == nNodos (Nodo x i d)

```

En efecto,

```

nNodos (espejo (Nodo x i d))
  = nNodos (Nodo x (espejo d) (espejo i))           [por espejo.2]
  = 1 + nNodos (espejo d) + nNodos (espejo i)       [por nNodos.2]
  = 1 + nNodos d + nNodos i                           [por hip.de inducción]
  = 1 + nNodos i + nNodos d                           [por aritmética]
  = nNodos (Nodo x i d)                               [por nNodos.2]

```

```
-}
```

```

-----
-- Ejercicio 10.13. Comprobar con QuickCheck que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol.
-----

```

-- La propiedad es

```

prop_length_preorden :: Arbol Int -> Bool
prop_length_preorden x =
  length (preorden x) == nNodos x

```

-- La comprobación es

```

-- ghci> quickCheck prop_length_preorden
-- OK, passed 100 tests.

```

```

-----
-- Ejercicio 10.14. Demostrar por inducción que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol.
-----

```

```
{-
```

*Demostración: Por inducción en  $x$ , hay que demostrar que*  
 $\text{length (preorden } x) == \text{nNodos } x$

*Caso base: Hay que demostrar que*  
 $\text{length (preorden Hoja)} = \text{nNodos Hoja}$

*En efecto,*  
 $\text{length (preorden Hoja)}$   
 $= \text{length []}$  [por preorden.1]  
 $= 0$  [por length.1]  
 $= \text{nNodos Hoja}$  [por nNodos.1]

*Paso de inducción: Se supone la hipótesis de inducción*

$\text{length (preorden } i) == \text{nNodos } i$   
 $\text{length (preorden } d) == \text{nNodos } d$

*Hay que demostrar que*

$\text{length (preorden (Nodo } x \ i \ d)) == \text{nNodos (Nodo } x \ i \ d)$

*En efecto,*

$\text{length (preorden (Nodo } x \ i \ d))$   
 $= \text{length ([x] ++ (preorden } i) ++ (preorden } d))$   
[por preorden.2]  
 $= \text{length [x]} + \text{length (preorden } i) + \text{length (preorden } d)$   
[propiedad de length:  $\text{length (xs++ys)} = \text{length xs} + \text{length ys}$ ]  
 $= 1 + \text{length (preorden } i) + \text{length (preorden } d)$   
[por def. de length]  
 $= 1 + \text{nNodos } i + \text{nNodos } d$   
[por hip. de inducción]  
 $= \text{nNodos (x } i \ d)$   
[por nNodos.2]

-}

-----  
-- Ejercicio 10.15. Definir la función

--  $\text{profundidad} :: \text{Arbol } a \rightarrow \text{Int}$

-- tal que  $(\text{profundidad } x)$  es la profundidad del árbol  $x$ . Por ejemplo,

--  $\text{ghci} > \text{arbol}$

--  $\text{Nodo } 9 \ (\text{Nodo } 3 \ (\text{Nodo } 2 \ \text{Hoja } \ \text{Hoja}) \ (\text{Nodo } 4 \ \text{Hoja } \ \text{Hoja})) \ (\text{Nodo } 7 \ \text{Hoja } \ \text{Hoja})$

--  $\text{ghci} > \text{profundidad arbol}$

-- 3  
-----

```

profundidad :: Arbol a -> Int
profundidad Hoja = 0
profundidad (Nodo x i d) = 1 + max (profundidad i) (profundidad d)

-----
-- Ejercicio 10.16. Comprobar con QuickCheck que para todo árbol biario
-- x, se tiene que
--   nNodos x <= 2^(profundidad x) - 1
-----

-- La propiedad es
prop_nNodosProfundidad :: Arbol Int -> Bool
prop_nNodosProfundidad x =
  nNodos x <= 2^(profundidad x) - 1

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

-----
-- Ejercicio 10.17. Demostrar por inducción que para todo árbol binario
-- x, se tiene que
--   nNodos x <= 2^(profundidad x) - 1
-----

{-
  Demostración por inducción en x

  Caso base: Hay que demostrar que
    nNodos Hoja <= 2^(profundidad Hoja) - 1
  En efecto,
    nNodos Hoja
    = 0                                [por nNodos.1]
    = 2^0 - 1                          [por aritmética]
    = 2^(profundidad Hoja) - 1        [por profundidad.1]

  Paso de inducción: Se supone la hipótesis de inducción
    nNodos i <= 2^(profundidad i) - 1
    nNodos d <= 2^(profundidad d) - 1
  Hay que demostrar que

```



```

    nNodos (Nodo x i d) <= 2^(profundidad (Nodo x i d)) - 1
En efecto,
nNodos (Nodo x i d)
= 1 + nNodos i + nNodos d
  [por nNodos.1]
<= 1 + (2^(profundidad i) - 1) + (2^(profundidad d) - 1)
  [por hip. de inducción]
= 2^(profundidad i) + 2^(profundidad d) - 1
  [por aritmética]
<= 2^máx(profundidad i,profundidad d)+2^máx(profundidad i,profundidad d)-1
  [por aritmética]
= 2*2^máx(profundidad i,profundidad d) - 1
  [por aritmética]
= 2^(1+máx(profundidad i,profundidad d)) - 1
  [por aritmética]
= 2^profundidad(Nodo x i d) - 1
  [por profundidad.2]
-}

```

```

-----
-- Ejercicio 10.18. Definir la función
--   nHojas :: Arbol a -> Int
-- tal que (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> nHojas arbol
--   6
-----

```

```

nHojas :: Arbol a -> Int
nHojas Hoja      = 1
nHojas (Nodo x i d) = nHojas i + nHojas d

```

```

-----
-- Ejercicio 10.19. Comprobar con QuickCheck que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-----

```

```

-- La propiedad es
prop_nHojas :: Arbol Int -> Bool

```

```

prop_nHojas x =
  nHojas x == nNodos x + 1

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

-----
-- Ejercicio 10.20. Demostrar por inducción que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-----

{-
Demostración: Hay que demostrar, por inducción en x, que
  nHojas x = nNodos x + 1

Caso base: Hay que demostrar que
  nHojas Hoja = nNodos Hoja + 1
En efecto,
  nHojas Hoja
= 1                [por nHojas.1]
= 0 + 1           [por aritmética]
= nNodos Hoja + 1 [por nNodos.1]

Paso de inducción: Se supone la hipótesis de inducción
  nHojas i = nNodos i + 1
  nHojas d = nNodos d + 1
Hay que demostrar que
  nHojas (Nodo x i d) = nNodos (Nodo x i d) + 1
En efecto,
  nHojas (Nodo x i d)
= nHojas i + nHojas d                [por nHojas.2]
= (nNodos i + 1) + (nNodos d + 1)    [por hip. de inducción]
= (1 + nNodos i + nNodos d) + 1     [por aritmética]
= nNodos (Nodo x i d) + 1           [por nNodos.2]
-}

-----
-- Ejercicio 10.21. Definir, usando un acumulador, la función
--   preordenIt :: Arbol a -> [a]

```

```

-- tal que (preordenIt x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--   ghci> arbol
--   Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--   ghci> preordenIt arbol
--   [9,3,2,4,7]
-- Nota: No usar (++) en la definición
-----

```

```

preordenIt :: Arbol a -> [a]
preordenIt x = preordenItAux x []

```

```

preordenItAux :: Arbol a -> [a] -> [a]
preordenItAux Hoja xs          = xs
preordenItAux (Nodo x i d) xs = x : preordenItAux i (preordenItAux d xs)

```

```

-----
-- Ejercicio 10.22. Comprobar con QuickCheck que preordenIt es
-- equivalente a preorden.
-----

```

```

-- La propiedad es
prop_preordenIt :: Arbol Int -> Bool
prop_preordenIt x =
  preordenIt x == preorden x

```

```

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

```

```

-----
-- Ejercicio 10.22. Demostrar que preordenIt es equivalente a preorden.
-----

```

```

prop_preordenItAux :: Arbol Int -> [Int] -> Bool
prop_preordenItAux x ys =
  preordenItAux x ys == preorden x ++ ys

```

{-

*Demostración: La propiedad es consecuencia del siguiente lema:*

*Lema: Para todo árbol binario  $x$ , se tiene que  
para toda  $ys$ ,  $\text{preordenItAux } x \text{ } ys = \text{preorden } x \text{ } ++ \text{ } ys$*

*Demostración de la propiedad usando el lema:*

```
preordenIt x
= preordenItAux x []      [por preordnIt]
= preorden x ++ []      [por el lema]
= preorden x              [propiedad de ++]
```

*Demostración del lema: Por inducción en  $x$ .*

*Caso base: Hay que demotrar que*

*para toda  $ys$ ,  $\text{preordenItAux Hoja } ys = \text{preorden Hoja } ++ \text{ } ys$*

*En efecto,*

```
preordenItAux Hoja ys
= ys                      [por preordenItAux.1]
= [] ++ ys                [por propiedad de ++]
= preorden Hoja ++ ys    [por preorden.1]
```

*Paso de inducción: Se supone la hipótesis de inducción*

*para toda  $ys$ ,  $\text{preordenItAux } i \text{ } ys = \text{preorden } i \text{ } ++ \text{ } ys$*

*para toda  $ys$ ,  $\text{preordenItAux } d \text{ } ys = \text{preorden } d \text{ } ++ \text{ } ys$*

*Hay que demostrar que*

*para toda  $ys$ ,  $\text{preordenItAux } (\text{Nodo } x \text{ } i \text{ } d) \text{ } ys = \text{preorden } (\text{Nodo } x \text{ } i \text{ } d) \text{ } ++ \text{ } ys$*

*En efecto,*

```
preordenItAux (Nodo x i d) ys
= x : (preordenItAux i (preordenItAux d ys))  [por preordenItAux.2]
= x : (preordenItAux i (preorden d ++ ys))    [por hip. de inducción]
= x : (preorden i ++ (preorden d ++ ys))     [por hip. de inducción]
= ([x] ++ preorden i ++ preorden d) ++ ys    [por prop. de listas]
= preorden (Nodo x i d) ++ ys                [por preorden.2]
```

-}

# Relación 32

## Relaciones binarias homogéneas

```
-----  
-- Introducción --  
-----  
  
-- El objetivo de esta relación de ejercicios es definir propiedades y  
-- operaciones sobre las relaciones binarias (homogéneas).  
--  
-- Como referencia se puede usar el artículo de la wikipedia  
-- http://bit.ly/HVHOPS  
  
-----  
-- § Librerías auxiliares --  
-----  
  
import Test.QuickCheck  
import Data.List  
  
-----  
-- Ejercicio 1. Una relación binaria R sobre un conjunto A puede  
-- representar mediante un par (xs,ps) donde xs es la lista de los  
-- elementos de A (el universo de R) y ps es la lista de pares de R (el  
-- grafo de R). Definir el tipo de dato (Rel a) para representar las  
-- relaciones binarias sobre a.  
-----  
  
type Rel a = ([a],[(a,a)])
```

```

-----
-- Nota. En los ejemplos usaremos las siguientes relaciones binarias:
--   r1, r2, r3 :: Rel Int
--   r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
--   r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
--   r3 = ([1..9],[(1,3), (2,6), (8,9), (3,6)])
-----

```

```

r1, r2, r3 :: Rel Int
r1 = ([1..9],[(1,3), (2,6), (8,9), (2,7)])
r2 = ([1..9],[(1,3), (2,6), (8,9), (3,7)])
r3 = ([1..9],[(1,3), (2,6), (8,9), (3,6)])

```

```

-----
-- Ejercicio 2. Definir la función
--   universo :: Eq a => Rel a -> [a]
-- tal que (universo r) es el universo de la relación r. Por ejemplo,
--   r1           == ([1,2,3,4,5,6,7,8,9],[(1,3), (2,6), (8,9), (2,7)])
--   universo r1 == [1,2,3,4,5,6,7,8,9]
-----

```

```

universo :: Eq a => Rel a -> [a]
universo (us,_) = us

```

```

-----
-- Ejercicio 3. Definir la función
--   grafo :: Eq a => ([a],[(a,a)]) -> [(a,a)]
-- tal que (grafo r) es el grafo de la relación r. Por ejemplo,
--   r1           == ([1,2,3,4,5,6,7,8,9],[(1,3), (2,6), (8,9), (2,7)])
--   grafo r1    == [(1,3), (2,6), (8,9), (2,7)]
-----

```

```

grafo :: Eq a => ([a],[(a,a)]) -> [(a,a)]
grafo (_,ps) = ps

```

```

-----
-- Ejercicio 4. Definir la función
--   reflexiva :: Eq a => Rel a -> Bool
-- tal que (reflexiva r) se verifica si la relación r es reflexiva. Por
-- ejemplo,

```

```
-- reflexiva ([1,3],[(1,1),(1,3),(3,3)]) == True
-- reflexiva ([1,2,3],[(1,1),(1,3),(3,3)]) == False
```

```
-----
reflexiva :: Eq a => Rel a -> Bool
reflexiva (us,ps) = and [elem (x,x) ps | x <- us]
```

```
-----
-- Ejercicio 5. Definir la función
-- simetrica :: Eq a => Rel a -> Bool
-- tal que (simetrica r) se verifica si la relación r es simétrica. Por
-- ejemplo,
-- simetrica ([1,3],[(1,1),(1,3),(3,1)]) == True
-- simetrica ([1,3],[(1,1),(1,3),(3,2)]) == False
-- simetrica ([1,3],[]) == True
```

```
-----
simetrica :: Eq a => Rel a -> Bool
simetrica (us,ps) = and [(y,x) 'elem' ps | (x,y) <- ps]
```

```
-----
-- Ejercicio 6. Definir la función
-- subconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys. Por ejemplo,
-- subconjunto [1,3] [3,1,5] == True
-- subconjunto [3,1,5] [1,3] == False
```

```
-----
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]
```

```
-----
-- Ejercicio 7. Definir la función
-- composicion :: Eq a => Rel a -> Rel a -> Rel a
-- tal que (composicion r s) es la composición de las relaciones r y
-- s. Por ejemplo,
-- ghci> composicion ([1,2],[(1,2),(2,2)]) ([1,2],[(2,1)])
-- ([1,2],[(1,1),(2,1)])
```

```

composicion :: Eq a => Rel a -> Rel a -> Rel a
composicion (xs,ps) (_,qs) =
  (xs,[(x,z) | (x,y) <- ps, (y',z) <- qs, y == y'])

```

```

-----
-- Ejercicio 8. Definir la función
--   transitiva :: Eq a => Rel a -> Bool
-- tal que (transitiva r) se verifica si la relación r es transitiva.
-- Por ejemplo,
--   transitiva ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)]) == True
--   transitiva ([1,3,5],[(1,1),(1,3),(3,1),(5,5)])       == False
-----

```

```

transitiva :: Eq a => Rel a -> Bool
transitiva r@(xs,ps) =
  subconjunto (grafo (composicion r r)) ps

```

```

-----
-- Ejercicio 9. Definir la función
--   esEquivalencia :: Eq a => Rel a -> Bool
-- tal que (esEquivalencia r) se verifica si la relación r es de
-- equivalencia. Por ejemplo,
--   ghci> esEquivalencia ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])
--   True
--   ghci> esEquivalencia ([1,2,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])
--   False
--   ghci> esEquivalencia ([1,3,5],[(1,1),(1,3),(3,3),(5,5)])
--   False
-----

```

```

esEquivalencia :: Eq a => Rel a -> Bool
esEquivalencia r = reflexiva r && simetrica r && transitiva r

```

```

-----
-- Ejercicio 10. Definir la función
--   irreflexiva :: Eq a => Rel a -> Bool
-- tal que (irreflexiva r) se verifica si la relación r es irreflexiva;
-- es decir, si ningún elemento de su universo está relacionado con
-- él mismo. Por ejemplo,

```



```
-- irreflexiva ([1,2,3],[(1,2),(2,1),(2,3)]) == True
-- irreflexiva ([1,2,3],[(1,2),(2,1),(3,3)]) == False
```

```
-----
irreflexiva :: Eq a => Rel a -> Bool
irreflexiva (xs,ps) = and [(x,x) 'notElem' ps | x <- xs]
```

```
-----
-- Ejercicio 11. Definir la función
-- antisimetrica :: Eq a => Rel a -> Bool
-- tal que (antisimetrica r) se verifica si la relación r es
-- antisimétrica; es decir, si (x,y) e (y,x) están relacionado, entonces
-- x=y. Por ejemplo,
-- antisimetrica ([1,2],[(1,2)]) == True
-- antisimetrica ([1,2],[(1,2),(2,1)]) == False
-- antisimetrica ([1,2],[(1,1),(2,1)]) == True
```

```
-----
antisimetrica :: Eq a => Rel a -> Bool
antisimetrica (_,ps) =
  null [(x,y) | (x,y) <- ps, x /= y, (y,x) 'elem' ps]
```

```
-- Otra definición es
antisimetrica' :: Eq a => Rel a -> Bool
antisimetrica' (xs,ps) =
  and [(x,y) 'elem' ps && (y,x) 'elem' ps --> (x == y)
       | x <- xs, y <- xs]
  where p --> q = not p || q
```

```
-- Las dos definiciones son equivalentes
prop_antisimetrica :: Rel Int -> Bool
prop_antisimetrica r =
  antisimetrica r == antisimetrica' r
```

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

```
-----
-- Ejercicio 12. Definir la función
```

```

-- total :: Eq a => Rel a -> Bool
-- tal que (total r) se verifica si la relación r es total; es decir, si
-- para cualquier par x, y de elementos del universo de r, se tiene que
-- x está relacionado con y ó y está relacionado con x. Por ejemplo,
-- total ([1,3],[(1,1),(3,1),(3,3)]) == True
-- total ([1,3],[(1,1),(3,1)]) == False
-- total ([1,3],[(1,1),(3,3)]) == False

```

```

total :: Eq a => Rel a -> Bool

```

```

total (xs,ps) =
  and [(x,y) 'elem' ps || (y,x) 'elem' ps | x <- xs, y <- xs]

```

```

-- -----
-- Ejercicio 13. Comprobar con QuickCheck que las relaciones totales son
-- reflexivas.
-- -----

```

```

prop_total_reflexiva :: Rel Int -> Property

```

```

prop_total_reflexiva r =
  total r ==> reflexiva r

```

```

-- La comprobación es
-- ghci> quickCheck prop_total_reflexiva
-- *** Gave up! Passed only 19 tests.

```

```

-- § Clausuras
-- -----

```

```

-- Ejercicio 14. Definir la función

```

```

-- clausuraReflexiva :: Eq a => Rel a -> Rel a
-- tal que (clausuraReflexiva r) es la clausura reflexiva de r; es
-- decir, la menor relación reflexiva que contiene a r. Por ejemplo,
-- ghci> clausuraReflexiva ([1,3],[(1,1),(3,1)])
-- ([1,3],[(1,1),(3,1),(3,3)])

```

```

clausuraReflexiva :: Eq a => Rel a -> Rel a

```

```
clausuraReflexiva (xs,ps) =
  (xs, ps 'union' [(x,x) | x <- xs])
```

```
-----
-- Ejercicio 15. Comprobar con QuickCheck que clausuraReflexiva es
-- reflexiva.
-----
```

```
prop_ClausuraReflexiva :: Rel Int -> Bool
prop_ClausuraReflexiva r =
  reflexiva (clausuraReflexiva r)
```

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

```
-----
-- Ejercicio 16. Definir la función
-- clausuraSimetrica :: Eq a => Rel a -> Rel a
-- tal que (clausuraSimetrica r) es la clausura simétrica de r; es
-- decir, la menor relación simétrica que contiene a r. Por ejemplo,
-- ghci> clausuraSimetrica ([1,3,5],[(1,1),(3,1),(1,5)])
-- [(1,3,5],[(1,1),(3,1),(1,5),(1,3),(5,1)])
-----
```

```
clausuraSimetrica :: Eq a => Rel a -> Rel a
clausuraSimetrica (xs,ps) =
  (xs, ps 'union' [(y,x) | (x,y) <- ps])
```

```
-----
-- Ejercicio 17. Comprobar con QuickCheck que clausuraSimetrica es
-- simétrica.
-----
```

```
prop_ClausuraSimetrica :: Rel Int -> Bool
prop_ClausuraSimetrica r =
  simetrica (clausuraSimetrica r)
```

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

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

-- -----
-- Ejercicio 18. Definir la función
--   clausuraTransitiva :: Eq a => Rel a -> Rel a
--   tal que (clausuraTransitiva r) es la clausura transitiva de r; es
--   decir, la menor relación transitiva que contiene a r. Por ejemplo,
--   ghci> clausuraTransitiva ([1..6],[(1,2),(2,5),(5,6)])
--   ([1,2,3,4,5,6],[(1,2),(2,5),(5,6),(1,5),(2,6),(1,6)])
-- -----
```

```
clausuraTransitiva :: Eq a => Rel a -> Rel a
clausuraTransitiva (xs,ps) = (xs, aux ps)
  where aux xs | cerradoTr xs = xs
             | otherwise     = aux (xs 'union' (comp xs xs))
  cerradoTr r = subconjunto (comp r r) r
  comp r s    = [(x,z) | (x,y) <- r, (y',z) <- s, y == y']
```

```
-- -----
-- Ejercicio 19. Comprobar con QuickCheck que clausuraTransitiva es
-- transitiva.
-- -----
```

```
prop_ClausuraTransitiva :: Rel Int -> Bool
prop_ClausuraTransitiva r =
  transitiva (clausuraTransitiva r)
```

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