Piensa en Haskell

(Ejercicios de programación funcional con Haskell)

José A. Alonso Jiménez
Mª José Hidalgo Doblado

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Sevilla, 10 de Julio de 2012 (Versión de 11 de octubre de 2012)

A mi amigo Felipe

Índice general

I Introducción a la programación funcional **19**

1 Definiciones elementales de funciones **21**

1.1 Media de 3 números . . . 22
1.2 Suma de euros de una colección de monedas . . . 23
1.3 Volumen de la esfera . . . 23
1.4 Área de una corona circular . . . 23
1.5 Última cifra de un número . . . 24
1.6 Máximo de 3 elementos . . . 24
1.7 Disyunción excluyente . . . 24
1.8 Rotación de listas . . . 25
1.9 Rango de una lista . . . 26
1.10 Reconocimiento de palíndromos . . . 26
1.11 Elementos interiores de una lista . . . 26
1.12 Finales de una lista . . . 27
1.13 Segmentos de una lista . . . 27
1.14 Extremos de una lista . . . 27
1.15 Mediano de 3 números . . . 27
1.16 Igualdad y diferencia de 3 elementos . . . 28
1.17 Igualdad de 4 elementos . . . 29
1.18 Propiedad triangular . . . 29
1.19 División segura . . . 29
1.20 Disyunción excluyente . . . 30
1.21 Módulo de un vector . . . 30
1.22 Rectángulo de área máxima . . . 31
1.23 Puntos del plano . . . 31
1.23.1 Cuadrante de un punto . . . 31
1.23.2 Intercambio de coordenadas . . . 31

1.23.3 Punto simétrico . . . 32
1.23.4 Distancia entre dos puntos . . . 32
1.23.5 Punto medio entre otros dos . . . 32
1.24 Números complejos . . . 33
1.24.1 Suma de dos números complejos . . . 33
1.24.2 Producto de dos números complejos . . . 33
1.24.3 Conjugado de un número complejo . . . 33
1.25 Intercalación de pares . . . 34
1.26 Permutación cíclica de una lista . . . 34
1.27 Mayor número de 2 cifras con dos dígitos dados . . . 34
1.28 Número de raíces de una ecuación cuadrática . . . 35
1.29 Raíces de las ecuaciones cuadráticas . . . 35
1.30 Área de un triángulo mediante la fórmula de Herón . . . 36
1.31 Números racionales como pares de enteros . . . 36
1.31.1 Forma reducida de un número racional . . . 36
1.31.2 Suma de dos números racionales . . . 36
1.31.3 Producto de dos números racionales . . . 37
1.31.4 Igualdad de números racionales . . . 37

2 Definiciones por comprensión 39
2.1 Suma de los cuadrados de los n primeros números . . . 40
2.2 Listas con un elemento replicado . . . 40
2.3 Triángulos aritméticos . . . 40
2.4 Números perfectos . . . 41
2.5 Números abundantes . . . 42
2.6 Problema 1 del proyecto Euler . . . 43
2.7 Número de pares de naturales en un círculo . . . 44
2.8 Aproximación del número e . . . 44
2.9 Aproximación del límite . . . 46
2.10 Cálculo del número π . . . 46
2.11 Ternas pitagóricas . . . 47
2.12 Problema 9 del Proyecto Euler . . . 48
2.13 Producto escalar . . . 49
2.14 Suma de pares de elementos consecutivos . . . 50
2.15 Posiciones de un elemento en una lista . . . 50
2.16 Representación densa de un polinomio representado dispersamente . . . 51
2.17 Producto cartesiano . . . 51

2.18 Consulta de bases de datos 52

3 Definiciones por recursión 55
3.1 Potencia de exponente natural 56
3.2 Replicación de un elemento 56
3.3 Doble factorial 57
3.4 Algoritmo de Euclides del máximo común divisor 57
3.5 Menor número divisible por una sucesión de números 58
3.6 Número de pasos para resolver el problema de las torres de Hanoi 59
3.7 Conjunción de una lista 59
3.8 Pertenencia a una lista 60
3.9 Último elemento de una lista 60
3.10 Concatenación de una lista 61
3.11 Selección de un elemento 61
3.12 Selección de los primeros elementos 61
3.13 Intercalación de la media aritmética 62
3.14 Ordenación por mezcla 62
3.14.1 Mezcla de listas ordenadas 62
3.14.2 Mitades de una lista 63
3.14.3 Ordenación por mezcla 63
3.14.4 La ordenación por mezcla da listas ordenadas 63
3.14.5 La ordenación por mezcla da una permutación 64
3.14.6 Determinación de permutaciones 65

4 Definiciones por recursión y por comprensión 67
4.1 Suma de los cuadrados de los primeros números 68
4.2 Número de bloques de escaleras triangulares 70
4.3 Suma de los cuadrados de los impares entre los primeros números 71
4.4 Operaciones con los dígitos de los números 72
4.4.1 Lista de los dígitos de un número 72
4.4.2 Suma de los dígitos de un número 73
4.4.3 Decidir si es un dígito del número 74
4.4.4 Número de dígitos de un número 74
4.4.5 Número correspondiente a una lista de dígitos 75
4.4.6 Concatenación de dos números 75
4.4.7 Primer dígito de un número 76
4.4.8 Último dígito de un número 77
4.4.9 Número con los dígitos invertidos 78

4.4.10 Decidir si un número es capicúa . . . 79
4.4.11 Suma de los dígitos de 2^{1000} . . . 79
4.4.12 Primitivo de un número . . . 80
4.4.13 Números con igual media de sus dígitos . . . 80
4.4.14 Números con dígitos duplicados en su cuadrado . . . 81
4.5 Cuadrados de los elementos de una lista . . . 82
4.6 Números impares de una lista . . . 83
4.7 Cuadrados de los elementos impares . . . 84
4.8 Suma de los cuadrados de los elementos impares . . . 85
4.9 Intervalo numérico . . . 86
4.10 Mitades de los pares . . . 87
4.11 Pertenencia a un rango . . . 88
4.12 Suma de elementos positivos . . . 89
4.13 Aproximación del número π . . . 90
4.14 Sustitución de impares por el siguiente par . . . 90
4.15 La compra de una persona agarrada . . . 91
4.16 Descomposición en productos de factores primos . . . 92
4.16.1 Lista de los factores primos de un número . . . 92
4.16.2 Decidir si un número es primo . . . 93
4.16.3 Factorización de un número . . . 93
4.16.4 Exponente de la mayor potencia de un número que divide a otro . . . 93
4.16.5 Expansion de la factorización de un número . . . 94
4.17 Menor número con todos los dígitos en la factorización de su factorial . . . 95
4.18 Suma de números especiales . . . 98
4.19 Distancia de Hamming . . . 99
4.20 Traspuesta de una matriz . . . 100
4.21 Números expresables como sumas acotadas de elementos de una lista . . . 101

5 Funciones sobre cadenas 103

5.1 Suma de los dígitos de una cadena . . . 103
5.2 Capitalización de una cadena . . . 105
5.3 Título con las reglas de mayúsculas iniciales . . . 106
5.4 Búsqueda en crucigramas . . . 107
5.5 Posiciones de un carácter en una cadena . . . 108
5.6 Decidir si una cadena es subcadena de otra . . . 109
5.7 Codificación de mensajes . . . 111
5.8 Números de ceros finales . . . 115

6 Funciones de orden superior **117**
6.1 Segmento inicial verificando una propiedad . . . 118
6.2 Complementario del segmento inicial verificando una propiedad . . . 118
6.3 Concatenación de una lista de listas . . . 119
6.4 División de una lista numérica según su media . . . 119
6.5 Segmentos cuyos elementos verifican una propiedad . . . 122
6.6 Listas con elementos consecutivos relacionados . . . 122
6.7 Agrupamiento de elementos de una lista de listas . . . 123
6.8 Números con dígitos pares . . . 123
6.9 Lista de los valores de los elementos que cumplen una propiedad . . . 125
6.10 Máximo elemento de una lista . . . 126
6.11 Mínimo elemento de una lista . . . 127
6.12 Inversa de una lista . . . 127
6.13 Número correspondiente a la lista de sus cifras . . . 130
6.14 Suma de valores de una aplicación a una lista . . . 131
6.15 Redefinición de la función `map` usando `foldr` . . . 132
6.16 Redefinición de la función `filter` usando `foldr` . . . 132
6.17 Suma de las sumas de las listas de una lista de listas . . . 133
6.18 Lista obtenida borrando las ocurrencias de un elemento . . . 134
6.19 Diferencia de dos listas . . . 135
6.20 Producto de los números que verifican una propiedad . . . 136
6.21 Las cabezas y las colas de una lista . . . 137

7 Listas infinitas **141**
7.1 Lista obtenida repitiendo un elemento . . . 142
7.2 Lista obtenida repitiendo cada elemento según su posición . . . 144
7.3 Potencias de un número menores que otro dado . . . 144
7.4 Múltiplos cuyos dígitos verifican una propiedad . . . 145
7.5 Aplicación iterada de una función a un elemento . . . 145
7.6 Agrupamiento de elementos consecutivos . . . 146
7.7 La sucesión de Collatz . . . 148
7.8 Números primos . . . 150
7.9 Descomposiciones como suma de dos primos . . . 151
7.10 Números expresables como producto de dos primos . . . 152
7.11 Números muy compuestos . . . 152

7.12 Suma de números primos truncables . . . 154
7.13 Primos permutables . . . 155
7.14 Ordenación de los números enteros . . . 155
7.15 La sucesión de Hamming . . . 157
7.16 Suma de los primos menores que n . . . 160
7.17 Menor número triangular con más de n divisores . . . 161
7.18 Números primos consecutivos con dígitos con igual media . . . 162
7.19 Decisión de pertenencia al rango de una función creciente . . . 163
7.20 Pares ordenados por posición . . . 163
7.21 Aplicación iterada de una función . . . 165
7.22 Expresión de un número como suma de dos de una lista . . . 165
7.23 La bicicleta de Turing . . . 166
7.24 Sucesión de Golomb . . . 167

8 Tipos definidos y de datos algebraicos 171
8.1 Puntos cercanos . . . 172
8.2 TDA de los números naturales . . . 173
8.2.1 Suma de números naturales . . . 173
8.2.2 Producto de números naturales . . . 173
8.3 TDA de árboles binarios con valores en los nodos y en las hojas . . . 174
8.3.1 Ocurrencia de un elemento en el árbol . . . 174
8.4 TDA de árboles binarios con valores en las hojas . . . 175
8.4.1 Número de hojas . . . 176
8.4.2 Carácter balanceado de un árbol . . . 176
8.4.3 Árbol balanceado correspondiente a una lista . . . 177
8.5 TDA de árboles binarios con valores en los nodos . . . 177
8.5.1 Número de hojas de un árbol . . . 178
8.5.2 Número de nodos de un árbol . . . 178
8.5.3 Profundidad de un árbol . . . 179
8.5.4 Recorrido preorden de un árbol . . . 179
8.5.5 Recorrido postorden de un árbol . . . 180
8.5.6 Recorrido preorden de forma iterativa . . . 180
8.5.7 Imagen especular de un árbol . . . 181
8.5.8 Subárbol de profundidad dada . . . 181
8.5.9 Árbol infinito generado con un elemento . . . 182
8.5.10 Árbol de profundidad dada cuyos nodos son iguales a un elemento 183
8.5.11 Rama izquierda de un árbol . . . 183
8.6 TAD de fórmulas proposicionales . . . 184

8.7 Modelización de un juego de cartas . . . 189
8.8 Evaluación de expresiones aritméticas . . . 196
8.9 Número de variables de una expresión aritmética . . . 197
8.10 Sustituciones en expresiones aritméticas . . . 198

9 Demostración de propiedades por inducción 199
9.1 Suma de los primeros números impares . . . 199
9.2 Uno más la suma de potencias de dos . . . 201
9.3 Copias de un elemento . . . 203

II Tipos abstractos de datos y algorítmica 207

10 Polinomios 209
10.1 El TAD de los polinomios . . . 210
10.1.1 Especificación del TAD de los polinomios . . . 210
10.1.2 Los polinomios como tipo de dato algebraico . . . 211
10.1.3 Los polinomios como listas dispersas . . . 214
10.1.4 Los polinomios como listas densas . . . 217
10.1.5 Comprobación de las implementaciones con QuickCheck . . . 220
10.2 Operaciones con polinomios . . . 223
10.2.1 Funciones sobre términos . . . 224
10.2.2 Suma de polinomios . . . 225
10.2.3 Producto de polinomios . . . 226
10.2.4 El polinomio unidad . . . 227
10.2.5 Resta de polinomios . . . 228
10.2.6 Valor de un polinomio en un punto . . . 228
10.2.7 Verificación de raices de polinomios . . . 228
10.2.8 Derivación de polinomios . . . 229
10.3 Ejercicios sobre polinomios . . . 229
10.3.1 Polinomio a partir de la representación dispersa . . . 230
10.3.2 Polinomio a partir de la representación densa . . . 230
10.3.3 Representación densa de un polinomio . . . 231
10.3.4 Transformación de la representación densa a dispersa . . . 231
10.3.5 Representación dispersa de un polinomio . . . 231
10.3.6 Coeficiente del término de grado k . . . 232
10.3.7 Lista de los coeficientes de un polinomio . . . 232
10.3.8 Potencia de un polinomio . . . 233
10.3.9 Integración de polinomios . . . 234
10.3.10 Multiplicación de un polinomio por un número . . . 235

10.3.11 División de polinomios . . . 235
10.3.12 Divisibilidad de polinomios . . . 236
10.4 La regla de Ruffini . . . 238
10.4.1 Divisores de un número . . . 238
10.4.2 Término independiente de un polinomio . . . 238
10.4.3 Paso de la regla de Ruffini . . . 239
10.4.4 Cociente mediante la regla de Ruffini . . . 239
10.4.5 Resto mediante la regla de Ruffini . . . 240
10.4.6 Raíces mediante la regla de Ruffini . . . 240
10.4.7 Factorización mediante la regla de Ruffini . . . 241

11 Vectores y matrices **243**
11.1 Posiciones de un elemento en una matriz . . . 244
11.2 Tipos de los vectores y de las matrices . . . 244
11.3 Operaciones básicas con matrices . . . 245
11.4 Suma de matrices . . . 247
11.5 Producto de matrices . . . 249
11.6 Traspuestas y simétricas . . . 250
11.7 Diagonales de una matriz . . . 251
11.8 Submatrices . . . 252
11.9 Transformaciones elementales . . . 252
11.10 Triangularización de matrices . . . 255
11.11 Algoritmo de Gauss para triangularizar matrices . . . 257
11.12 Determinante . . . 260
11.13 Máximo de las sumas de elementos de una matriz en líneas distintas . . . 261

12 Relaciones binarias homogéneas **265**
12.1 Tipo de dato de las relaciones binarias . . . 266
12.2 Universo de una relación . . . 266
12.3 Grafo de una relación . . . 267
12.4 Relaciones reflexivas . . . 267
12.5 Relaciones simétricas . . . 267
12.6 Reconocimiento de subconjuntos . . . 268
12.7 Composición de relaciones . . . 268
12.8 Relación transitiva . . . 268
12.9 Relación de equivalencia . . . 269
12.10 Relación irreflexiva . . . 269

12.11 Relación antisimétrica . . . 269
12.12 Relación total . . . 270
12.13 Clausura reflexiva . . . 271
12.14 Clausura simétrica . . . 272
12.15 Clausura transitiva . . . 273

13 Operaciones con conjuntos 275
13.1 Representación de conjuntos y operaciones básicas . . . 276
13.1.1 El tipo de los conjuntos . . . 276
13.1.2 El conjunto vacío . . . 277
13.1.3 Reconocimiento del conjunto vacío . . . 277
13.1.4 Pertenencia de un elemento a un conjunto . . . 277
13.1.5 Inserción de un elemento en un conjunto . . . 278
13.1.6 Eliminación de un elemento de un conjunto . . . 278
13.2 Ejercicios sobre conjuntos . . . 278
13.2.1 Reconocimiento de subconjuntos . . . 278
13.2.2 Reconocimiento de subconjunto propio . . . 280
13.2.3 Conjunto unitario . . . 281
13.2.4 Cardinal de un conjunto . . . 281
13.2.5 Unión de conjuntos . . . 281
13.2.6 Unión de una lista de conjuntos . . . 282
13.2.7 Intersección de conjuntos . . . 283
13.2.8 Intersección de una lista de conjuntos . . . 284
13.2.9 Conjuntos disjuntos . . . 284
13.2.10 Diferencia de conjuntos . . . 285
13.2.11 Diferencia simétrica de conjuntos . . . 285
13.2.12 Filtrado en conjuntos . . . 285
13.2.13 Partición de un conjunto según una propiedad . . . 286
13.2.14 División de un conjunto según un elemento . . . 286
13.2.15 Aplicación de una función a un conjunto . . . 286
13.2.16 Todos los elementos verifican una propiedad . . . 287
13.2.17 Algunos elementos verifican una propiedad . . . 287
13.2.18 Producto cartesiano . . . 287
13.2.19 Orden en el tipo de los conjuntos . . . 288
13.2.20 Conjunto potencia . . . 288
13.2.21 Verificación de propiedades de conjuntos . . . 288

14 Grafos 295
14.1 El TAD de los grafos . . . 296
14.1.1 Especificación del TAD de los grafos . . . 296

14.1.2 Los grafos como vectores de adyacencia . . . 297
14.1.3 Los grafos como matrices de adyacencia . . . 300
14.1.4 Los grafos como listas . . . 304
14.2 Ejercicios sobre grafos . . . 309
14.2.1 Generador de grafos . . . 310
14.2.2 El grafo completo de orden n . . . 312
14.2.3 El ciclo de orden n . . . 312
14.2.4 Número de vértices . . . 313
14.2.5 Reconocimiento de grafos no dirigidos . . . 313
14.2.6 Vértices incidentes . . . 314
14.2.7 Vértices contiguos . . . 314
14.2.8 Lazos . . . 314
14.2.9 Número de lazos . . . 315
14.2.10 Número de aristas . . . 315
14.2.11 Grado positivo de un vértice . . . 316
14.2.12 Grado negativo de un vértice . . . 317
14.2.13 Grado de un vértice . . . 317
14.2.14 Grafos regulares . . . 319
14.2.15 Grafos k–regulares . . . 320

III Casos de estudio 323

15 El cifrado César 325
15.1 Codificación y descodificación . . . 325
15.2 Análisis de frecuencias . . . 328
15.3 Descifrado . . . 329

16 Codificación y transmisión de mensajes 331
16.1 Cambios de bases . . . 331
16.2 Codificación . . . 333
16.3 Descodificación . . . 335
16.4 Transmisión . . . 336

17 Resolución de problemas matemáticos 337
17.1 El problema de Ullman sobre la existencia de subconjunto del tamaño dado y con su suma acotada . . . 338
17.2 Descomposiciones de un número como suma de dos cuadrados . . . 339
17.3 Números reversibles . . . 340
17.4 Grafo de una función sobre los elementos que cumplen una propiedad . . . 341

17.5 Números semiperfectos . . . 342
17.6 Decidir el carácter funcional de una relación . . . 344
17.7 La identidad de Bézout . . . 344
17.8 Distancia entre dos conjuntos de números . . . 346
17.9 Expresables como suma de números consecutivos . . . 346
17.10 Solución de una ecuación diofántica . . . 348

18 El 2011 y los números primos **351**
18.1 La criba de Eratótenes . . . 351
18.2 2011 es primo . . . 353
18.3 Primera propiedad del 2011 . . . 353
18.4 Segunda propiedad del 2011 . . . 355
18.5 Tercera propiedad del 2011 . . . 357

19 Combinatoria **359**
19.1 Reconocimiento y generación de subconjuntos . . . 359
19.2 Permutaciones . . . 361
19.3 Combinaciones sin repetición . . . 364
19.4 Combinaciones con repetición . . . 367
19.5 Variaciones sin repetición . . . 369
19.6 Variaciones con repetición . . . 370
19.7 El triángulo de Pascal . . . 372

20 Cálculo numérico **375**
20.1 Diferenciación numérica . . . 375
20.2 Cálculo de la raíz cuadrada mediante el método de Herón . . . 377
20.3 Cálculo de los ceros de una función por el método de Newton . . . 379
20.4 Cálculo de funciones inversas . . . 380

21 Ecuación con factoriales **385**
21.1 Cálculo de factoriales . . . 385
21.2 Decisión de si un número es un factorial . . . 386
21.3 Inversa del factorial . . . 386
21.4 Enumeración de los pares de números naturales . . . 387
21.5 Solución de la ecuación con factoriales . . . 388

22 Cuadrados mágicos **391**
22.1 Reconocimiento de los cuadrados mágicos . . . 392

22.1.1 Traspuesta de una matriz . . . 392
22.1.2 Suma de las filas de una matriz . . . 393
22.1.3 Suma de las columnas de una matriz . . . 393
22.1.4 Diagonal principal de una matriz . . . 393
22.1.5 Diagonal secundaria de una matriz . . . 394
22.1.6 Lista con todos los elementos iguales . . . 394
22.1.7 Reconocimiento de matrices cuadradas . . . 394
22.1.8 Elementos de una lista de listas . . . 395
22.1.9 Eliminación de la primera ocurrencia de un elemento . . . 395
22.1.10 Reconocimiento de permutaciones . . . 395
22.1.11 Reconocimiento de cuadrados mágicos . . . 396
22.2 Cálculo de los cuadrados mágicos . . . 396
22.2.1 Matriz cuadrada correspondiente a una lista de elementos . . . 396
22.2.2 Cálculo de cuadrados mágicos por permutaciones . . . 397
22.2.3 Cálculo de los cuadradros mágicos mediante generación y poda . 397

23 Enumeraciones de los números racionales 401
23.1 Numeración de los racionales mediante representaciones hiperbinarias . 402
23.1.1 Lista de potencias de dos . . . 402
23.1.2 Determinación si los dos primeros elementos son iguales a uno dado . . . 402
23.1.3 Lista de las representaciones hiperbinarias de n . . . 403
23.1.4 Número de representaciones hiperbinarias de n . . . 403
23.1.5 Sucesiones hiperbinarias . . . 404
23.2 Numeraciones mediante árboles de Calkin–Wilf . . . 406
23.2.1 Hijos de un nodo en el árbol de Calvin–Wilf . . . 406
23.2.2 Niveles del árbol de Calvin–Wilf . . . 407
23.2.3 Sucesión de Calvin–Wilf . . . 408
23.3 Número de representaciones hiperbinarias mediante la función `fusc` . . 408
23.3.1 La función `fusc` . . . 408

IV Apéndices 411

A Resumen de funciones predefinidas de Haskell 413

B Método de Pólya para la resolución de problemas 417
B.1 Método de Pólya para la resolución de problemas matemáticos . . . 417
B.2 Método de Pólya para resolver problemas de programación . . . 418

Bibliografía 421

Indice de definiciones 421

Este libro es una introducción a la programación funcional con Haskell a través de ejercicios que se complementa con los *Temas de programación funcional*[1].

El libro consta de tres partes. En la primera parte se presentan los elementos básicos de la programación funcional. En la segunda, se estudian la implementación en Haskell de tipos abstractos de datos y sus aplicaciones así como cuestiones algorítmicas. En la tercera, se presentan casos de estudios. También se han incluido dos apéndices: uno con un resumen de las funciones de Haskell utilizadas y otro con el método de Pólya para la resolución de problemas.

Estos ejercicios se han utilizado en los cursos de "Informática (del Grado en Matemáticas)"[2] y "Programación declarativa (de la Ingeniería en Informática)"[3].

[1] http://www.cs.us.es/~jalonso/cursos/i1m/temas/2011-12-IM-temas-PF.pdf
[2] http://www.cs.us.es/~jalonso/cursos/i1m-11
[3] http://www.cs.us.es/~jalonso/cursos/pd-09

Parte I

Introducción a la programación funcional

Capítulo 1

Definiciones elementales de funciones

En este capítulo se plantean ejercicios con definiciones elementales (no recursivas) de funciones. Se corresponden con los 4 primeros temas de [1].

Contenido

1.1 Media de 3 números . . . 22
1.2 Suma de euros de una colección de monedas . . . 23
1.3 Volumen de la esfera . . . 23
1.4 Área de una corona circular . . . 23
1.5 Última cifra de un número . . . 24
1.6 Máximo de 3 elementos . . . 24
1.7 Disyunción excluyente . . . 24
1.8 Rotación de listas . . . 25
1.9 Rango de una lista . . . 26
1.10 Reconocimiento de palíndromos . . . 26
1.11 Elementos interiores de una lista . . . 26
1.12 Finales de una lista . . . 27
1.13 Segmentos de una lista . . . 27
1.14 Extremos de una lista . . . 27
1.15 Mediano de 3 números . . . 27
1.16 Igualdad y diferencia de 3 elementos . . . 28
1.17 Igualdad de 4 elementos . . . 29
1.18 Propiedad triangular . . . 29

1.19 División segura 29
1.20 Disyunción excluyente 30
1.21 Módulo de un vector 30
1.22 Rectángulo de área máxima 31
1.23 Puntos del plano 31
1.23.1 Cuadrante de un punto 31
1.23.2 Intercambio de coordenadas 31
1.23.3 Punto simétrico 32
1.23.4 Distancia entre dos puntos 32
1.23.5 Punto medio entre otros dos 32
1.24 Números complejos 33
1.24.1 Suma de dos números complejos 33
1.24.2 Producto de dos números complejos 33
1.24.3 Conjugado de un número complejo 33
1.25 Intercalación de pares 34
1.26 Permutación cíclica de una lista 34
1.27 Mayor número de 2 cifras con dos dígitos dados 34
1.28 Número de raíces de una ecuación cuadrática 35
1.29 Raíces de las ecuaciones cuadráticas 35
1.30 Área de un triángulo mediante la fórmula de Herón 36
1.31 Números racionales como pares de enteros 36
1.31.1 Forma reducida de un número racional 36
1.31.2 Suma de dos números racionales 36
1.31.3 Producto de dos números racionales 37
1.31.4 Igualdad de números racionales 37

1.1. Media de 3 números

Ejercicio 1.1.1. *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
```

Solución:

```
media3 x y z = (x+y+z)/3
```

1.2. Suma de euros de una colección de monedas

Ejercicio 1.2.1. *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
```

Solución:

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

1.3. Volumen de la esfera

Ejercicio 1.3.1. *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`*.*

Solución:

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

1.4. Área de una corona circular

Ejercicio 1.4.1. *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
```

Solución:

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

1.5. Última cifra de un número

Ejercicio 1.5.1. *Definir la función* `ultimaCifra` *tal que* `(ultimaCifra x)` *es la última cifra del número* `x`*. Por ejemplo,*

```
ultimaCifra 325  ==  5
```

Solución:

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

1.6. Máximo de 3 elementos

Ejercicio 1.6.1. *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
```

Solución:

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

1.7. Disyunción excluyente

La disyunción excluyente `xor` de dos fórmulas se verifica si una es verdadera y la otra es falsa.

Ejercicio 1.7.1. *Definir la función* `xor1` *que calcule la disyunción excluyente a partir de la tabla de verdad. Usar 4 ecuaciones, una por cada línea de la tabla.*

Solución:

```
xor1 True  True  = False
xor1 True  False = True
xor1 False True  = True
xor1 False False = False
```

Ejercicio 1.7.2. *Definir la función* `xor2` *que calcule la disyunción excluyente a partir de la tabla de verdad y patrones. Usar 2 ecuaciones, una por cada valor del primer argumento.*

Solución:

```
xor2 True  y = not y
xor2 False y = y
```

Ejercicio 1.7.3. *Definir la función* `xor3` *que calcule la disyunción excluyente a partir de la disyunción (*`||`*), conjunción (*`&&`*) y negación (*`not`*). Usar 1 ecuación.*

Solución:

```
xor3 x y = (x || y) && not (x && y)
```

Ejercicio 1.7.4. *Definir la función* `xor4` *que calcule la disyunción excluyente a partir de desigualdad (*`/=`*). Usar 1 ecuación.*

Solución:

```
xor4 x y = x /= y
```

1.8. Rotación de listas

Ejercicio 1.8.1. *Definir la función* `rota1` *tal que* `(rota1 xs)` *es la lista obtenida poniendo el primer elemento de* `xs` *al final de la lista. Por ejemplo,*

```
rota1 [3,2,5,7]  ==  [2,5,7,3]
```

Solución:

```
rota1 xs = tail xs ++ [head xs]
```

Ejercicio 1.8.2. *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]
```

Solución:

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

1.9. Rango de una lista

Ejercicio 1.9.1. *Definir la función* `rango` *tal que* `(rango xs)` *es la lista formada por el menor y mayor elemento de* `xs`*. Por ejemplo,*

```
rango [3,2,7,5]  ==  [2,7]
```

Indicación: *Se pueden usar* `minimum` *y* `maximum`*.*

Solución:

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

1.10. Reconocimiento de palíndromos

Ejercicio 1.10.1. *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
```

Solución:

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

1.11. Elementos interiores de una lista

Ejercicio 1.11.1. *Definir la función* `interior` *tal que* `(interior xs)` *es la lista obtenida eliminando los extremos de la lista* `xs`*. Por ejemplo,*

```
interior [2,5,3,7,3]  ==  [5,3,7]
interior [2..7]       ==  [3,4,5,6]
```

Solución:

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

1.12. Finales de una lista

Ejercicio 1.12.1. *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]
```

Solución:

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

1.13. Segmentos de una lista

Ejercicio 1.13.1. *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]  ==  []
```

Solución:

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

1.14. Extremos de una lista

Ejercicio 1.14.1. *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]
```

Solución:

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

1.15. Mediano de 3 números

Ejercicio 1.15.1. *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
```

Solución: Se presentan dos soluciones. La primera es

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

La segunda es

```
mediano' x y z
    | a <= x && x <= b = x
    | a <= y && y <= b = y
    | otherwise        = z
    where a = minimum [x,y,z]
          b = maximum [x,y,z]
```

1.16. Igualdad y diferencia de 3 elementos

Ejercicio 1.16.1. *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
```

Solución:

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

Ejercicio 1.16.2. *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
```

Solución:

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

1.17. Igualdad de 4 elementos

Ejercicio 1.17.1. *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`.

Solución:

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

1.18. Propiedad triangular

Ejercicio 1.18.1. *Las longitudes de los lados de un triángulo no pueden ser cualesquiera. Para que pueda construirse el triángulo, tiene que cumplirse la propiedad triangular; es decir, longitud de cada lado tiene que ser menor que la suma de los otros dos lados.*

Definir la función `triangular` *tal que* `(triangular a b c)` *se verifica si* `a`, `b` *y* `c` *complen la propiedad triangular. Por ejemplo,*

```
triangular  3  4  5   ==  True
triangular 30  4  5  ==  False
triangular  3 40  5  ==  False
triangular  3  4 50  ==  False
```

Solución:

```
triangular a b c = a < b+c && b < a+c && c < a+b
```

1.19. División segura

Ejercicio 1.19.1. *Definir la función* `divisionSegura` *tal que* `(divisionSegura x y)` *es* $\frac{x}{y}$ *si y no es cero y 9999 en caso contrario. Por ejemplo,*

```
divisionSegura 7 2  ==  3.5
divisionSegura 7 0  ==  9999.0
```

Solución:

```
divisionSegura _ 0 = 9999
divisionSegura x y = x/y
```

1.20. Disyunción excluyente

La disyunción excluyente `xor` de dos fórmulas se verifica si una es verdadera y la otra es falsa.

Ejercicio 1.20.1. *Definir la función* `xor1` *que calcule la disyunción excluyente a partir de la tabla de verdad. Usar 4 ecuaciones, una por cada línea de la tabla.*

Solución:

```
xor1 True  True  = False
xor1 True  False = True
xor1 False True  = True
xor1 False False = False
```

Ejercicio 1.20.2. *Definir la función* `xor2` *que calcule la disyunción excluyente a partir de la tabla de verdad y patrones. Usar 2 ecuaciones, una por cada valor del primer argumento.*

Solución:

```
xor2 True  y = not y
xor2 False y = y
```

Ejercicio 1.20.3. *Definir la función* `xor3` *que calcule la disyunción excluyente a partir de la disyunción (*`||`*), conjunción (*`&&`*) y negación (*`not`*). Usar 1 ecuación.*

Solución:

```
xor3 x y = (x || y) && not (x && y)
```

Ejercicio 1.20.4. *Definir la función* `xor4` *que calcule la disyunción excluyente a partir de desigualdad (*`/=`*). Usar 1 ecuación.*

Solución:

```
xor4 x y = x /= y
```

1.21. Módulo de un vector

Ejercicio 1.21.1. *Definir la función* `modulo` *tal que* `(modulo v)` *es el módulo del vector* `v`*. Por ejemplo,*

```
modulo (3,4)  ==  5.0
```

Solución:

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

1.22. Rectángulo de área máxima

Ejercicio 1.22.1. *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 entre* `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)
```

Solución:

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

1.23. Puntos del plano

Los puntos del plano se puede representar por un par de números que son sus coordenadas.

1.23.1. Cuadrante de un punto

Ejercicio 1.23.1. *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
```

Solución:

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

1.23.2. Intercambio de coordenadas

Ejercicio 1.23.2. *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)
```

Solución:

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

1.23.3. Punto simétrico

Ejercicio 1.23.3. *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)
```

Solución:

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

1.23.4. Distancia entre dos puntos

Ejercicio 1.23.4. *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
```

Solución:

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

1.23.5. Punto medio entre otros dos

Ejercicio 1.23.5. *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)
```

Solución:

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

1.24. Números complejos

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

1.24.1. Suma de dos números complejos

Ejercicio 1.24.1. *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)
```

Solución:

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

1.24.2. Producto de dos números complejos

Ejercicio 1.24.2. *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)
```

Solución:

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

1.24.3. Conjugado de un número complejo

Ejercicio 1.24.3. *Definir la función* `conjugado` *tal que* `(conjugado z)` *es el conjugado del número complejo* `z`*. Por ejemplo,*

```
conjugado (2,3) ==  (2,-3)
```

Solución:

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

1.25. Intercalación de pares

Ejercicio 1.25.1. *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]
```

Solución:

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

1.26. Permutación cíclica de una lista

Ejercicio 1.26.1. *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]
```

Solución:

```
ciclo [] = []
ciclo xs = last xs : init xs
```

1.27. Mayor número de 2 cifras con dos dígitos dados

Ejercicio 1.27.1. *Definir la funcion* `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
```

Solución:

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

1.28. Número de raíces de una ecuación cuadrática

Ejercicio 1.28.1. *Definir la función* `numeroDeRaices` *tal que* `(numeroDeRaices a b c)` *es el número de raíces reales de la ecuación* $ax^2 + bx + c = 0$. *Por ejemplo,*

```
numeroDeRaices 2 0 3    ==  0
numeroDeRaices 4 4 1    ==  1
numeroDeRaices 5 23 12  ==  2
```

Solución:

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

1.29. Raíces de las ecuaciones cuadráticas

Ejercicio 1.29.1. *Definir la función* `raices` *de forma que* `(raices a b c)` *devuelve la lista de las raices 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]
```

Solución: Se presenta dos soluciones. La primera es

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

La segunda es

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

1.30. Área de un triángulo mediante la fórmula de Herón

Ejercicio 1.30.1. *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* $\sqrt{s(s-a)(s-b)(s-c)}$*, donde s es el semiperímetro* $\left(s = \frac{a+b+c}{2}\right)$.

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
```

Solución:

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

1.31. Números racionales como pares de enteros

Los números racionales pueden representarse mediante pares de números enteros. Por ejemplo, el número $\frac{2}{5}$ puede representarse mediante el par `(2,5)`.

1.31.1. Forma reducida de un número racional

Ejercicio 1.31.1. *Definir la función* `formaReducida` *tal que* `(formaReducida x)` *es la forma reducida del número racional* `x`. *Por ejemplo,*

```
formaReducida (4,10)  ==  (2,5)
```

Solución:

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

1.31.2. Suma de dos números racionales

Ejercicio 1.31.2. *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)
```

Solución:

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

1.31.3. Producto de dos números racionales

Ejercicio 1.31.3. *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)
```

Solución:

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

1.31.4. Igualdad de números racionales

Ejercicio 1.31.4. *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
```

Solución:

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

Capítulo 2

Definiciones por comprensión

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el tema 5 de [1].

Contenido

2.1 Suma de los cuadrados de los n primeros números 40
2.2 Listas con un elemento replicado 40
2.3 Triángulos aritméticos 40
2.4 Números perfectos 41
2.5 Números abundantes 42
2.6 Problema 1 del proyecto Euler 43
2.7 Número de pares de naturales en un círculo 44
2.8 Aproximación del número e 44
2.9 Aproximación del límite 46
2.10 Cálculo del número π 46
2.11 Ternas pitagóricas 47
2.12 Problema 9 del Proyecto Euler 48
2.13 Producto escalar 49
2.14 Suma de pares de elementos consecutivos 50
2.15 Posiciones de un elemento en una lista 50
2.16 Representación densa de un polinomio representado dispersamente . 51
2.17 Producto cartesiano 51
2.18 Consulta de bases de datos 52

2.1. Suma de los cuadrados de los n primeros números

Ejercicio 2.1.1. *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,* $1^2 + 2^2 + \cdots + n^2$. *Por ejemplo,*

```
sumaDeCuadrados 3    ==  14
sumaDeCuadrados 100  ==  338350
```

Solución:

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

2.2. Listas con un elemento replicado

Ejercicio 2.2.1. *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`.

Solución:

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

2.3. Triángulos aritméticos

Ejercicio 2.3.1. *Definir la función* `suma` *tal* `(suma n)` *es la suma de los* `n` *primeros números. Por ejemplo,*

```
suma 3 == 6
```

Solución:

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

Otra definición es

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

Ejercicio 2.3.2. *Los triángulo aritmético 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 `linea` *tal que* `(linea n)` *es la línea* `n`*–ésima de los triángulos aritméticos. Por ejemplo,*

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

Solución:

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

Ejercicio 2.3.3. *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]]
```

Solución:

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

2.4. Números perfectos

Ejercicio 2.4.1. *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]
```

Solución:

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

donde `(factores n)` es la lista de los factores de `n`

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

2.5. Números abundantes

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 2.5.1. *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
```

Solución:

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

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

Ejercicio 2.5.2. *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]
```

Solución:

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

Ejercicio 2.5.3. *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
```

Solución:

```
todosPares :: Int -> Bool
todosPares n = and [even x | x <- numerosAbundantesMenores n]
```

Ejercicio 2.5.4. *Definir la constante* `primerAbundanteImpar` *que calcule el primer número natural abundante impar. Determinar el valor de dicho número.*

Solución:

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

Su cálculo es

```
ghci> primerAbundanteImpar
945
```

2.6. Problema 1 del proyecto Euler

Ejercicio 2.6.1. *Definir la función*

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

tal que `(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.

Solución:

```
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
```

El cálculo es

```
ghci> euler1 1000
233168
```

2.7. Número de pares de naturales en un círculo

Ejercicio 2.7.1. *Definir la función*

```
circulo :: Int -> Int
```

tal que `(circulo n)` *es 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
```

Solución:

```
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
```

donde `(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))
```

2.8. Aproximación del número e

Ejercicio 2.8.1. *Definir la función* `aproxE` *tal que* `(aproXE n)` *es la lista cuyos elementos son los términos de la sucesión* $\left(1+\frac{1}{m}\right)^m$ *desde* 1 *hasta* `n`*. Por ejemplo,*

```
aproxE 1 == [2.0]
aproxE 4 == [2.0,2.25,2.37037037037037,2.44140625]
```

Solución:

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

Ejercicio 2.8.2. *¿Cuál es el límite de la sucesión* $\left(1+\frac{1}{m}\right)^m$*?*

Solución: El límite de la sucesión es el número e.

Ejercicio 2.8.3. *Definir la función* `errorE` *tal que* `(errorE x)` *es el menor número de términos de la sucesión* $\left(1+\frac{1}{m}\right)^m$ *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)`*.*

Solución:

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

Ejercicio 2.8.4. *El número e también se puede definir como la suma de la serie:*

$$\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\cdots$$

Definir la función `aproxE'` *tal que* `(aproxE' n)` *es la aproximación de e que se obtiene sumando los términos de la serie hasta* $\frac{1}{n!}$*. Por ejemplo,*

```
aproxE' 10  ==  2.718281801146385
aproxE' 100  ==  2.7182818284590455
```

Solución:

```
aproxE' n = 1 + sum [ 1 / factorial k | k <- [1..n]]

factorial n = product [1..n]
```

Ejercicio 2.8.5. *Definir la constante e como* 2,71828459*.*

Solución:

```
e = 2.71828459
```

Ejercicio 2.8.6. *Definir la función* `errorE'` *tal que* `(errorE' x)` *es el menor número de términos de la serie anterior necesarios para obtener e con un error menor que* `x`*. Por ejemplo,*

```
errorE' 0.1     ==  3.0
errorE' 0.01    ==  4.0
errorE' 0.001   ==  6.0
errorE' 0.0001  ==  7.0
```

Solución:

```
errorE' x = head [n | n <- [0..], abs(aproxE' n - e) < x]
```

2.9. Aproximación del límite

Ejercicio 2.9.1. *Definir la función* `aproxLimSeno` *tal que* `(aproxLimSeno n)` *es la lista cuyos elementos son los términos de la sucesión* $\frac{sen(\frac{1}{m})}{\frac{1}{m}}$ *desde* 1 *hasta* `n`*. Por ejemplo,*

```
aproxLimSeno 1 == [0.8414709848078965]
aproxLimSeno 2 == [0.8414709848078965,0.958851077208406]
```

Solución:

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

Ejercicio 2.9.2. *¿Cuál es el límite de la sucesión* $\frac{sen(\frac{1}{m})}{\frac{1}{m}}$*?*

Solución: El límite es 1.

Ejercicio 2.9.3. *Definir la función* `errorLimSeno` *tal que* `(errorLimSeno x)` *es el menor número de términos de la sucesión* $\frac{sen(\frac{1}{m})}{\frac{1}{m}}$ *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
```

Solución:

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

2.10. Cálculo del número π

Ejercicio 2.10.1. *Definir la función* `calculaPi` *tal que* `(calculaPi n)` *es la aproximación del número* π *calculada mediante la expresión*

$$4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1})$$

Por ejemplo,

```
calculaPi 3    ==  2.8952380952380956
calculaPi 300  ==  3.1449149035588526
```

Solución:

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

Ejercicio 2.10.2. *Definir la función* `errorPi` *tal que* `(errorPi x)` *es el menor número de términos de la serie*

$$4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1})$$

necesarios para obtener π *con un error menor que* `x`*. Por ejemplo,*

```
errorPi 0.1    ==   9.0
errorPi 0.01   ==  99.0
errorPi 0.001  ==  999.0
```

Solución:

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

2.11. Ternas pitagóricas

Ejercicio 2.11.1. *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)]
```

Solución:

```
pitagoricas :: Int -> [(Int,Int,Int)]
pitagoricas n = [(x,y,z) | x <- [1..n],
                           y <- [1..n],
                           z <- [1..n],
                           x^2 + y^2 == z^2]
```

Ejercicio 2.11.2. *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
```

Solución:

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

Ejercicio 2.11.3. *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
```

Solución:

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

Ejercicio 2.11.4. *Demostrar la conjetura para todas las ternas pitagóricas.*

Solución: 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.

2.12. Problema 9 del Proyecto Euler

Ejercicio 2.12.1. *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)]
```

Solución:

```
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 2.12.2. *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`.

Solución:

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

El cálculo del valor de euler9 es

```
ghci> euler9
31875000
```

2.13. Producto escalar

Ejercicio 2.13.1. *El producto escalar de dos listas de enteros* `xs` *e* `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
```

Solución:

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

2.14. Suma de pares de elementos consecutivos

Ejercicio 2.14.1. *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]        ==  []
```

Solución:

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

2.15. Posiciones de un elemento en una lista

Ejercicio 2.15.1. *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`*.*

Solución: 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..])
```

2.16. Representación densa de un polinomio representado dispersamente

Ejercicio 2.16.1. *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)]
```

Solución:

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

2.17. Producto cartesiano

Ejercicio 2.17.1. *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.*

Solución:

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

2.18. Consulta de bases de datos

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 los siguientes ejercicios

```
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 2.18.1. *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"]
```

Solución:

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

Ejercicio 2.18.2. *Definir la función* `musicos` *tal que* `(musicos bd)` *es la lista de los nombres de los músicos de la base de datos* `bd`*. Por ejemplo,*

```
ghci> musicos personas
["Beethoven","Mozart","Bach"]
```

Solución:

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

Ejercicio 2.18.3. *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"]
```

Solución:

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

Ejercicio 2.18.4. *Definir, usando el apartado anterior, la función* `musicos'` *tal que* `(musicos' bd)` *es la lista de los nombres de los músicos de la base de datos* `bd`*. Por ejemplo,*

```
ghci> musicos' personas
["Beethoven","Mozart","Bach"]
```

Solución:

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

Ejercicio 2.18.5. *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"]
```

Solución:

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

Nota. Un caso de estudio para las definiciones por comprensión es el capítulo 15 *"El cifrado César"* (página 325).

Capítulo 3

Definiciones por recursión

En este capítulo se presentan ejercicios con definiciones por recursión. Se corresponden con el tema 6 de [1].

Contenido

3.1 Potencia de exponente natural **56**
3.2 Replicación de un elemento **56**
3.3 Doble factorial **57**
3.4 Algoritmo de Euclides del máximo común divisor **57**
3.5 Menor número divisible por una sucesión de números **58**
3.6 Número de pasos para resolver el problema de las torres de Hanoi . . **59**
3.7 Conjunción de una lista **59**
3.8 Pertenencia a una lista **60**
3.9 Último elemento de una lista **60**
3.10 Concatenación de una lista **61**
3.11 Selección de un elemento **61**
3.12 Selección de los primeros elementos **61**
3.13 Intercalación de la media aritmética **62**
3.14 Ordenación por mezcla **62**
3.14.1 Mezcla de listas ordenadas 62
3.14.2 Mitades de una lista 63
3.14.3 Ordenación por mezcla 63
3.14.4 La ordenación por mezcla da listas ordenadas 63

3.14.5 La ordenación por mezcla da una permutación 64
3.14.6 Determinación de permutaciones 65

Nota. En esta relación se usa la librería de `QuickCheck`.

```
import Test.QuickCheck
```

3.1. Potencia de exponente natural

Ejercicio 3.1.1. *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
```

Solución:

```
potencia :: Integer -> Integer -> Integer
potencia m 0 = 1
potencia m n = m*(potencia m (n-1))
```

3.2. Replicación de un elemento

Ejercicio 3.2.1. *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]
```

Solución:

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

3.3. Doble factorial

Ejercicio 3.3.1. *El doble factorial de un número* `n` *se define por*

```
0!! = 1
1!! = 1
n!! = n*(n-2)* ... * 3 * 1, si n es impar
n!! = n*(n-2)* ... * 4 * 2, si n es par
```

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
```

Solución:

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

3.4. Algoritmo de Euclides del máximo común divisor

Ejercicio 3.4.1. *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:*

$$mcd(a,b) = \begin{cases} a, & si\ b = 0 \\ mcd(b, a\ módulo\ b), & si\ b > 0 \end{cases}$$

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
```

Solución:

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

3.5. Menor número divisible por una sucesión de números

Los siguientes ejercicios tienen como objetivo resolver el problema 5 del proyecto Euler que consiste en calcular el menor número divisible por los números del 1 al 20.

Ejercicio 3.5.1. *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`.

Solución:

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

Ejercicio 3.5.2. *Definir la constante*

```
euler5 :: Integer
```

tal que `euler5` *es el menor número divisible por los números del 1 al 20 y calcular su valor.*

Solución:

```
euler5 :: Integer
euler5 = menorDivisible 1 20
```

El cálculo es

```
ghci> euler5
232792560
```

3.6. Número de pasos para resolver el problema de las torres de Hanoi

Ejercicio 3.6.1. *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
```

Solución: 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) = 1 + 2 * numPasosHanoi n
```

3.7. Conjunción de una lista

Ejercicio 3.7.1. *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
```

Solución:

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

3.8. Pertenencia a una lista

Ejercicio 3.8.1. *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
```

Solución:

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

3.9. Último elemento de una lista

Ejercicio 3.9.1. *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
```

Solución:

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

3.10. Concatenación de una lista

Ejercicio 3.10.1. *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]
```

Solución:

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

3.11. Selección de un elemento

Ejercicio 3.11.1. *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
```

Solución:

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

3.12. Selección de los primeros elementos

Ejercicio 3.12.1. *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]
```

Solución:

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

3.13. Intercalación de la media aritmética

Ejercicio 3.13.1. *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 []         ==  []
```

Solución:

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

3.14. Ordenación por mezcla

3.14.1. Mezcla de listas ordenadas

Ejercicio 3.14.1. *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]
```

Solución:

```
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
```

3.14.2. Mitades de una lista

Ejercicio 3.14.2. *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])
```

Solución:

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

3.14.3. Ordenación por mezcla

Ejercicio 3.14.3. *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]
```

Solución:

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

3.14.4. La ordenación por mezcla da listas ordenadas

Ejercicio 3.14.4. *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
```

Solución:

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

Ejercicio 3.14.5. *Comprobar con QuickCheck que la ordenación por mezcla de una lista es una lista ordenada.*

Solución: La propiedad es

```
prop_ordMezcla_ordenada :: Ord a => [a] -> Bool
prop_ordMezcla_ordenada xs = ordenada (ordMezcla xs)
```

La comprobación es

```
ghci> quickCheck prop_ordMezcla_ordenada
+++ OK, passed 100 tests.
```

3.14.5. La ordenación por mezcla da una permutación

Ejercicio 3.14.6. *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]
```

Solución:

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

3.14.6. Determinación de permutaciones

Ejercicio 3.14.7. *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
```

Solución:

```
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 3.14.8. *Comprobar con QuickCheck que la ordenación por mezcla de una lista es una permutación de la lista.*

Solución: La propiedad es

```
prop_ordMezcla_pemutacion :: Ord a => [a] -> Bool
prop_ordMezcla_pemutacion xs = esPermutacion (ordMezcla xs) xs
```

La comprobación es

```
ghci> quickCheck prop_ordMezcla_permutacion
+++ OK, passed 100 tests.
```

Capítulo 4

Definiciones por recursión y por comprensión

En este capítulo se presentan ejercicios con dos definiciones (una por recursión y otra por comprensión). Los ejercicios se corresponden con los temas 5 y 6 de [1].

Contenido

4.1 Suma de los cuadrados de los primeros números **68**

4.2 Número de bloques de escaleras triangulares **70**

4.3 Suma de los cuadrados de los impares entre los primeros números . . **71**

4.4 Operaciones con los dígitos de los números **72**

4.4.1 Lista de los dígitos de un número 72

4.4.2 Suma de los dígitos de un número 73

4.4.3 Decidir si es un dígito del número 74

4.4.4 Número de dígitos de un número 74

4.4.5 Número correspondiente a una lista de dígitos 75

4.4.6 Concatenación de dos números 75

4.4.7 Primer dígito de un número 76

4.4.8 Último dígito de un número 77

4.4.9 Número con los dígitos invertidos 78

4.4.10 Decidir si un número es capicúa 79

4.4.11 Suma de los dígitos de 2^{1000} 79

4.4.12 Primitivo de un número 80

4.4.13 Números con igual media de sus dígitos 80

4.4.14 Números con dígitos duplicados en su cuadrado 81

4.5 Cuadrados de los elementos de una lista 82
4.6 Números impares de una lista 83
4.7 Cuadrados de los elementos impares 84
4.8 Suma de los cuadrados de los elementos impares 85
4.9 Intervalo numérico 86
4.10 Mitades de los pares 87
4.11 Pertenencia a un rango 88
4.12 Suma de elementos positivos 89
4.13 Aproximación del número π 90
4.14 Sustitución de impares por el siguiente par 90
4.15 La compra de una persona agarrada 91
4.16 Descomposición en productos de factores primos 92
4.16.1 Lista de los factores primos de un número 92
4.16.2 Decidir si un número es primo 93
4.16.3 Factorización de un número 93
4.16.4 Exponente de la mayor potencia de un número que divide a otro 93
4.16.5 Expansion de la factorización de un número 94
4.17 Menor número con todos los dígitos en la factorización de su factorial 95
4.18 Suma de números especiales 98
4.19 Distancia de Hamming 99
4.20 Traspuesta de una matriz 100
4.21 Números expresables como sumas acotadas de elementos de una lista 101

Nota. Se usarán las librerías `List` y `QuickCheck`.

```
import Data.List
import Test.QuickCheck
```

4.1. Suma de los cuadrados de los primeros números

Ejercicio 4.1.1. *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
```

Solución:

```
sumaCuadradosR :: Integer -> Integer
sumaCuadradosR 0 = 0
sumaCuadradosR n = n^2 + sumaCuadradosR (n-1)
```

Ejercicio 4.1.2. *Comprobar con QuickCheck si* `(sumaCuadradosR n)` *es igual a* $\frac{n(n+1)(2n+1)}{6}$.

Solución: La propiedad es

```
prop_SumaCuadrados n =
  n >= 0 ==>
    sumaCuadradosR n == n * (n+1) * (2*n+1) `div` 6
```

La comprobación es

```
ghci> quickCheck prop_SumaCuadrados
OK, passed 100 tests.
```

Ejercicio 4.1.3. *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
```

Solución:

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

Ejercicio 4.1.4. *Comprobar con QuickCheck que las funciones* `sumaCuadradosR` *y* `sumaCuadradosC` *son equivalentes sobre los números naturales.*

Solución: La propiedad es

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

La comprobación es

```
ghci> quickCheck prop_sumaCuadrados
+++ OK, passed 100 tests.
```

4.2. Número de bloques de escaleras triangulares

Ejercicio 4.2.1. *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
```

Solución:

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

Ejercicio 4.2.2. *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
```

Solución:

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

Ejercicio 4.2.3. *Comprobar con QuickCheck que* `(numeroBloquesC n)` *es igual a* $n + n^2$.

Solución: La propiedad es

```
prop_numeroBloques n =
    n > 0 ==> numeroBloquesC n == n+n^2
```

La comprobación es

```
ghci> quickCheck prop_numeroBloques
+++ OK, passed 100 tests.
```

4.3. Suma de los cuadrados de los impares entre los primeros números

Ejercicio 4.3.1. *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`*. Por ejemplo,*

```
sumaCuadradosImparesR 1  ==  1
sumaCuadradosImparesR 7  ==  84
sumaCuadradosImparesR 4  ==  10
```

Solución:

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

Ejercicio 4.3.2. *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`*. Por ejemplo,*

```
sumaCuadradosImparesC 1  ==  1
sumaCuadradosImparesC 7  ==  84
sumaCuadradosImparesC 4  ==  10
```

Solución:

```
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]]
```

4.4. Operaciones con los dígitos de los números

4.4.1. Lista de los dígitos de un número

Ejercicio 4.4.1. *Definir, por recursión, la función*

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

tal que `(digitosR n)` *es la lista de los dígitos del número* `n`*. Por ejemplo,*

```
digitosR 320274 == [3,2,0,2,7,4]
```

Solución:

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

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

Ejercicio 4.4.2. *Definir, por comprensión, la función*

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

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`*.*

Solución:

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

Ejercicio 4.4.3. *Comprobar con QuickCheck que las funciones* `digitosR` *y* `digitos` *son equivalentes.*

Solución: La propiedad es

```
prop_dígitos n =
    n >= 0 ==>
    digitosR n == digitosC n
```

La comprobación es

```
ghci> quickCheck prop_dígitos
+++ OK, passed 100 tests.
```

4.4.2. Suma de los dígitos de un número

Ejercicio 4.4.4. *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
```

Solución:

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

Ejercicio 4.4.5. *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
```

Solución:

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

Ejercicio 4.4.6. *Comprobar con QuickCheck que las funciones* `sumaDigitosR` *y* `sumaDigitosNR` *son equivalentes.*

Solución: La propiedad es

```
prop_sumaDígitos n =
    n >= 0 ==>
    sumaDigitosR n == sumaDigitosNR n
```

La comprobación es

```
ghci> quickCheck prop_sumaDígitos
+++ OK, passed 100 tests.
```

4.4.3. Decidir si es un dígito del número

Ejercicio 4.4.7. *Definir la función*

```
esDigito :: Integer -> Integer -> Bool
```

tal que `(esDigito x n)` *se verifica si* `x` *es una dígito de* `n`*. Por ejemplo,*

```
esDigito 4 1041  ==  True
esDigito 3 1041  ==  False
```

Solución:

```
esDigito :: Integer -> Integer -> Bool
esDigito x n = elem x (digitosR n)
```

4.4.4. Número de dígitos de un número

Ejercicio 4.4.8. *Definir la función*

```
numeroDeDigitos :: Integer -> Integer
```

tal que `(numeroDeDigitos x)` *es el número de dígitos de* `x`*. Por ejemplo,*

```
numeroDeDigitos 34047 == 5
```

Solución:

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

4.4.5. Número correspondiente a una lista de dígitos

Ejercicio 4.4.9. *Definir, por recursión, la función*

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

tal que `(listaNumeroR xs)` *es el número formado por los dígitos de la lista* `xs`*. Por ejemplo,*

```
listaNumeroR [5]        == 5
listaNumeroR [1,3,4,7]  == 1347
listaNumeroR [0,0,1]    == 1
```

Solución:

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

listaNumeroR' :: [Integer] -> Integer
listaNumeroR' [x]    = x
listaNumeroR' (x:xs) = x + 10 * (listaNumeroR' xs)
```

Ejercicio 4.4.10. *Definir, por comprensión, la función*

```
listaNumeroC :: [Integer] -> Integer
```

tal que `(listaNumeroC xs)` *es el número formado por los dígitos de la lista* `xs`*. Por ejemplo,*

```
listaNumeroC [5]        == 5
listaNumeroC [1,3,4,7]  == 1347
listaNumeroC [0,0,1]    == 1
```

Solución:

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

4.4.6. Concatenación de dos números

Ejercicio 4.4.11. *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
```

Solución:

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

Ejercicio 4.4.12. *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
```

Solución:

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

Ejercicio 4.4.13. *Comprobar con QuickCheck que las funciones* `pegaNumerosR` *y* `pegaNumerosNR` *son equivalentes.*

Solución: La propiedad es

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

La comprobción es

```
ghci> quickCheck prop_pegaNumeros
+++ OK, passed 100 tests.
```

4.4.7. Primer dígito de un número

Ejercicio 4.4.14. *Definir, por recursión, la función*

```
primerDigitoR :: Integer -> Integer
```

tal que `(primerDigitoR n)` *es el primer dígito de* `n`*. Por ejemplo,*

```
primerDigitoR 425 == 4
```

Solución:

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

Ejercicio 4.4.15. *Definir, sin usar recursión, la función*

```
primerDigitoNR :: Integer -> Integer
```

tal que `(primerDigitoNR n)` *es el primer dígito de* `n`*. Por ejemplo,*

```
primerDigitoNR 425 == 4
```

Solución:

```
primerDigitoNR :: Integer -> Integer
primerDigitoNR n = head (digitosR n)
```

Ejercicio 4.4.16. *Comprobar con QuickCheck que las funciones* `primerDigitoR` *y* `primerDigitoNR` *son equivalentes.*

Solución: La propiedad es

```
prop_primerDigito x =
    x >= 0 ==>
    primerDigitoR x == primerDigitoNR x
```

La comprobación es

```
ghci> quickCheck prop_primerDigito
+++ OK, passed 100 tests.
```

4.4.8. Último dígito de un número

Ejercicio 4.4.17. *Definir la función*

```
ultimoDigito :: Integer -> Integer
```

tal que `(ultimoDigito n)` *es el último dígito de* `n`*. Por ejemplo,*

```
ultimoDigito 425 == 5
```

Solución:

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

4.4.9. Número con los dígitos invertidos

Ejercicio 4.4.18. *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
```

Solución:

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

Ejercicio 4.4.19. *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
```

Solución:

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

Ejercicio 4.4.20. *Comprobar con QuickCheck que las funciones* `inverso` *e* `inverso'` *son equivalentes.*

Solución: La propiedad es

```
prop_inverso n =
    n >= 0 ==>
    inverso n == inverso' n
```

La comprobación es

```
ghci> quickCheck prop_inverso
+++ OK, passed 100 tests.
```

4.4.10. Decidir si un número es capicúa

Ejercicio 4.4.21. *Definir la función*

```
capicua :: Integer -> Bool
```

tal que `(capicua n)` *se verifica si los dígitos de* `n` *son los mismos de izquierda a derecha que de derecha a izquierda. Por ejemplo,*

```
capicua 1234  =  False
capicua 1221  =  True
capicua 4     =  True
```

Solución:

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

4.4.11. Suma de los dígitos de 2^{1000}

En el problema 16 del proyecto Euler[1] se pide calcular la suma de las dígitos de 2^{1000}. Lo resolveremos mediante los distintos apartados de este ejercicio.

Ejercicio 4.4.22. *Definir la función*

```
euler16 :: Integer -> Integer
```

tal que `(euler16 n)` *es la suma de los dígitos de* 2^n. *Por ejemplo,*

```
euler16 4 == 7
```

Solución:

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

Ejercicio 4.4.23. *Calcular la suma de los dígitos de* 2^{1000}.

Solución: El cálculo es

```
ghci> euler16 1000
1366
```

[1] http://projecteuler.net/problem=16

4.4.12. Primitivo de un número

Ejercicio 4.4.24. *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
```

Solución:

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

donde `(producto n)` es el producto de los dígitos de `n`. Por ejemplo,

```
producto 327 == 42
```

```
producto :: Integer -> Integer
producto = product . digitosC
```

4.4.13. Números con igual media de sus dígitos

Ejercicio 4.4.25. *Dos números son equivalentes si la media de sus dígitos son iguales. Por ejemplo, 3205 y 41 son equivalentes ya que*

$$\frac{3+2+0+5}{4} = \frac{4+1}{2}$$

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
```

Solución:

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

donde `(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))
```

4.4.14. Números con dígitos duplicados en su cuadrado

Ejercicio 4.4.26. *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.

Solución:

```
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
```

4.5. Cuadrados de los elementos de una lista

Ejercicio 4.5.1. *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]
```

Solución:

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

Ejercicio 4.5.2. *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]
```

Solución:

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

Ejercicio 4.5.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_cuadrados :: [Integer] -> Bool
prop_cuadrados xs =
    cuadradosC xs == cuadradosR xs
```

La comprobación es

```
ghci> quickCheck prop_cuadrados
+++ OK, passed 100 tests.
```

4.6. Números impares de una lista

Ejercicio 4.6.1. *Definir, por comprensión, la función*

```
imparesC :: [Integer] -> [Integer]
```

tal que `(imparesC xs)` *es la lista de los números impares de* `xs`*. Por ejemplo,*

```
imparesC [1,2,4,3,6] == [1,3]
```

Solución:

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

Ejercicio 4.6.2. *Definir, por recursión, la función*

```
imparesR :: [Integer] -> [Integer]
```

tal que `(imparesR xs)` *es la lista de los números impares de* `xs`*. Por ejemplo,*

```
imparesR [1,2,4,3,6] == [1,3]
```

Solución:

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

Ejercicio 4.6.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_impares :: [Integer] -> Bool
prop_impares xs =
    imparesC xs == imparesR xs
```

La comprobación es

```
ghci> quickCheck prop_impares
+++ OK, passed 100 tests.
```

4.7. Cuadrados de los elementos impares

Ejercicio 4.7.1. *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 [1,2,4,3,6] == [1,9]*

Solución:

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

Ejercicio 4.7.2. *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 [1,2,4,3,6] == [1,9]*

Solución:

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

Ejercicio 4.7.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_imparesCuadrados :: [Integer] -> Bool
prop_imparesCuadrados xs =
    imparesCuadradosC xs == imparesCuadradosR xs
```

La comprobación es

```
ghci> quickCheck prop_imparesCuadrados
+++ OK, passed 100 tests.
```

4.8. Suma de los cuadrados de los elementos impares

Ejercicio 4.8.1. *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 [1,2,4,3,6] == 10*

Solución:

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

Ejercicio 4.8.2. *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 [1,2,4,3,6] == 10
```

Solución:

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

Ejercicio 4.8.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_sumaCuadradosImpares :: [Integer] -> Bool
prop_sumaCuadradosImpares xs =
    sumaCuadradosImparesC xs == sumaCuadradosImparesR xs
```

La comprobación es

```
ghci> quickCheck prop_sumaCuadradosImpares
+++ OK, passed 100 tests.
```

4.9. Intervalo numérico

Ejercicio 4.9.1. *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]
```

Solución:

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

Ejercicio 4.9.2. *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]
```

Solución:

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

Ejercicio 4.9.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_entre :: Integer -> Integer -> Bool
prop_entre m n =
    entreL m n == entreR m n
```

La comprobación es

```
ghci> quickCheck prop_entre
+++ OK, passed 100 tests.
```

4.10. Mitades de los pares

Ejercicio 4.10.1. *Definir, por comprensión, la función*

```
mitadParesC :: [Int] -> [Int]
```

tal que (`mitadParesC xs`) *es la lista de las mitades de los elementos de* `xs` *que son pares. Por ejemplo,*

```
mitadParesC [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
```

Solución:

```
mitadParesC :: [Int] -> [Int]
mitadParesC xs = [x `div` 2 | x <- xs, x `mod` 2 == 0]
```

Ejercicio 4.10.2. *Definir, por recursión, la función*

```
mitadParesR :: [Int] -> [Int]
```

tal que (`mitadParesR xs`) *es la lista de las mitades de los elementos de* `xs` *que son pares. Por ejemplo,*

```
mitadParesR [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
```

Solución:

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

Ejercicio 4.10.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_mitadPares :: [Int] -> Bool
prop_mitadPares xs =
    mitadParesC xs == mitadParesR xs
```

La comprobación es

```
ghci> quickCheck prop_mitadPares
+++ OK, passed 100 tests.
```

4.11. Pertenencia a un rango

Ejercicio 4.11.1. *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 10  5 [1..15]   ==  []
enRangoC  5  5 [1..15]   ==  [5]
```

Solución:

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

Ejercicio 4.11.2. *Definir, por recursión, la función*

```
enRangoR :: Int -> Int -> [Int] -> [Int]
```

tal que `(enRangoR a b xs)` *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 10 5 [1..15]   ==  []
enRangoR  5 5 [1..15]   ==  [5]
```

Solución:

```
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 4.11.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

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

La comprobación es

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

4.12. Suma de elementos positivos

Ejercicio 4.12.1. *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
```

Solución:

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

Ejercicio 4.12.2. *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
```

Solución:

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

Ejercicio 4.12.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_sumaPositivos :: [Int] -> Bool
prop_sumaPositivos xs =
    sumaPositivosC xs == sumaPositivosR xs
```

La comprobación es

```
ghci> quickCheck prop_sumaPositivos
+++ OK, passed 100 tests.
```

4.13. Aproximación del número π

La suma de la serie

$$\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

es $\frac{\pi^2}{6}$. Por tanto, π se puede aproximar mediante la raíz cuadrada de 6 por la suma de la serie.

Ejercicio 4.13.1. *Definir, por comprensión, la función* `aproximaPiC` *tal que* `(aproximaPiC n)` *es la aproximación de* π *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
```

Solución:

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

Ejercicio 4.13.2. *Definir, por comprensión, la función* `aproximaPiR` *tal que* `(aproximaPiR n)` *es la aproximación de* π *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
```

Solución:

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

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

4.14. Sustitución de impares por el siguiente par

Ejercicio 4.14.1. *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]
```

Solución:

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

Ejercicio 4.14.2. *Comprobar con QuickChek la siguiente propiedad: para cualquier lista de números enteros* `xs`*, todos los elementos de la lista* `(sustituyeImpar xs)` *son números pares.*

Solución: 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.
```

4.15. La compra de una persona agarrada

Ejercicio 4.15.1. *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
```

Solución:

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

Ejercicio 4.15.2. *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
```

Solución:

```
agarradoR :: [Float] -> Float
agarradoR [] = 0
agarradoR (p:ps)
    | precioConDescuento <= 199 = precioConDescuento + agarradoR ps
    | otherwise                 = agarradoR ps
    where precioConDescuento = p * 0.9
```

Ejercicio 4.15.3. *Comprobar con QuickCheck que ambas definiciones son similares; es decir, el valor absoluto de su diferencia es menor que una décima.*

Solución: La propiedad es

```
prop_agarrado :: [Float] -> Bool
prop_agarrado xs = abs (agarradoR xs - agarradoC xs) <= 0.1
```

La comprobación es

```
ghci> quickCheck prop_agarrado
+++ OK, passed 100 tests.
```

4.16. Descomposición en productos de factores primos

4.16.1. Lista de los factores primos de un número

Ejercicio 4.16.1. *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]
```

Solución:

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

4.16.2. Decidir si un número es primo

Ejercicio 4.16.2. *Definir la función*

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

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

```
primo 7  ==  True
primo 9  ==  False
```

Solución:

```
primo :: Integer -> Bool
primo x = factores x == [1,x]
```

4.16.3. Factorización de un número

Ejercicio 4.16.3. *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]
```

Solución:

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

4.16.4. Exponente de la mayor potencia de un número que divide a otro

Ejercicio 4.16.4. *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 a* `b`*. Por ejemplo,*

```
mayorExponenteR 2 8    ==  3
mayorExponenteR 2 9    ==  0
mayorExponenteR 5 100  ==  2
mayorExponenteR 2 60   ==  2
```

Solución:

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

Ejercicio 4.16.5. *Definir, por comprensión, 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
```

Solución:

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

Ejercicio 4.16.6. *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)]
```

Solución:

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

4.16.5. Expansion de la factorización de un número

Ejercicio 4.16.7. *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
```

Solución:

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

Ejercicio 4.16.8. *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
```

Solución:

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

Ejercicio 4.16.9. *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
```

Solución:

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

4.17. Menor número con todos los dígitos en la factorización de su factorial

Ejercicio 4.17.1. *El enunciado del problema 652 de "Números y algo más"*[2] *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*

[2] http://simplementenumeros.blogspot.com.es/2011/04/652-desafios-del-contest-center-i.html

- $6! = 2^4 3^2 5^1$, *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.

Solución:

```
digitosDeFactorizacion :: Integer -> [Integer]
digitosDeFactorizacion n =
   sort (nub (concat [digitos x | x <- numerosDeFactorizacion n]))
```

donde se usan las siguientes funciones auxiliares

- `(digitos n)` es la lista de los digitos 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
```

4.18. Suma de números especiales

Los siguientes ejercicios están basados en el problema 357 del proyecto Euler[3].

Ejercicio 4.18.1. *Un número natural n es especial si para todo divisor d de n, $d + \frac{n}{d}$ 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
```

Solución:

```
especial :: Integer -> Bool
especial x = and [esPrimo (d + x `div` d) | d <- divisores x]
```

donde se usan las siguientes funciones auxiliares

- `(divisores x)` es la lista de los divisores de `x`. Por ejemplo,

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

```
divisores :: Integer -> [Integer]
divisores x = [d | d <- [1..x], x `rem` d == 0]
```

- `(esPrimo x)` se verifica si `x` es primo. Por ejemplo,

```
esPrimo 7  ==  True
esPrimo 8  ==  False
```

```
esPrimo :: Integer -> Bool
esPrimo x = divisores x == [1,x]
```

Ejercicio 4.18.2. *Definir, por comprensión, 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
```

[3] http://projecteuler.net/problem=357

Solución:

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

Ejercicio 4.18.3. *Definir, por recursion, la función*

```
sumaEspecialesR :: Integer -> Integer
```

tal que `(sumaEspecialesR n)` *es la suma de los números especiales menores o iguales que* `n`. *Por ejemplo,*

```
sumaEspecialesR 100 == 401
```

Solución:

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

4.19. Distancia de Hamming

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).

Ejercicio 4.19.1. *Definir, por comprensión, la función*

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

tal que `(distanciaC xs ys)` *es la distanciaC 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
```

Solución:

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

Ejercicio 4.19.2. *Definir, por recursión, la función*

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

tal que `(distanciaR xs ys)` *es la distanciaR 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
```

Solución:

```
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 4.19.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_distancia :: [Int] -> [Int] -> Bool
prop_distancia xs ys =
    distanciaC xs ys == distanciaR xs ys
```

La comprobación es

```
ghci> quickCheck prop_distancia
+++ OK, passed 100 tests.
```

4.20. Traspuesta de una matriz

Ejercicio 4.20.1. *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]]
```

Solución:

```
traspuesta :: [[a]] -> [[a]]
traspuesta []           = []
traspuesta ([]:xss)     = traspuesta xss
traspuesta ((x:xs):xss) =
    (x:[h | (h:_) <- xss]) : traspuesta (xs : [t | (_:t) <- xss])
```

4.21. Números expresables como sumas acotadas de elementos de una lista

Ejercicio 4.21.1. *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]
```

Solución:

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

Capítulo 5

Funciones sobre cadenas

En este capítulo se presentan ejercicios sobre cadenas. Se corresponden con el tema 5 de [1].

Contenido

5.1 Suma de los dígitos de una cadena 103
5.2 Capitalización de una cadena 105
5.3 Título con las reglas de mayúsculas iniciales 106
5.4 Búsqueda en crucigramas 107
5.5 Posiciones de un carácter en una cadena 108
5.6 Decidir si una cadena es subcadena de otra 109
5.7 Codificación de mensajes 111
5.8 Números de ceros finales 115

Nota. En esta relación se usan las librerias `Char` y `QuickCheck`.

```
import Data.Char
import Test.QuickCheck
```

5.1. Suma de los dígitos de una cadena

Ejercicio 5.1.1. *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` *y* `digitToInt`.

Solución:

```
sumaDigitosC :: String -> Int
sumaDigitosC xs = sum [digitToInt x | x <- xs, isDigit x]
```

Ejercicio 5.1.2. *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`.

Solución:

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

Ejercicio 5.1.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_sumaDigitos :: String -> Bool
prop_sumaDigitos xs =
    sumaDigitosC xs == sumaDigitosR xs
```

La comprobación es

```
ghci> quickCheck prop_sumaDigitos
+++ OK, passed 100 tests.
```

5.2. Capitalización de una cadena

Ejercicio 5.2.1. *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`*y* `toUpper`*.*

Solución:

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

Ejercicio 5.2.2. *Definir, por recursión, la función*

```
mayusculaInicialR :: String -> String
```

tal que (`mayusculaInicialR xs`) *es la palabra* `xs` *con la letra inicial en mayúscula y las restantes en minúsculas. Por ejemplo,*

```
mayusculaInicialR "sEviLLa" == "Sevilla"
```

Solución:

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

Ejercicio 5.2.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_mayusculaInicial :: String -> Bool
prop_mayusculaInicial xs =
    mayusculaInicial xs == mayusculaInicialR xs
```

La comprobación es

```
ghci> quickCheck prop_mayusculaInicial
+++ OK, passed 100 tests.
```

5.3. Título con las reglas de mayúsculas iniciales

Ejercicio 5.3.1. *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"]
```

Solución:

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

donde `(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
```

y `(minuscula xs)` es la palabra `xs` en minúscula.

```
minuscula :: String -> String
minuscula xs = [toLower x | x <- xs]
```

Ejercicio 5.3.2. *Definir, por recursión, la función*

```
tituloR :: [String] -> [String]
```

tal que `(tituloR ps)` *es la lista de las palabras de* `ps` *con las reglas de mayúsculas iniciales de los títulos. Por ejemplo,*

```
ghci> tituloR ["eL","arTE","DE","La","proGraMacion"]
["El","Arte","de","la","Programacion"]
```

Solución:

```
tituloR :: [String] -> [String]
tituloR []     = []
tituloR (p:ps) = mayusculaInicial p : tituloRAux ps
    where tituloRAux []     = []
          tituloRAux (p:ps) = transforma p : tituloRAux ps
```

Ejercicio 5.3.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: La propiedad es

```
prop_titulo :: [String] -> Bool
prop_titulo xs = titulo xs == tituloR xs
```

La comprobación es

```
ghci> quickCheck prop_titulo
+++ OK, passed 100 tests.
```

5.4. Búsqueda en crucigramas

Ejercicio 5.4.1. *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", "acabado", "ocupado"]
["acabado","ocupado"]
```

Solución:

```
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 5.4.2. *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 posen la letra* `l` *en la posición* `pos` *(comenzando en 0). Por ejemplo,*

```
ghci> buscaCrucigramaR 'c' 1 7 ["ocaso", "acabado", "ocupado"]
["acabado","ocupado"]
```

Solución:

```
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 5.4.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: 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.
```

5.5. Posiciones de un carácter en una cadena

Ejercicio 5.5.1. *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]
```

Solución:

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

Ejercicio 5.5.2. *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]
```

Solución:

```
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 5.5.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: 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.
```

5.6. Decidir si una cadena es subcadena de otra

Ejercicio 5.6.1. *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`*.*

Solución:

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

Ejercicio 5.6.2. *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`*.*

Solución:

```
contiene :: String -> String -> Bool
contiene xs ys = sufijosComenzandoCon xs ys /= []
```

donde `(sufijosComenzandoCon xs ys)` es la lista de los sufijos de `xs` que comienzan con `ys`. Por ejemplo,

```
sufijosComenzandoCon "abacbad" "ba"  ==  ["bacbad","bad"]
```

```
sufijosComenzandoCon :: String -> String -> [String]
sufijosComenzandoCon xs ys = [x | x <- sufijos xs, isPrefixOf ys x]
```

y `(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]]
```

Ejercicio 5.6.3. *Comprobar con QuickCheck que ambas definiciones son equivalentes.*

Solución: 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.
```

5.7. Codificación de mensajes

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 inverte 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 esta sección se definirá el anterior proceso de codificación.

Ejercicio 5.7.1. *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])
```

Solución:

```
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 5.7.2. *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"]
```

Solución:

```
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 5.7.3. *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]
```

Solución:

```
longitudes :: [[a]] -> [Int]
longitudes = map length
```

Ejercicio 5.7.4. *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"
```

Solución:

```
une :: [[a]] -> [a]
une = concat
```

Ejercicio 5.7.5. *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"]
```

Solución:

```
reagrupa :: [a] -> [[a]]
reagrupa [] = []
reagrupa xs = take 4 xs : reagrupa (drop 4 xs)
```

Ejercicio 5.7.6. *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"
```

Solución:

```
inversas :: [[a]] -> [[a]]
inversas = map reverse
```

Ejercicio 5.7.7. *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"]
```

Solución:

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

Ejercicio 5.7.8. *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"
```

Solución:

```
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 5.7.9. *Definir la función*

```
clave :: String -> String
```

que realice el proceso completo. Por ejemplo,

```
clave "eres lo que piensas" == "sereuqo lei pe sasn"
```

Solución:

```
clave :: String -> String
clave xss = frase (agrupa (une (inversas (reagrupa (une ps))))
                          (reverse (longitudes ps)))
    where ps = palabras xss
```

5.8. Números de ceros finales

Ejercicio 5.8.1. *Definir, por recursión, la función*

```
ceros :: Int -> Int
```

tal que `(ceros n)` *es el número de ceros en los que termina el número* `n`*. Por ejemplo,*

```
ceros 30500  ==  2
ceros 30501  ==  0
```

Solución:

```
ceros :: Int -> Int
ceros n | n `rem` 10 == 0 = 1 + ceros (n `div`10)
        | otherwise       = 0
```

Ejercicio 5.8.2. *Definir, sin recursión, la función*

```
ceros' :: Int -> Int
```

tal que `(ceros' n)` *es el número de ceros en los que termina el número* `n`*. Por ejemplo,*

```
ceros' 30500  ==  2
ceros' 30501  ==  0
```

Solución:

```
ceros' :: Int -> Int
ceros' n = length (takeWhile (=='0') (reverse (show n)))
```

Capítulo 6

Funciones de orden superior

En este capítulo se presentan ejercicios para definir funciones de orden superior. Se corresponde con el tema 7 de [1].

Contenido

6.1 Segmento inicial verificando una propiedad . . . 118
6.2 Complementario del segmento inicial verificando una propiedad . . . 118
6.3 Concatenación de una lista de listas . . . 119
6.4 División de una lista numérica según su media . . . 119
6.5 Segmentos cuyos elementos verifican una propiedad . . . 122
6.6 Listas con elementos consecutivos relacionados . . . 122
6.7 Agrupamiento de elementos de una lista de listas . . . 123
6.8 Números con dígitos pares . . . 123
6.9 Lista de los valores de los elementos que cumplen una propiedad . . . 125
6.10 Máximo elemento de una lista . . . 126
6.11 Mínimo elemento de una lista . . . 127
6.12 Inversa de una lista . . . 127
6.13 Número correspondiente a la lista de sus cifras . . . 130
6.14 Suma de valores de una aplicación a una lista . . . 131
6.15 Redefinición de la función `map` usando `foldr` . . . 132
6.16 Redefinición de la función `filter` usando `foldr` . . . 132
6.17 Suma de las sumas de las listas de una lista de listas . . . 133
6.18 Lista obtenida borrando las ocurrencias de un elemento . . . 134

6.19 Diferencia de dos listas 135
6.20 Producto de los números que verifican una propiedad 136
6.21 Las cabezas y las colas de una lista 137

6.1. Segmento inicial verificando una propiedad

Ejercicio 6.1.1. *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]
```

Solución:

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

6.2. Complementario del segmento inicial verificando una propiedad

Ejercicio 6.2.1. *Redefinir por recursión la función*

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

tal que `(dropWhile p xs)` *es la lista obtenida 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]
```

Solución:

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

6.3. Concatenación de una lista de listas

Ejercicio 6.3.1. *Redefinir, por recursión, la función* `concat`*. Por ejemplo,*

```
concatR [[1,3],[2,4,6],[1,9]] == [1,3,2,4,6,1,9]
```

Solución:

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

Ejercicio 6.3.2. *Redefinir, usando* `foldr`*, la función* `concat`*. Por ejemplo,*

```
concatP [[1,3],[2,4,6],[1,9]] == [1,3,2,4,6,1,9]
```

Solución:

```
concatP :: [[a]] -> [a]
concatP = foldr (++) []
```

6.4. División de una lista numérica según su media

Ejercicio 6.4.1. *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.*

Solución: La definición por filtrado es

```
divideMediaF :: [Double] -> ([Double],[Double])
divideMediaF xs = (filter (<m) xs, filter (>m) xs)
    where m = media xs
```

donde `(media xs)` es la media de `xs`. Por ejemplo,

```
media [1,2,3]       ==  2.0
media [1,-2,3.5,4]  ==  1.625
```

```
media :: [Double] -> Double
media xs = (sum xs) / fromIntegral (length xs)
```

En la definición de `media` se usa la función `fromIntegral` tal que `(fromIntegral x)` es el número real correspondiente al número entero `x`.

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 6.4.2. *Comprobar con QuickCheck que las tres definiciones anteriores* `divideMediaF`*,* `divideMediaC` *y* `divideMediaR` *son equivalentes.*

Solución: 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 6.4.3. *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`*.*

Solución: 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 6.4.4. *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`*.*

Solución: 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 6.4.5. *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`*.*

Solución: 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.
```

6.5. Segmentos cuyos elementos verifican una propiedad

Ejercicio 6.5.1. *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]]
```

Solución:

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

6.6. Listas con elementos consecutivos relacionados

Ejercicio 6.6.1. *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
```

Solución:

```
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)]
```

6.7. Agrupamiento de elementos de una lista de listas

Ejercicio 6.7.1. *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, ... de forma que las longitudes de las lista del resultado sean iguales a la más corta de* `xss`*. Por ejemplo,*

```
agrupa [[1..6],[7..9],[10..20]]  ==  [[1,7,10],[2,8,11],[3,9,12]]
agrupa []                        ==  []
```

Solución:

```
agrupa :: Eq a => [[a]] -> [[a]]
agrupa []  = []
agrupa xss
    | [] `elem` xss = []
    | otherwise     = primeros xss : agrupa (restos xss)
    where primeros = map head
          restos   = map tail
```

6.8. Números con dígitos pares

Ejercicio 6.8.1. *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
```

Solución:

```
superpar :: Int -> Bool
superpar n | n < 10    = even n
           | otherwise = even n && superpar (n `div` 10)
```

Ejercicio 6.8.2. *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
```

Solución:

```
superpar2 :: Int -> Bool
superpar2 n = and [even d | d <- digitos n]
```

Donde `(digitos n)` es la lista de los dígitos de `n`.

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

Ejercicio 6.8.3. *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
```

Solución:

```
superpar3 :: Int -> Bool
superpar3 n = sonPares (digitos n)
    where sonPares []     = True
          sonPares (d:ds) = even d && sonPares ds
```

Ejercicio 6.8.4. *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
```

Solución:

```
superpar4 :: Int -> Bool
superpar4 n = all even (digitos n)
```

Ejercicio 6.8.5. *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
```

Solución:

```
superpar5 :: Int -> Bool
superpar5 n = filter even (digitos n) == digitos n
```

6.9. Lista de los valores de los elementos que cumplen una propiedad

Ejercicio 6.9.1. *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`*).*

Solución: 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) []
```

6.10. Máximo elemento de una lista

Ejercicio 6.10.1. *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
```

Nota: *La función* `maximumR` *es equivalente a la predefinida* `maximum`*.*

Solución:

```
maximumR :: Ord a => [a] -> a
maximumR [x]      = x
maximumR (x:y:ys) = max x (maximumR (y:ys))
```

Ejercicio 6.10.2. *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
```

Nota: *La función* `maximumP` *es equivalente a la predefinida* `maximum`*.*

Solución:

```
maximumP :: Ord a => [a] -> a
maximumP = foldr1 max
```

6.11. Mínimo elemento de una lista

Ejercicio 6.11.1. *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
```

Nota: *La función* `minimunP` *es equivalente a la predefinida* `minimun`*.*

Solución:

```
minimumP :: Ord a => [a] -> a
minimumP = foldr1 min
```

6.12. Inversa de una lista

Ejercicio 6.12.1. *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]
```

Solución:

```
inversaR :: [a] -> [a]
inversaR []     = []
inversaR (x:xs) = (inversaR xs) ++ [x]
```

Ejercicio 6.12.2. *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]
```

Solución:

```
inversaP :: [a] -> [a]
inversaP = foldr f []
    where f x y = y ++ [x]
```

La definición anterior puede simplificarse a

```
inversaP_2 :: [a] -> [a]
inversaP_2 = foldr f []
    where f x = (++ [x])
```

Ejercicio 6.12.3. *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]
```

Solución:

```
inversaR' :: [a] -> [a]
inversaR' xs = inversaAux [] xs
    where inversaAux ys []     = ys
          inversaAux ys (x:xs) = inversaAux (x:ys) xs
```

Ejercicio 6.12.4. *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]
```

Solución:

```
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 6.12.5. *Comprobar con QuickCheck que las funciones* `reverse`*,* `inversaP` *e* `inversaP'` *son equivalentes.*

Solución: 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 6.12.6. *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])
```

Solución: 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
```

6.13. Número correspondiente a la lista de sus cifras

Ejercicio 6.13.1. *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
```

Solución:

```
dec2entR :: [Int] -> Int
dec2entR xs = dec2entR' 0 xs
    where dec2entR' a []     = a
          dec2entR' a (x:xs) = dec2entR' (10*a+x) xs
```

Ejercicio 6.13.2. *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
```

Solución:

```
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
```

6.14. Suma de valores de una aplicación a una lista

Ejercicio 6.14.1. *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
```

Solución:

```
sumaR :: Num b => (a -> b) -> [a] -> b
sumaR f []     = 0
sumaR f (x:xs) = f x + sumaR f xs
```

Ejercicio 6.14.2. *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
```

Solución:

```
sumaP :: Num b => (a -> b) -> [a] -> b
sumaP f = foldr (\x y -> (f x) + y) 0
```

6.15. Redefinición de la función `map` usando `foldr`

Ejercicio 6.15.1. *Redefinir, por recursión, la función* `map`*. Por ejemplo,*

```
mapR (+2) [1,7,3] == [3,9,5]
```

Solución:

```
mapR :: (a -> b) -> [a] -> [b]
mapR f [] = []
mapR f (x:xs) = f x : mapR f xs
```

Ejercicio 6.15.2. *Redefinir, usando* `foldr`*, la función* `map`*. Por ejemplo,*

```
mapP (+2) [1,7,3] == [3,9,5]
```

Solución:

```
mapP :: (a -> b) -> [a] -> [b]
mapP f = foldr g []
         where g x xs = f x : xs
```

La definición por plegado usando lambda es

```
mapP' :: (a -> b) -> [a] -> [b]
mapP' f = foldr (\x y -> f x:y) []
```

Otra definición es

```
mapP'' :: (a -> b) -> [a] -> [b]
mapP'' f = foldr ((:) . f) []
```

6.16. Redefinición de la función `filter` usando `foldr`

Ejercicio 6.16.1. *Redefinir, por recursión, la función* `filter`*. Por ejemplo,*

```
filterR (<4) [1,7,3,2] => [1,3,2]
```

Solución:

```
filterR :: (a -> Bool) -> [a] -> [a]
filterR p [] = []
filterR p (x:xs) | p x       = x : filterR p xs
                 | otherwise = filterR p xs
```

Ejercicio 6.16.2. *Redefinir, usando* `foldr`*, la función* `filter`*. Por ejemplo,*

```
filterP (<4) [1,7,3,2] => [1,3,2]
```

Solución:

```
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

```
filterP' :: (a -> Bool) -> [a] -> [a]
filterP' p = foldr (\x y -> if (p x) then (x:y) else y) []
```

6.17. Suma de las sumas de las listas de una lista de listas

Ejercicio 6.17.1. *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
```

Solución:

```
sumllR :: Num a => [[a]] -> a
sumllR []       = 0
sumllR (xs:xss) = sum xs + sumllR xss
```

Ejercicio 6.17.2. *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
```

Solución:

```
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 6.17.3. *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
```

Solución:

```
sumllA :: Num a => [[a]] -> a
sumllA xs = aux 0 xs
       where aux a [] = a
             aux a (xs:xss) = aux (a + sum xs) xss
```

Ejercicio 6.17.4. *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
```

Solución:

```
sumllAP :: Num a => [[a]] -> a
sumllAP = foldl (\a xs -> a + sum xs) 0
```

6.18. Lista obtenida borrando las ocurrencias de un elemento

Ejercicio 6.18.1. *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]
```

Solución:

```
borraR :: Eq a => a -> [a] -> [a]
borraR z [] = []
borraR z (x:xs) | z == x    = borraR z xs
                | otherwise = x : borraR z xs
```

Ejercicio 6.18.2. *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]
```

Solución:

```
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) []
```

6.19. Diferencia de dos listas

Ejercicio 6.19.1. *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]
```

Solución:

```
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 de plegado, 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 6.19.2. *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]
```

Solución:

```
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)
```

6.20. Producto de los números que verifican una propiedad

Ejercicio 6.20.1. *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
```

Solución:

```
producto :: Num a => [a] -> a
producto = foldr (*) 1
```

Ejercicio 6.20.2. *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
```

Solución:

```
productoPred :: Num a => (a -> Bool) -> [a] -> a
productoPred p = foldr (\x y -> if p x then x*y else y) 1
```

Ejercicio 6.20.3. *Definir la función*

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

tal que `(productoPos xs)` *esel producto de los elementos estríctamente positivos de la lista* `xs`. *Por ejemplo,*

```
productoPos [2,1,-3,4,-5,6] == 48
```

Solución:

```
productoPos :: (Num a, Ord a) => [a] -> a
productoPos = productoPred (>0)
```

6.21. Las cabezas y las colas de una lista

Ejercicio 6.21.1. *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],[]]
colas [4,1,2,5] == [[4,1,2,5],[1,2,5],[2,5],[5],[]]
```

Solución:

```
colas :: [a] -> [[a]]
colas []     = [[]]
colas (x:xs) = (x:xs) : colas xs
```

Ejercicio 6.21.2. *Comprobar con QuickCheck que las funciones* `colas` *y* `tails` *son equivalentes.*

Solución: 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 6.21.3. *Se denomina cabeza de una lista* `xs` *a una sublista no vacía de* `xs` *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]]
```

Solución:

```
cabezas :: [a] -> [[a]]
cabezas []     = [[]]
cabezas (x:xs) = [] : [x:ys | ys <- cabezas xs]
```

Ejercicio 6.21.4. *Definir, por plegado, la función*

```
cabezasP :: [a] -> [[a]]
```

tal que `(cabezasP xs)` *es la lista de las cabezasP 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]]
```

Solución:

```
cabezasP :: [a] -> [[a]]
cabezasP = foldr (\x y -> [x]:[x:ys | ys <- y]) []
```

Ejercicio 6.21.5. *Definir, mediante funciones de orden superior, la función*

```
cabezasS :: [a] -> [[a]]
```

tal que `(cabezasS xs)` *es la lista de las cabezasS 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]]
```

Solución:

```
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 6.21.6. *Comprobar con QuickCheck que las funciones* `cabezas` *y* `inits` *son equivalentes.*

Solución: 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.
```

Nota. Un caso de estudio para las funciones de orden superior es el capítulo 16 *"Codificación y transmisión de mensajes"* (página 331).

Capítulo 7

Listas infinitas

En este capítulo se presentan ejercicios para definir funciones que usan listas infinitas y evaluación perezosa. Se corresponde con el tema 10 de [1].

Contenido

7.1 Lista obtenida repitiendo un elemento 142
7.2 Lista obtenida repitiendo cada elemento según su posición 144
7.3 Potencias de un número menores que otro dado 144
7.4 Múltiplos cuyos dígitos verifican una propiedad 145
7.5 Aplicación iterada de una función a un elemento 145
7.6 Agrupamiento de elementos consecutivos 146
7.7 La sucesión de Collatz 148
7.8 Números primos 150
7.9 Descomposiciones como suma de dos primos 151
7.10 Números expresables como producto de dos primos 152
7.11 Números muy compuestos 152
7.12 Suma de números primos truncables 154
7.13 Primos permutables 155
7.14 Ordenación de los números enteros 155
7.15 La sucesión de Hamming 157
7.16 Suma de los primos menores que n 160
7.17 Menor número triangular con más de n divisores 161
7.18 Números primos consecutivos con dígitos con igual media 162

7.19 Decisión de pertenencia al rango de una función creciente 163
7.20 Pares ordenados por posición 163
7.21 Aplicación iterada de una función 165
7.22 Expresión de un número como suma de dos de una lista 165
7.23 La bicicleta de Turing 166
7.24 Sucesión de Golomb 167

7.1. Lista obtenida repitiendo un elemento

Ejercicio 7.1.1. *Definir, por recursión, la función*

```
repite :: a -> [a]
```

tal que `(repite x)` *es la lista infinita cuyos elementos son* `x`*. Por ejemplo,*

```
repite 5           == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
take 3 (repite 5)  ==  [5,5,5]
```

Nota: *La función* `repite` *es equivalente a la función* `repeat` *definida en el preludio de Haskell.*

Solución:

```
repite :: a -> [a]
repite x = x : repite x
```

Ejercicio 7.1.2. *Definir, por comprensión, la función*

```
repiteC :: a -> [a]
```

tal que `(repiteC x)` *es la lista infinita cuyos elementos son* `x`*. Por ejemplo,*

```
repiteC 5           == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
take 3 (repiteC 5)  ==  [5,5,5]
```

Solución:

```
repiteC :: a -> [a]
repiteC x = [x | _ <- [1..]]
```

Ejercicio 7.1.3. *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 preludio de Haskell.*

Solución:

```
repiteFinita :: Int -> a -> [a]
repiteFinita 0 x = []
repiteFinita n x = x : repiteFinita (n-1) x
```

Ejercicio 7.1.4. *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]
```

Solución:

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

Ejercicio 7.1.5. *Definir, por 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]
```

Solución:

```
repiteFinita' :: Int -> a -> [a]
repiteFinita' n x = take n (repite x)
```

7.2. Lista obtenida repitiendo cada elemento según su posición

Ejercicio 7.2.1. *Definir, por comprensión, la función*

```
ecoC :: [a] -> [a]
```

tal que `(ecoC xs)` *es la lista obtenida a partir de la lista* `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"
take 10 (ecoC [1..])  ==  [1,2,2,3,3,3,4,4,4,4]
```

Solución:

```
ecoC :: [a] -> [a]
ecoC xs = concat [replicate i x | (i,x) <- zip [1..] xs]
```

Ejercicio 7.2.2. *Definir, por recursión, la función*

```
ecoR :: [a] -> [a]
```

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"
take 10 (ecoR [1..])  ==  [1,2,2,3,3,3,4,4,4,4]
```

Solución:

```
ecoR :: [a] -> [a]
ecoR xs = aux 1 xs
    where aux n [] = []
          aux n (x:xs) = replicate n x ++ aux (n+1) xs
```

7.3. Potencias de un número menores que otro dado

Ejercicio 7.3.1. *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]
```

Solución:

```
potenciasMenores :: Int -> Int -> [Int]
potenciasMenores x y = takeWhile (<y) (map (x^) [1..])
```

7.4. Múltiplos cuyos dígitos verifican una propiedad

Ejercicio 7.4.1 (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 todos 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]
```

Solución:

```
multiplosRestringidos :: Int -> (Int -> Bool) -> [Int]
multiplosRestringidos n p =
    [y | y <- [n,2*n..], all p (digitos y)]
```

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

```
digitos 327 == [3,2,7]
```

```
digitos :: Int -> [Int]
digitos n = [read [x] | x <- show n]
```

7.5. Aplicación iterada de una función a un elemento

Ejercicio 7.5.1. *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,Interrupted!
```

Nota: *La función* `itera` *es equivalente a la función* `iterate` *definida en el preludio de Haskell.*

Solución:

```
itera :: (a -> a) -> a -> [a]
itera f x = x : itera f (f x)
```

7.6. Agrupamiento de elementos consecutivos

Ejercicio 7.6.1. *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"]
```

Solución:

```
agrupa :: Int -> [a] -> [[a]]
agrupa n [] = []
agrupa n xs = take n xs : agrupa n (drop n xs)
```

Ejercicio 7.6.2. *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"]
```

Solución:

```
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 7.6.3. *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.*

Solución: La primera propiedad es

```
prop_AgrupaLongitud :: Int -> [Int] -> Property
prop_AgrupaLongitud 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_AgrupaLongitud
OK, passed 100 tests.
```

La segunda propiedad es

```
prop_AgrupaCombina :: Int -> [Int] -> Property
prop_AgrupaCombina n xs =
    n > 0 ==> concat (agrupa n xs) == xs
```

La comprobación es

```
ghci> quickCheck prop_AgrupaCombina
OK, passed 100 tests.
```

7.7. La sucesión de Collatz

Se considera 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.

Ejercicio 7.7.1. *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
```

Solución:

```
siguiente n | even n    = n `div` 2
            | otherwise = 3*n+1
```

Ejercicio 7.7.2. *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]
```

Solución:

```
collatz :: Integer -> [Integer]
collatz 1 = [1]
collatz n = n : collatz (siguiente n)
```

Ejercicio 7.7.3. *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`*.*

Solución:

```
collatz' :: Integer -> [Integer]
collatz' n = (takeWhile (/=1) (iterate siguiente n)) ++ [1]
```

Ejercicio 7.7.4. *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
```

Solución:

```
menorCollatzMayor :: Int -> Integer
menorCollatzMayor x = head [y | y <- [1..], length (collatz y) > x]
```

Ejercicio 7.7.5. *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
```

Solución:

```
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]
```

7.8. Números primos

Ejercicio 7.8.1. *Definir la constante*

```
primos :: Integral a => [a]
```

tal que `primos` *es la lista de los primos mediante la criba de Eratóstenes. Ejemplo,*

```
take 10 primos  ==  [2,3,5,7,11,13,17,19,23,29]
```

Solución:

```
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 7.8.2. *Definir la función*

```
primo :: Integral a => a -> Bool
```

tal que `(primo x)` *se verifica si* `x` *es primo. Por ejemplo,*

```
primo 7  ==  True
primo 8  ==  False
```

Solución:

```
primo :: Integral a => a -> Bool
primo x = x == head (dropWhile (<x) primos)
```

7.9. Descomposiciones como suma de dos primos

Ejercicio 7.9.1. *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)]
```

Calcular, usando la función `sumaDeDosPrimos`*, el menor número que puede escribirse de 10 formas distintas como suma de dos primos.*

Solución:

```
sumaDeDosPrimos :: Int -> [(Int,Int)]
sumaDeDosPrimos n =
    [(x,n-x) | x <- primosN, x < n-x, elem (n-x) primosN]
    where primosN = takeWhile (<=n) primos
```

donde `primos` está definida en la página 150.

El cálculo es

```
ghci> head [x | x <- [1..], length (sumaDeDosPrimos x) == 10]
114
```

7.10. Números expresables como producto de dos primos

Ejercicio 7.10.1. *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
```

Solución:

```
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
```

donde `primos` está definida en la página 150.

7.11. Números muy compuestos

Ejercicio 7.11.1. *Un número es muy compuesto si tiene más divisores que sus anteriores. Por ejemplo, 12 es muy compuesto porque tiene 6 divisores (1, 2, 3, 4, 6, 12) y todos los números del 1 al 11 tienen menos de 6 divisores.*

Definir la función

```
esMuyCompuesto :: Int -> Bool
```

tal que `(esMuyCompuesto x)` *se verifica si* `x` *es un número muy compuesto. Por ejemplo,*

```
esMuyCompuesto 24  ==  True
esMuyCompuesto 25  ==  False
```

Solución:

```
esMuyCompuesto :: Int -> Bool
esMuyCompuesto x =
    and [numeroDivisores y < n | y <- [1..x-1]]
    where n = numeroDivisores x
```

donde se usan las siguiente funciones auxiliares:

- `(numeroDivisores x)` es el número de divisores de `x`. Por ejemplo,

```
numeroDivisores 24  ==  8
```

```
numeroDivisores :: Int -> Int
numeroDivisores = length . divisores
```

- `(divisores x)` es la lista de los divisores de `x`. Por ejemplo,

```
divisores 24  ==  [1,2,3,4,6,8,12,24]
```

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

Los primeros números muy compuestos son

```
ghci> take 14 [x | x <- [1..], esMuyCompuesto x]
[1,2,4,6,12,24,36,48,60,120,180,240,360,720]
```

Ejercicio 7.11.2. *Calcular el menor número muy compuesto de 4 cifras.*

Solución: El cálculo del menor número muy compuesto de 4 cifras es

```
ghci> head [x | x <- [1000..], esMuyCompuesto x]
1260
```

Ejercicio 7.11.3. *Definir la función*

```
muyCompuesto :: Int -> Int
```

tal que `(muyCompuesto n)` *es el* `n`*–ésimo número muy compuesto. Por ejemplo,*

```
muyCompuesto 10 == 180
```

Solución:

```
muyCompuesto :: Int -> Int
muyCompuesto n =
    [x | x <- [1..], esMuyCompuesto x] !! n
```

7.12. Suma de números primos truncables

Los siguientes ejercicios están basados en el problema 37 del proyecto Euler[1].

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.

Ejercicio 7.12.1. *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
```

Solución:

```
primoTruncable :: Int -> Bool
primoTruncable x
    | x < 10    = primo x
    | otherwise = primo x && primoTruncable (x `div` 10)
```

donde se usan la función `primo` definida en la página 151.

Ejercicio 7.12.2. *Definir la función*

```
sumaPrimosTruncables :: Int -> Int
```

tal que `(sumaPrimosTruncables n)` *es la suma de los* `n` *primeros primos truncables. Por ejemplo,*

```
sumaPrimosTruncables 10 == 249
```

Solución:

```
sumaPrimosTruncables :: Int -> Int
sumaPrimosTruncables n =
    sum (take n [x | x <- primos, primoTruncable x])
```

Ejercicio 7.12.3. *Calcular la suma de los 20 primos truncables.*

Solución: El cálculo es

```
ghci> sumaPrimosTruncables 20
2551
```

[1] http://projecteuler.net/problem=37

7.13. Primos permutables

Ejercicio 7.13.1. *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 :: Int -> Bool
```

tal que `(primoPermutable x)` *se verifica si* `x` *es un primo permutable. Por ejemplo,*

```
primoPermutable 17  ==  True
primoPermutable 19  ==  False
```

Solución:

```
primoPermutable :: Int -> Bool
primoPermutable x = and [primo y | y <- permutacionesN x]
```

donde `(permutacionesN x)` es la lista de los números obtenidos permutando los dígitos de `x`. Por ejemplo,

```
permutacionesN 325  ==  [325,235,253,352,532,523]
```

```
permutacionesN :: Int -> [Int]
permutacionesN x = [read ys | ys <- permutaciones (show x)]
```

Se han usado como auxiliares las funciones `permutaciones` (definida en la página 362) y `primo` (definida en la página 151).

7.14. Ordenación de los números enteros

Los números enteros se pueden ordenar como sigue: 0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5, -6, 6, -7, 7, ...

Ejercicio 7.14.1. *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]
```

Solución:

```
enteros :: [Int]
enteros = 0 : concat [[-x,x] | x <- [1..]]
```

Ejercicio 7.14.2. *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]
```

Solución:

```
enteros' :: [Int]
enteros' = iterate siguiente 0
    where siguiente x | x >= 0    = -x-1
                      | otherwise = -x
```

Ejercicio 7.14.3. *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
```

Solución:

```
posicion :: Int -> Int
posicion x = length (takeWhile (/=x) enteros)
```

Ejercicio 7.14.4. *Definir, por recursión, la función*

```
posicion1 :: Int -> Int
```

tal que `(posicion1 x)` *es la posición del entero* `x` *en la ordenación anterior. Por ejemplo,*

```
posicion1 2 == 4
```

Solución:

```
posicion1 :: Int -> Int
posicion1 x = aux enteros 0
 where aux (y:ys) n | x == y    = n
                    | otherwise = aux ys (n+1)
```

Ejercicio 7.14.5. *Definir, por comprension, la función*

```
posicion2 :: Int -> Int
```

tal que `(posicion2 x)` *es la posición del entero* `x` *en la ordenación anterior. Por ejemplo,*

```
posicion2 2 == 4
```

Solución:

```
posicion2 :: Int -> Int
posicion2 x = head [n | (n,y) <- zip [0..] enteros, y == x]
```

Ejercicio 7.14.6. *Definir, sin búsqueda, la función*

```
posicion3 :: Int -> Int
```

tal que `(posicion3 x)` *es la posición del entero* `x` *en la ordenación anterior. Por ejemplo,*

```
posicion3 2 == 4
```

Solución:

```
posicion3 :: Int -> Int
posicion3 x | x >= 0    = 2*x
            | otherwise = 2*(-x)-1
```

7.15. La sucesión de Hamming

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.

Ejercicio 7.15.1. *Definir la constante*

```
hamming :: [Int]
```

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]
```

Solución:

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]
```

donde se usan las siguientes funciones auxiliares

- `(mezcla3 xs ys zs)` es la lista obtenida mezclando las listas ordenadas `xs`, `ys` y `zs` y eliminando los elementos duplicados. Por ejemplo,

```
ghci> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
[2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

- `(mezcla2 xs ys zs)` es la lista obtenida mezclando las listas ordenadas `xs` e `ys` y eliminando los elementos duplicados. Por ejemplo,

```
ghci> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
[2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []       ys                   = ys
mezcla2 xs       []                   = xs
```

Ejercicio 7.15.2. *Definir la función*

```
divisoresEn :: Int -> [Int] -> 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
```

Solución:

```
divisoresEn :: Int -> [Int] -> 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 7.15.3. *Definir, usando* `divisoresEn`*, la constante*

```
hamming' :: [Int]
```

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]
```

Solución:

```
hamming' :: [Int]
hamming' = [x | x <- [1..], divisoresEn x [2,3,5]]
```

Ejercicio 7.15.4. *Definir la función*

```
cantidadHammingMenores :: Int -> 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
```

Solución:

```
cantidadHammingMenores :: Int -> Int
cantidadHammingMenores x = length (takeWhile (<x) hamming')
```

Ejercicio 7.15.5. *Definir la función*

```
siguienteHamming :: Int -> Int
```

tal que `(siguienteHamming x)` *es el menor número de la sucesión de Hamming mayor que* `x`*. Por ejemplo,*

```
siguienteHamming 6  ==  8
siguienteHamming 21  ==  24
```

Solución:

```
siguienteHamming :: Int -> Int
siguienteHamming x = head (dropWhile (<=x) hamming')
```

Ejercicio 7.15.6. *Definir la función*

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

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)
```

Solución:

```
huecoHamming :: Int -> [(Int,Int)]
huecoHamming n = [(x,y) | x <- hamming',
                          let y = siguienteHamming x,
                          y-x > n]
```

Ejercicio 7.15.7. *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`*.*

Solución: La propiedad es

```
prop_Hamming :: Int -> Bool
prop_Hamming n = huecoHamming n' /= []
                 where n' = abs n
```

La comprobación es

```
ghci> quickCheck prop_Hamming
OK, passed 100 tests.
```

7.16. Suma de los primos menores que *n*

Ejercicio 7.16.1 (Problema 10 del Proyecto Euler)**.** *Definir la función*

```
sumaPrimoMenores :: Int -> Int
```

tal que `(sumaPrimoMenores n)` *es la suma de los primos menores que* `n`*. Por ejemplo,*

```
sumaPrimoMenores 10 == 17
```

Solución:

```
sumaPrimoMenores :: Int -> Int
sumaPrimoMenores n = sumaMenores n primos 0
   where sumaMenores n (x:xs) a | n <= x    = a
                                | otherwise = sumaMenores n xs (a+x)
```

donde `primos` es la lista de los número primos (definida en la página 150).

7.17. Menor número triangular con más de n divisores

Ejercicio 7.17.1 (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

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \ldots$$

Los divisores de los primeros 7 números triangulares son:

$$\begin{array}{ll} 1: & 1 \\ 3: & 1, 3 \\ 6: & 1, 2, 3, 6 \\ 10: & 1, 2, 5, 10 \\ 15: & 1, 3, 5, 15 \\ 21: & 1, 3, 7, 21 \\ 28: & 1, 2, 4, 7, 14, 28 \end{array}$$

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
```

Solución:

```
euler12 :: Int -> Integer
euler12 n = head [x | x <- triangulares, nDivisores x > n]
```

dode se usan las siguientes funciones auxiliares

- `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)
  ```

7.18. Números primos consecutivos con dígitos con igual media

Ejercicio 7.18.1. *Definir la función*

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

tal que `(primosEquivalentes n)` *es la lista de las sucesiones de* `n` *números primos consecutivos con la media de sus dígitos iguales. Por ejemplo,*

```
take 2 (primosEquivalentes 2)  ==  [[523,541],[1069,1087]]
head (primosEquivalentes 3)    ==  [22193,22229,22247]
```

Solución:

```
primosEquivalentes :: Int -> [[Integer]]
primosEquivalentes n = aux primos
    where aux (x:xs) | relacionados equivalentes ys = ys : aux xs
                     | otherwise                    = aux xs
                     where ys = take n (x:xs)
```

donde `primos` está definido en la página 150, relacionados en la 122 y `equivalentes` en la 81.

7.19. Decisión de pertenencia al rango de una función creciente

Ejercicio 7.19.1. *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
```

Solución:

```
perteneceRango :: Int -> (Int -> Int) -> Bool
perteneceRango y f = y `elem` takeWhile (<=y) (imagenes f)
    where imagenes f = [f x | x <- [0..]]
```

7.20. Pares ordenados por posición

Ejercicio 7.20.1. *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)]
```

Solución:

```
paresOrdenados :: [a] -> [(a,a)]
paresOrdenados []     = []
paresOrdenados (x:xs) = [(x,y) | y <- xs] ++ paresOrdenados xs
```

Ejercicio 7.20.2. *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)]
```

Solución:

```
paresOrdenados2 :: [a] -> [(a,a)]
paresOrdenados2 [] = []
paresOrdenados2 (x:xs) =
    foldr (\y ac -> (x,y):ac) (paresOrdenados2 xs) xs
```

Ejercicio 7.20.3. *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)]
```

Solución:

```
paresOrdenados3 :: [a] -> [(a,a)]
paresOrdenados3 []     = []
paresOrdenados3 (x:xs) = zip (repeat x) xs ++ paresOrdenados3 xs
```

7.21. Aplicación iterada de una función

Ejercicio 7.21.1. *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
```

Solución:

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

Ejercicio 7.21.2. *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
```

Solución:

```
potenciaFunc2 :: Int -> (a -> a) -> a -> a
potenciaFunc2 n f x = last (take (n+1) (iterate f x))
```

7.22. Expresión de un número como suma de dos de una lista

Ejercicio 7.22.1. *Definir, por recursión, la función*

```
sumaDeDos :: Int -> [Int] -> Maybe (Int,Int)
```

tal que `(sumaDeDos x ys)` *decide si* `x` *puede expresarse como suma de dos elementos de* `ys` *y, en su caso, devuelve un par de elementos de* `ys` *cuya suma es* `x`*. Por ejemplo,*

```
sumaDeDos 9 [7,4,6,2,5]  ==  Just (7,2)
sumaDeDos 5 [7,4,6,2,5]  ==  Nothing
```

Solución:

```
sumaDeDos :: Int -> [Int] -> Maybe (Int,Int)
sumaDeDos _ []  =  Nothing
sumaDeDos _ [_] =  Nothing
sumaDeDos y (x:xs) | y-x `elem` xs = Just (x,y-x)
                   | otherwise     = sumaDeDos y xs
```

Ejercicio 7.22.2. *Definir, usando la función* `paresOrdenados` *(definida en la página 164), la función*

```
sumaDeDos' :: Int -> [Int] -> Maybe (Int,Int)
```

tal que `(sumaDeDos' x ys)` *decide si* `x` *puede expresarse como suma de dos elementos de* `ys` *y, en su caso, devuelve un par de elementos de* `ys` *cuya suma es* `x`*. Por ejemplo,*

```
sumaDeDos' 9 [7,4,6,2,5]  ==  Just (7,2)
sumaDeDos' 5 [7,4,6,2,5]  ==  Nothing
```

Solución:

```
sumaDeDos' :: Int -> [Int] -> Maybe (Int,Int)
sumaDeDos' x xs
    | null ys   = Nothing
    | otherwise = Just (head ys)
    where ys = [(a,b) | (a,b) <- paresOrdenados xs , a+b == x]
```

7.23. La bicicleta de Turing

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 $i = 2$, $d = 7$ y $n = 25$, entonces la lista con el número de eslabón que toca el radio doblado en cada vuelta es

$$[2, 9, 16, 23, 5, 12, 19, 1, 8, 15, 22, 4, 11, 18, 0, 7, 14, 21, 3, 10, 17, 24, 6, \ldots$$

Con lo que la cadena se rompe en la vuelta número 14.

Ejercicio 7.23.1. *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]
```

Solución:

```
eslabones :: Int -> Int -> Int -> [Int]
eslabones i d n = [(i+d*j) `mod` n | j <- [0..]]
```

Se puede definir usando `iterate`:

```
eslabones2 :: Int -> Int -> Int -> [Int]
eslabones2 i d n = map (\x-> mod x n) (iterate (+d) i)
```

Ejercicio 7.23.2. *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
```

Solución:

```
numeroVueltas :: Int -> Int -> Int -> Int
numeroVueltas i d n = length (takeWhile (/=0) (eslabones i d n))
```

7.24. Sucesión de Golomb

Esta seción está basada 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 esta sección se definirá una función para calcular los términos de la sucesión de Golomb.

Ejercicio 7.24.1. *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 siguiente ejercicio.*

Solución:

```
golomb :: Int -> Int
golomb n = sucGolomb !! (n-1)
```

Ejercicio 7.24.2. *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 siguiente ejercicio.*

Solución:

```
sucGolomb :: [Int]
sucGolomb = subSucGolomb 1
```

Ejercicio 7.24.3. *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 ejercicio anterior.*

Solución:

```
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)
```

Capítulo 8

Tipos definidos y de datos algebraicos

En este capítulo se presenta ejercicios sobre tipos definidos y tipos de datos algebraicos (TDA). Los ejercicios corresponden al tema 9 de [1].

Contenido

8.1 Puntos cercanos **172**

8.2 TDA de los números naturales **173**

8.2.1 Suma de números naturales 173

8.2.2 Producto de números naturales 173

8.3 TDA de árboles binarios con valores en los nodos y en las hojas **174**

8.3.1 Ocurrencia de un elemento en el árbol 174

8.4 TDA de árboles binarios con valores en las hojas **175**

8.4.1 Número de hojas 176

8.4.2 Carácter balanceado de un árbol 176

8.4.3 Árbol balanceado correspondiente a una lista 177

8.5 TDA de árboles binarios con valores en los nodos **177**

8.5.1 Número de hojas de un árbol 178

8.5.2 Número de nodos de un árbol 178

8.5.3 Profundidad de un árbol 179

8.5.4 Recorrido preorden de un árbol 179

8.5.5 Recorrido postorden de un árbol 180

8.5.6 Recorrido preorden de forma iterativa 180

8.5.7 Imagen especular de un árbol 181

8.5.8 Subárbol de profundidad dada 181

8.5.9 Árbol infinito generado con un elemento 182
8.5.10 Árbol de profundidad dada cuyos nodos son iguales a un elemento 183
8.5.11 Rama izquierda de un árbol 183
8.6 TAD de fórmulas proposicionales **184**
8.7 Modelización de un juego de cartas **189**
8.8 Evaluación de expresiones aritméticas **196**
8.9 Número de variables de una expresión aritmética **197**
8.10 Sustituciones en expresiones aritméticas **198**

8.1. Puntos cercanos

Ejercicio 8.1.1. *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* `\begin{sesion}` *y el segundo de* `qs`*, que son los más cercanos (es decir, no hay otro par* `(p',q')` *con* `p'` *en* `\begin{sesion}` *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))
```

Solución:

```
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)
```

8.2. TDA de los números naturales

En los diguientes ejercicios se usará el tipo algebraico de datos de los números naturales definido por

```
data Nat = Cero | Suc Nat
           deriving (Eq, Show)
```

8.2.1. Suma de números naturales

Ejercicio 8.2.1. *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))))
```

Solución:

```
suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

8.2.2. Producto de números naturales

Ejercicio 8.2.2. *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)))))
```

Solución:

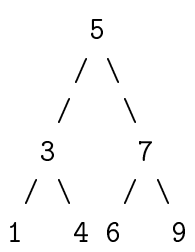
```
producto :: Nat -> Nat -> Nat
producto Cero _    = Cero
producto (Suc m) n = suma n (producto m n)
```

8.3. TDA de árboles binarios con valores en los nodos y en las hojas

En los siguientes ejercicios se trabajará con el tipo de datos algebraico de los árboles binarios definidos como sigue

```
data Arbol = Hoja Int
           | Nodo Arbol Int Arbol
           deriving (Show, Eq)
```

Por ejemplo, el árbol



se representa por

```
ejArbol :: Arbol
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))
```

8.3.1. Ocurrencia de un elemento en el árbol

Ejercicio 8.3.1. *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
```

Solución:

```
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 8.3.2. *En el preludio 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.

Solución:

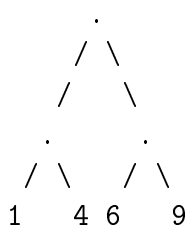
```
ocurre' :: Int -> Arbol -> Bool
ocurre' m (Hoja n)     = m == n
ocurre' m (Nodo i n d) = case compare m n of
                           LT -> ocurre' m i
                           EQ -> True
                           GT -> ocurre' m d
```

8.4. TDA de árboles binarios con valores en las hojas

En los siguientes ejercicios se trabajará con el tipo algebraico de dato de los árboles binarios con valores en las hojas definido por

```
data ArbolB = HojaB Int
            | NodoB ArbolB ArbolB
            deriving Show
```

Por ejemplo, el árbol



se representa por

```
ejArbolB :: ArbolB
ejArbolB = NodoB (NodoB (HojaB 1) (HojaB 4))
                 (NodoB (HojaB 6) (HojaB 9))
```

8.4.1. Número de hojas

Ejercicio 8.4.1. *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
```

Solución:

```
nHojas :: ArbolB -> Int
nHojas (HojaB _)     = 1
nHojas (NodoB a1 a2) = nHojas a1 + nHojas a2
```

8.4.2. Carácter balanceado de un árbol

Ejercicio 8.4.2. *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
```

Solución:

```
balanceado :: ArbolB -> Bool
balanceado (HojaB _)     = True
balanceado (NodoB a1 a2) = abs (nHojas a1 - nHojas a2) <= 1
                           && balanceado a1
                           && balanceado a2
```

8.4.3. Árbol balanceado correspondiente a una lista

Ejercicio 8.4.3. *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])
```

Solución:

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

Ejercicio 8.4.4. *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))
```

Solución:

```
arbolBalanceado :: [Int] -> ArbolB
arbolBalanceado [x] = HojaB x
arbolBalanceado xs  = NodoB (arbolBalanceado ys) (arbolBalanceado zs)
                      where (ys,zs) = mitades xs
```

8.5. TDA de árboles binarios con valores en los nodos

En los siguientes ejercicios se trabajará con el tipo algebraico de datos 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)
```

8.5.1. Número de hojas de un árbol

Ejercicio 8.5.1. *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
```

5

Solución:

```
nHojas :: Arbol a -> Int
nHojas Hoja         = 1
nHojas (Nodo x i d) = nHojas i + nHojas d
```

8.5.2. Número de nodos de un árbol

Ejercicio 8.5.2. *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
```

Solución:

```
nNodos :: Arbol a -> Int
nNodos Hoja         = 0
nNodos (Nodo x i d) = 1 + nNodos i + nNodos d
```

8.5.3. Profundidad de un árbol

Ejercicio 8.5.3. *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
```

Solución:

```
profundidad :: Arbol a -> Int
profundidad Hoja = 0
profundidad (Nodo x i d) = 1 + max (profundidad i) (profundidad d)
```

8.5.4. Recorrido preorden de un árbol

Ejercicio 8.5.4. *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]
```

Solución:

```
preorden :: Arbol a -> [a]
preorden Hoja         = []
preorden (Nodo x i d) = x : (preorden i ++ preorden d)
```

8.5.5. Recorrido postorden de un árbol

Ejercicio 8.5.5. *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]
```

Solución:

```
postorden :: Arbol a -> [a]
postorden Hoja         = []
postorden (Nodo x i d) = postorden i ++ postorden d ++ [x]
```

8.5.6. Recorrido preorden de forma iterativa

Ejercicio 8.5.6. *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.*

Solución:

```
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)
```

8.5.7. Imagen especular de un árbol

Ejercicio 8.5.7. *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))
```

Solución:

```
espejo :: Arbol a -> Arbol a
espejo Hoja         = Hoja
espejo (Nodo x i d) = Nodo x (espejo d) (espejo i)
```

8.5.8. Subárbol de profundidad dada

Ejercicio 8.5.8. *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)
```

Solución:

```
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)
```

8.5.9. Árbol infinito generado con un elemento

Ejercicio 8.5.9. *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,...
```

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)
```

Solución:

```
repeatArbol :: a -> Arbol a
repeatArbol x = Nodo x t t
                where t = repeatArbol x
```

8.5.10. Árbol de profundidad dada cuyos nodos son iguales a un elemento

Ejercicio 8.5.10. *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)
```

Solución:

```
replicateArbol :: Int -> a -> Arbol a
replicateArbol n = takeArbol n . repeatArbol
```

8.5.11. Rama izquierda de un árbol

Ejercicio 8.5.11. *Definir la función*

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

tal que `(ramaIzquierda a)` *es la lista de los valores de los nodos de la rama izquierda del árbol* `a`*. Por ejemplo,*

```
ramaIzquierda arbol  ==  [9,3,2]
```

Solución:

```
ramaIzquierda :: Arbol a -> [a]
ramaIzquierda Hoja         = []
ramaIzquierda (Nodo x i d) = x : ramaIzquierda i
```

8.6. TAD de fórmulas proposicionales

En el tema 9 de [1] se presenta un programa para decidir si una fórmula proposicional es tautología, que reproducimos a continuación.

Las fórmulas proposicionales se definen por:

- Las constantes booleanas son fórmulas proposicionales.
- Las fórmulas atómicas son fórmulas proposicionales.
- Si F es una fómula proposicional, entonces $\neg F$ también los es.
- Si F y G son fórmulas proposicionales, entonces $(F \wedge G)$ y $(F \to G)$ también lo son.

Las fórmulas se representan por el siguiente tipo de datos algebraico:

```
data Prop = Const Bool
          | Var Char
          | Neg Prop
          | Conj Prop Prop
          | Impl Prop Prop
          deriving Show
```

Por ejemplo, las fórmulas

- $p_1 := A \wedge \neg A$
- $p_2 := (A \wedge B) \to A$
- $p_3 := A \to (A \wedge B)$
- $p_4 := (A \to (A \to B)) \to B$

se representan por

```
p1, p2, p3, p4 :: Prop
p1 = Conj (Var 'A') (Neg (Var 'A'))
p2 = Impl (Conj (Var 'A') (Var 'B')) (Var 'A')
p3 = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
p4 = Impl (Conj (Var 'A') (Impl (Var 'A') (Var 'B'))) (Var 'B')
```

Las interpretaciones son listas formadas por el nombre de una variable proposicional y un valor de verdad. El tipo de las interpretaciones es `Interpretacion`

```
type Interpretacion = Asoc Char Bool
```

Las funciones del programa son

- `(valor i p)` es el valor de la proposición `p` en la interpretación `i`. Por ejemplo,

```
valor [('A',False),('B',True)] p3  =>  True
valor [('A',True),('B',False)] p3  =>  False
```

```
valor :: Interpretacion -> Prop -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q
```

- `(busca x ys)` es la segunda componente del primer par de ys cuya primera componente es igual a `x`.

```
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

- `(variables p)` es la lista de los nombres de las variables de la fórmula `p`. Por ejemplo,

```
variables p3 == "AAB"
```

```
variables :: Prop -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

- `(interpretacionesVar n)` es la lista de las interpretaciones con `n` variables. Por ejemplo,

```
ghci> interpretacionesVar 2
[[False,False],
 [False,True],
 [True,False],
 [True,True]]
```

```
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0     = [[]]
interpretacionesVar (n+1) = map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar n
```

- `(interpretaciones p)` es la lista de las interpretaciones de la fórmula `p`. Por ejemplo,

```
ghci> interpretaciones p3
[[('A',False),('B',False)],
 [('A',False),('B',True)],
 [('A',True),('B',False)],
 [('A',True),('B',True)]]
```

```
interpretaciones :: Prop -> [Interpretacion]
interpretaciones p =
    map (zip vs) (interpretacionesVar (length vs))
    where vs = nub (variables p)
```

Una definición alternativa es

```
interpretaciones' :: Prop -> [Interpretacion]
interpretaciones' p =
    [zip vs i | i <- is]
    where vs = nub (variables p)
          is = (interpretacionesVar (length vs))
```

- `(esTautologia p)` se verifica si la fórmula `p` es una tautología. Por ejemplo,

```
esTautologia p1  ==  False
esTautologia p2  ==  True
esTautologia p3  ==  False
esTautologia p4  ==  True
```

```
esTautologia :: Prop -> Bool
esTautologia p = and [valor i p | i <- interpretaciones p]
```

En esta sección se extiende el demostrador proposicional estudiado para incluir disyunciones y equivalencias.

Ejercicio 8.6.1. *Extender el procedimiento de decisión de tautologías para incluir las disyunciones (*`Disj`*) y las equivalencias (*`Equi`*). Por ejemplo,*

```
ghci> esTautologia (Equi (Var 'A') (Disj (Var 'A') (Var 'A')))
True
ghci> esTautologia (Equi (Var 'A') (Disj (Var 'A') (Var 'B')))
False
```

Solución:

```
data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Disj FProp FProp   -- Añadido
           | Impl FProp FProp
           | Equi FProp FProp   -- Añadido
           deriving Show

type Interpretacion = [(Char, Bool)]

valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Disj p q) = valor i p || valor i q    -- Añadido
valor i (Impl p q) = valor i p <= valor i q
valor i (Equi p q) = valor i p == valor i q    -- Añadido

busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']

variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Disj p q) = variables p ++ variables q   -- Añadido
variables (Impl p q) = variables p ++ variables q
variables (Equi p q) = variables p ++ variables q   -- Añadido
```

```
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0     = [[]]
interpretacionesVar (n+1) =
    map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar n

interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
    map (zip vs) (interpretacionesVar (length vs))
    where vs = nub (variables p)

esTautologia :: FProp -> Bool
esTautologia p =
    and [valor i p | i <- interpretaciones p]
```

Ejercicio 8.6.2. *Definir la función*

```
interpretacionesVar' :: Int -> [[Bool]]
```

que sea equivalente a `interpretacionesVar` *pero que en su definición use listas de comprensión en lugar de* `map`*. Por ejemplo,*

```
ghci> interpretacionesVar' 2
[[False,False],[False,True],[True,False],[True,True]]
```

Solución:

```
interpretacionesVar' :: Int -> [[Bool]]
interpretacionesVar' 0     = [[]]
interpretacionesVar' (n+1) =
    [False:bs | bs <- bss] ++ [True:bs | bs <- bss]
    where bss = interpretacionesVar' n
```

Ejercicio 8.6.3. *Definir la función*

```
interpretaciones' :: FProp -> [Interpretacion]
```

que sea equivalente a `interpretaciones` *pero que en su definición use listas de comprensión en lugar de* `map`*. Por ejemplo,*

```
ghci> interpretaciones' (Impl (Var 'A') (Conj (Var 'A') (Var 'B')))
[[('A',False),('B',False)],
 [('A',False),('B',True)],
 [('A',True),('B',False)],
 [('A',True),('B',True)]]
```

Solución:

```
interpretaciones' :: FProp -> [Interpretacion]
interpretaciones' p =
    [zip vs i | i <- is]
    where vs = nub (variables p)
          is = interpretacionesVar (length vs)
```

8.7. Modelización de un juego de cartas

En esta sección 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.

Nota. Se usan las siguientes librerías auxiliares

```
import Test.QuickCheck
import Data.Char
import Data.List
```

Ejercicio 8.7.1. *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`*.*

Solución:

```
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 8.7.2. *Definir el tipo de dato* `Color` *para representar los colores de las cartas: rojo y negro. Hacer que* `Color` *sea instancia de* `Show`*.*

Solución:

```
data Color = Rojo | Negro
             deriving Show
```

Ejercicio 8.7.3. *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.*

Solución:

```
color :: Palo -> Color
color Picas     = Negro
color Corazones = Rojo
color Diamantes = Rojo
color Treboles  = Negro
```

Ejercicio 8.7.4. *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`*. Por ejemplo,*

```
ghci :type Sota
Sota :: Valor
ghci :type Reina
Reina :: Valor
ghci :type Rey
Rey :: Valor
ghci :type As
As :: Valor
ghci :type Numerico 3
Numerico 3 :: Valor
```

Solución:

```
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 8.7.5. *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
```

Solución:

```
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 8.7.6. *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.*

Solución: La propiedad es

```
prop_MayorValor1 a b =
    mayor a b || mayor b a
```

La comprobación es

```
ghci 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

```
ghci quickCheck prop_MayorValor
OK, passed 100 tests.
```

Ejercicio 8.7.7. *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,*

```
ghci :type Carta Rey Corazones
Carta Rey Corazones :: Carta
ghci :type Carta (Numerico 4) Corazones
Carta (Numerico 4) Corazones :: Carta
```

Solución:

```
data Carta = Carta Valor Palo
             deriving (Eq, Show)
```

Ejercicio 8.7.8. *Definir la función*

```
valor :: Carta -> Valor
```

tal que `(valor c)` *es el valor de la carta* `c`*. Por ejemplo,*

```
valor (Carta Rey Corazones) == Rey
```

Solución:

```
valor :: Carta -> Valor
valor (Carta v p) = v
```

Ejercicio 8.7.9. *Definir la función*

```
palo :: Carta -> Valor
```

tal que `(palo c)` *es el palo de la carta* `c`*. Por ejemplo,*

```
palo (Carta Rey Corazones) == Corazones
```

Solución:

```
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 8.7.10. *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
```

Solución:

```
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 8.7.11. *Comprobar con QuickCheck si dadas dos cartas, una siempre gana a la otra.*

Solución: La propiedad es

```
prop_GanaCarta t c1 c2 =
    ganaCarta t c1 c2 || ganaCarta t c2 c1
```

La comprobación es

```
ghci 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 8.7.12. *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,*

```
ghci :type Agrega (Carta Rey Corazones) Vacia
Agrega (Carta Rey Corazones) Vacia :: Mano
```

Solución:

```
data Mano = Vacia | 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 []     = Vacia
               mano (c:cs) = Agrega c (mano cs)
           return (mano cs)
```

Ejercicio 8.7.13. *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) Vacia) (Carta Rey Corazones)
==>  True
ganaMano Picas (Agrega (Carta Sota Picas) Vacia) (Carta Rey Picas)
==>  False
```

Solución:

```
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 8.7.14. *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.*

Solución:

```
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 8.7.15. *Comprobar con QuickCheck que si una mano es ganadora, entonces la carta elegida es ganadora.*

Solución: La propiedad es

```
prop_eligeCartaGanaSiEsPosible triunfo c m =
    m /= Vacia ==>
    ganaMano triunfo m c == ganaCarta triunfo (eligeCarta triunfo c m) c
```

La comprobación es

```
ghci quickCheck prop_eligeCartaGanaSiEsPosible
Falsifiable, 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.

8.8. Evaluación de expresiones aritméticas

Ejercicio 8.8.1. *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
```

Solución:

```
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)
```

8.9. Número de variables de una expresión aritmética

Ejercicio 8.9.1. *La expresiones aritméticas con una variable (denotada por* `X`*) se pueden representar mediante el siguiente tipo*

```
data Expr = Num Int
          | Suma Expr Expr
          | X
```

Por ejemplo, la expresión $X + (13 + X)$ *se representa por* `Suma X (Suma (Num 13) X)`.

Definir la función

```
numVars :: Expr -> Int
```

tal que (`numVars e`) *es el número de variables en la expresión* `e`. *Por ejemplo,*

```
numVars (Num 3)                      ==  0
numVars X                            ==  1
numVars (Suma X (Suma (Num 13) X))   ==  2
```

Solución:

```
data Expr = Num Int
          | Suma Expr Expr
          | X

numVars :: Expr -> Int
numVars (Num n)    = 0
numVars (Suma a b) = numVars a + numVars b
numVars X          = 1
```

8.10. Sustituciones en expresiones aritméticas

Ejercicio 8.10.1. *La expresiones aritméticas se pueden representar mediante el siguiente tipo*

```
data Expr = V Char
          | N Int
          | S Expr Expr
          | P Expr Expr
          deriving Show
```

por ejemplo, la expresión $z(3+x)$ *se representa por* `(P (V 'z') (S (N 3) (V 'x')))`.

Definir la función

```
sustitucion :: Expr -> [(Char, Int)] -> Expr
```

tal que `(sustitucion e s)` *es la expresión obtenida sustituyendo las variables de la expresión* `e` *según se indica en la sustitución* `s`. *Por ejemplo,*

```
ghci> sustitucion (P (V 'z') (S (N 3) (V 'x'))) [('x',7),('z',9)]
P (N 9) (S (N 3) (N 7))
ghci> sustitucion (P (V 'z') (S (N 3) (V 'y'))) [('x',7),('z',9)]
P (N 9) (S (N 3) (V 'y'))
```

Solución:

```
data Expr = V Char
          | N Int
          | S Expr Expr
          | P Expr Expr
          deriving Show

sustitucion :: Expr -> [(Char, Int)] -> Expr
sustitucion e [] = e
sustitucion (V c) ((d,n):ps) | c == d = N n
                             | otherwise = sustitucion (V c) ps
sustitucion (N n) _ = N n
sustitucion (S e1 e2) ps = S (sustitucion e1 ps) (sustitucion e2 ps)
sustitucion (P e1 e2) ps = P (sustitucion e1 ps) (sustitucion e2 ps)
```

Capítulo 9

Demostración de propiedades por inducción

Contenido

9.1 Suma de los primeros números impares 199

9.2 Uno más la suma de potencias de dos 201

9.3 Copias de un elemento 203

En este capítulo se presenta ejercicios para demostrar propiedades de programas por inducción en los números naturales. Los ejercicios de este capítulo se corresponden con el tema 8 de [1].

Nota. Se usará librería `QuickCheck`

```
import Test.QuickCheck
```

9.1. Suma de los primeros números impares

Ejercicio 9.1.1. *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
```

Solución:

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

Ejercicio 9.1.2. *Definir, sin usar recursión, la función*

```
sumaImpares' :: Int -> Int
```

tal que `(sumaImpares' n)` *es la suma de los* `n` *primeros números impares. Por ejemplo,*

```
sumaImpares' 5 == 25
```

Solución:

```
sumaImpares' :: Int -> Int
sumaImpares' n = sum [1,3..(2*n-1)]
```

Ejercicio 9.1.3. *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.*

Solución: 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 9.1.4. *Definir la función*

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

tal que `(grafoSumaImpares m n)` *es la lista formadas por los números* `x` *entre* `m` *y* `n` *y los valores de* `(sumaImpares x)`.

Calcular `(grafoSumaImpares 1 9)`.

Solución: 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 9.1.5. *Demostrar por inducción que para todo* `n`*,* `(sumaImpares n)` *es igual a* n^2*.*

Solución: Por inducción en n.

Caso base: Hay que demostrar que

$$\texttt{sumaImpares}(0) = 0^2$$

En efecto,

$$\begin{array}{ll} & \texttt{sumaImpares}(0) \\ = & 0 \qquad\qquad [\text{por } \texttt{sumaImpares.1}] \\ = & 0^2 \qquad\qquad [\text{por aritmética}] \end{array}$$

Caso inductivo: Se supone la hipótesis de inducción (H.I.)

$$\texttt{sumaImpares}(n) = n^2$$

Hay que demostrar que

$$\texttt{sumaImpares}(n+1) = (n+1)^2$$

En efecto,

$$\begin{array}{ll} & \texttt{sumaImpares}(n+1) \\ = & (2*n+1) + \texttt{sumaImpares}(n) \qquad [\text{por } \texttt{sumaImpares.2}] \\ = & (2*n+1) + n^2 \qquad\qquad [\text{por H.I.}] \\ = & (n+1)^2 \qquad\qquad [\text{por álgebra}] \end{array}$$

9.2. Uno más la suma de potencias de dos

Ejercicio 9.2.1. *Definir, por recursión, la función*

```
sumaPotenciasDeDosMasUno :: Int -> Int
```

tal que `(sumaPotenciasDeDosMasUno n)` *es igual a* $1 + 2^0 + 2^1 + 2^2 + \cdots + 2^n$*. Por ejemplo,*

```
sumaPotenciasDeDosMasUno 3 == 16
```

Solución:

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

Ejercicio 9.2.2. *Definir por comprensión la función*

```
sumaPotenciasDeDosMasUno' :: Int -> Int
```

tal que $\texttt{sumaPotenciasDeDosMasUno}'n = 1 + 2^0 + 2^1 + 2^2 + \cdots + 2^n$. *Por ejemplo,*

```
sumaPotenciasDeDosMasUno' 3 == 16
```

Solución:

```
sumaPotenciasDeDosMasUno' :: Int -> Int
sumaPotenciasDeDosMasUno' n = 1 + sum [2^x | x <- [0..n]]
```

Ejercicio 9.2.3. *Demostrar por inducción que* $\texttt{sumaPotenciasDeDosMasUno}n = 2^{n+1}$.

Solución: Por inducción en n.

Caso base: Hay que demostrar que

$$\texttt{sumaPotenciasDeDosMasUno}(0) = 2^{0+1}$$

En efecto,

$$\begin{array}{ll} & \texttt{sumaPotenciasDeDosMasUno}(0) \\ = & 2 \qquad\qquad\qquad\qquad [\text{por } \texttt{sumaPotenciasDeDosMasUno}.1] \\ = & 2^{0+1} \qquad\qquad\qquad\qquad [\text{por aritmética}] \end{array}$$

Caso inductivo: Se supone la hipótesis de inducción (H.I.)

$$\texttt{sumaPotenciasDeDosMasUno}(n) = 2^{n+1}$$

Hay que demostrar que

$$\texttt{sumaPotenciasDeDosMasUno}(n+1) = 2^{(n+1)+1}$$

En efecto,

$$\begin{array}{ll} & \texttt{sumaPotenciasDeDosMasUno}(n+1) \\ = & 2^{n+1} + \texttt{sumaPotenciasDeDosMasUno}(n) \qquad [\text{por } \texttt{sumaPotenciasDeDosMasUno}.2] \\ = & 2^{n+1} + 2^{n+1} \qquad\qquad\qquad\qquad [\text{por H.I.}] \\ = & 2^{(n+1)+1} \qquad\qquad\qquad\qquad [\text{por aritmética}] \end{array}$$

9.3. Copias de un elemento

Ejercicio 9.3.1. *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]
```

Solución:

```
copia :: Int -> a -> [a]
copia 0 _  = []                 -- copia.1
copia n x = x : copia (n-1) x   -- copia.2
```

Ejercicio 9.3.2. *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
```

Solución:

```
todos :: (a -> Bool) -> [a] -> Bool
todos p []       = True                 -- todos.1
todos p (x : xs) = p x && todos p xs    -- todos.2
```

Ejercicio 9.3.3. *Comprobar con QuickCheck que todos los elementos de* `(copia n x)` *son iguales a* `x`*.*

Solución: 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 9.3.4. *Demostrar, por inducción en* `n`*, que todos los elementos de* `(copia n x)` *son iguales a* `x`*.*

Solución: Hay que demostrar que para todo n y todo x,

$$\texttt{todos}\ (==x)\ (\texttt{copia}\ n\ x)$$

Caso base: Hay que demostrar que

$$\texttt{todos}\ (==x)\ (\texttt{copia}\ 0\ x) = \mathit{True}$$

En efecto,

$$\begin{array}{ll}
\texttt{todos}\ (==x)\ (\texttt{copia}\ 0\ x) & \\
= \texttt{todos}\ (==x)\ [] & [\text{por } \texttt{copia}.1] \\
= \mathit{True} & [\text{por } \texttt{todos}.1]
\end{array}$$

Caso inductivo: Se supone la hipótesis de inducción (H.I.)

$$\texttt{todos}\ (==x)\ (\texttt{copia}\ n\ x) = \mathit{True}$$

Hay que demostrar que

$$\texttt{todos}\ (==x)\ (\texttt{copia}\ (n+1)\ x) = \mathit{True}$$

En efecto,

$$\begin{array}{ll}
\texttt{todos}\ (==x)\ (\texttt{copia}\ (n+1)\ x) & \\
= \texttt{todos}\ (==x)\ (x : \texttt{copia}\ n\ x) & [\text{por } \texttt{copia}.2] \\
= x == x\ \&\&\ \texttt{todos}\ (==x)\ (\texttt{copia}\ n\ x) & [\text{por } \texttt{todos}.2] \\
= \mathit{True}\ \&\&\ \texttt{todos}\ (==x)\ (\texttt{copia}\ n\ x) & [\text{por def. de } ==] \\
= \texttt{todos}\ (==x)\ (\texttt{copia}\ n\ x) & [\text{por def. de } \&\&] \\
= \mathit{True} & [\text{por H.I.}]
\end{array}$$

Ejercicio 9.3.5. *Definir por plegado 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
```

Solución: Se presentan 3 definiciones. La primera definición es

```
todos'_1 :: (a -> Bool) -> [a] -> Bool
todos'_1 p = foldr f True
             where f x y = p x && y
```

La segunda definición es

```
todos'_2 :: (a -> Bool) -> [a] -> Bool
todos'_2 p = foldr f True
             where f x y = (((&&) . p) x) y
```

La tercera definición es

```
todos' :: (a -> Bool) -> [a] -> Bool
todos' p = foldr ((&&) . p) True
```

Parte II

Tipos abstractos de datos y algorítmica

Capítulo 10

Polinomios

En este capítulo se proponen ejercicios con el tipo abstracto de datos (TAD) de los polinomios presentados en el tema 21 de [1]. Para hacerlo autocontenido se recuerda el TAD y sus implementaciones.

Contenido

10.1 El TAD de los polinomios **210**
10.1.1 Especificación del TAD de los polinomios 210
10.1.2 Los polinomios como tipo de dato algebraico 211
10.1.3 Los polinomios como listas dispersas 214
10.1.4 Los polinomios como listas densas 217
10.1.5 Comprobación de las implementaciones con QuickCheck . . . 220
10.2 Operaciones con polinomios **223**
10.2.1 Funciones sobre términos 224
10.2.2 Suma de polinomios 225
10.2.3 Producto de polinomios 226
10.2.4 El polinomio unidad 227
10.2.5 Resta de polinomios 228
10.2.6 Valor de un polinomio en un punto 228
10.2.7 Verificación de raices de polinomios 228
10.2.8 Derivación de polinomios 229
10.3 Ejercicios sobre polinomios **229**
10.3.1 Polinomio a partir de la representación dispersa 230
10.3.2 Polinomio a partir de la representación densa 230
10.3.3 Representación densa de un polinomio 231

10.3.4 Transformación de la representación densa a dispersa 231
10.3.5 Representación dispersa de un polinomio 231
10.3.6 Coeficiente del término de grado k 232
10.3.7 Lista de los coeficientes de un polinomio 232
10.3.8 Potencia de un polinomio 233
10.3.9 Integración de polinomios 234
10.3.10 Multiplicación de un polinomio por un número 235
10.3.11 División de polinomios 235
10.3.12 Divisibilidad de polinomios 236
10.4 La regla de Ruffini **238**
10.4.1 Divisores de un número 238
10.4.2 Término independiente de un polinomio 238
10.4.3 Paso de la regla de Ruffini 239
10.4.4 Cociente mediante la regla de Ruffini 239
10.4.5 Resto mediante la regla de Ruffini 240
10.4.6 Raíces mediante la regla de Ruffini 240
10.4.7 Factorización mediante la regla de Ruffini 241

10.1. El TAD de los polinomios

10.1.1. Especificación del TAD de los polinomios

La signatura del TAD de los polinomios es

```
polCero   :: Polinomio a
esPolCero :: Num a =>  Polinomio a -> Bool
consPol   :: Num a => Int -> a -> Polinomio a -> Polinomio a
grado     :: Polinomio a -> Int
coefLider :: Num a => Polinomio a -> a
restoPol  :: Polinomio a -> Polinomio a
```

donde el significado de las operaciones es

- `polCero` es el polinomio cero.
- `(esPolCero p)` se verifica si `p` es el polinomio cero.
- `(consPol n b p)` es el polinomio $bx^n + p$.

- `(grado p)` es el grado del polinomio `p`.
- `(coefLider p)` es el coeficiente líder del polinomio `p`.
- `(restoPol p)` es el resto del polinomio `p`.

La propiedades del TAD de los polinomios son

1. `polCero` es el polinomio cero.
2. Si n es mayor que el grado de p y b no es cero, entonces `(consPol n b p)` es un polinomio distinto del cero.
3. `(consPol (grado p) (coefLider p) (restoPol p))` es igual a `p`.
4. Si n es mayor que el grado de p y b no es cero, entonces el grado de `(consPol n b p)` es n.
5. Si n es mayor que el grado de p y b no es cero, entonces el coeficiente líder de `(consPol n b p)` es b.
6. Si n es mayor que el grado de p y b no es cero, entonces el resto de `(consPol n b p)` es `p`.

10.1.2. Los polinomios como tipo de dato algebraico

Los polinomios se pueden representar mediante los constructores `ConsPol` y `PolCero`. Por ejemplo, el polinomio $6x^4 - 5x^2 + 4x - 7$ se representa por

```
ConsPol 4 6 (ConsPol 2 (-5) (ConsPol 1 4 (ConsPol 0 (-7) PolCero)))
```

En el módulo `PolRepTDA` se implementa el TAD de los polinomios como tipos de dato algebraico. La cabecera del módulo es

```
module PolRepTDA
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
    consPol,   -- (Num a) => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num t => Polinomio t -> t
    restoPol   -- Polinomio t -> Polinomio t
  ) where
```

La definición del tipo de los polinomios es

```
data Polinomio a = PolCero
                 | ConsPol Int a (Polinomio a)
                 deriving Eq
```

Para facilitar la escritura de los polinomios se define

```
instance Num a => Show (Polinomio a) where
    show PolCero               = "0"
    show (ConsPol 0 b PolCero) = show b
    show (ConsPol 0 b p)       = concat [show b, " + ", show p]
    show (ConsPol 1 b PolCero) = concat [show b, "*x"]
    show (ConsPol 1 b p)       = concat [show b, "*x + ", show p]
    show (ConsPol n 1 PolCero) = concat ["x^", show n]
    show (ConsPol n b PolCero) = concat [show b, "*x^", show n]
    show (ConsPol n 1 p)       = concat ["x^", show n, " + ", show p]
    show (ConsPol n b p)       = concat [show b, "*x^", show n, " + ", show p]
```

Los siguientes ejemplos muestran polinomios con coeficientes enteros:

```
ejPol1, ejPol2, ejPol3:: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)
```

y los siguientes con coeficientes reales

```
ejPol5, ejPol6, ejPol7:: Polinomio Float
ejPol5 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol6 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol7 = consPol 1 2 (consPol 4 6 polCero)
```

Se puede comprobar la función de escritura como sigue:

```
ghci> ejPol1
3*x^4 + -5*x^2 + 3
ghci> ejPol2
x^5 + 5*x^2 + 4*x
ghci> ejPol3
6*x^4 + 2*x
ghci> ejPol5
```

```
3.0*x^4 + -5.0*x^2 + 3.0
ghci> ejPol6
x^5 + 5.0*x^2 + 4.0*x
ghci> ejPol7
6.0*x^4 + 2.0*x
```

La implementación de la especificación es la siguiente:

- `polCero` es el polinomio cero. Por ejemplo,

```
ghci> polCero
0
```

```
polCero :: Polinomio a
polCero = PolCero
```

- `(esPolCero p)` se verifica si `p` es el polinomio cero. Por ejemplo,

```
esPolCero polCero  ==  True
esPolCero ejPol1   ==  False
```

```
esPolCero :: Polinomio a -> Bool
esPolCero PolCero = True
esPolCero _       = False
```

- `(consPol n b p)` es el polinomio $bx^n + p$. Por ejemplo,

```
ejPol2               ==  x^5 + 5*x^2 + 4*x
consPol 3 0 ejPol2   ==  x^5 + 5*x^2 + 4*x
consPol 3 2 polCero  ==  2*x^3
consPol 6 7 ejPol2   ==  7*x^6 + x^5 + 5*x^2 + 4*x
consPol 4 7 ejPol2   ==  x^5 + 7*x^4 + 5*x^2 + 4*x
consPol 5 7 ejPol2   ==  8*x^5 + 5*x^2 + 4*x
```

```
consPol :: Num a => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b PolCero = ConsPol n b PolCero
consPol n b (ConsPol m c p)
    | n > m      = ConsPol n b (ConsPol m c p)
    | n < m      = ConsPol m c (consPol n b p)
    | b+c == 0   = p
    | otherwise  = ConsPol n (b+c) p
```

- `(grado p)` es el grado del polinomio `p`. Por ejemplo,

```
ejPol3        ==  6*x^4 + 2*x
grado ejPol3  ==  4
```

```
grado :: Polinomio a -> Int
grado PolCero         = 0
grado (ConsPol n _ _) = n
```

- `(coefLider p)` es el coeficiente líder del polinomio `p`. Por ejemplo,

```
ejPol3            ==  6*x^4 + 2*x
coefLider ejPol3  ==  6
```

```
coefLider:: Num t => Polinomio t -> t
coefLider PolCero         = 0
coefLider (ConsPol _ b _) = b
```

- `(restoPol p)` es el resto del polinomio `p`. Por ejemplo,

```
ejPol3           ==  6*x^4 + 2*x
restoPol ejPol3  ==  2*x
ejPol2           ==  x^5 + 5*x^2 + 4*x
restoPol ejPol2  ==  5*x^2 + 4*x
```

```
restoPol :: Polinomio t -> Polinomio t
restoPol PolCero         = PolCero
restoPol (ConsPol _ _ p) = p
```

10.1.3. Los polinomios como listas dispersas

Los polinomios se pueden representar mediante la lista de sus coeficientes ordenados en orden decreciente según el grado. Por ejemplo, el polinomio $6x^4 - 5x^2 + 4x - 7$ se representa por `[6,0,-2,4,-7]`. Dicha representación se llama listas dispersas.

En el módulo `PolRepDispersa` se implementa el TAD de los polinomios como listas dispersas. La cabecera del módulo es

```
module PolRepDispersa
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
```

```
    consPol,   -- (Num a) => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- Polinomio a -> Polinomio a
  ) where
```

La definición del tipo de los polinomios es

```
data Polinomio a = Pol [a]
                   deriving Eq
```

Para facilitar la escritura de los polinomios se define

```
instance Num a => Show (Polinomio a) where
  show pol
      | esPolCero pol         = "0"
      | n == 0 && esPolCero p = show a
      | n == 0                = concat [show a, " + ", show p]
      | n == 1 && esPolCero p = concat [show a, "*x"]
      | n == 1                = concat [show a, "*x + ", show p]
      | a == 1 && esPolCero p = concat ["x^", show n]
      | esPolCero p           = concat [show a, "*x^", show n]
      | a == 1                = concat ["x^", show n, " + ", show p]
      | otherwise             = concat [show a, "*x^", show n, " + ", show p]
     where n = grado pol
           a = coefLider pol
           p = restoPol pol
```

Los siguientes ejemplos muestran polinomios con coeficientes enteros:

```
ejPol1, ejPol2, ejPol3:: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)
```

Se puede comprobar la función de escritura como sigue:

```
ghci> ejPol1
3*x^4 + -5*x^2 + 3
ghci> ejPol2
x^5 + 5*x^2 + 4*x
ghci> ejPol3
6*x^4 + 2*x
```

La implementación de la especificación es la siguiente:

- `polCero` es el polinomio cero. Por ejemplo,

```
ghci> polCero
0
```

```
polCero :: Polinomio a
polCero = Pol []
```

- `(esPolCero p)` se verifica si `p` es el polinomio cero. Por ejemplo,

```
esPolCero polCero  ==  True
esPolCero ejPol1   ==  False
```

```
esPolCero :: Polinomio a -> Bool
esPolCero (Pol []) = True
esPolCero _        = False
```

- `(consPol n b p)` es el polinomio $bx^n + p$. Por ejemplo,

```
ejPol2               ==  x^5 + 5*x^2 + 4*x
consPol 3 0 ejPol2   ==  x^5 + 5*x^2 + 4*x
consPol 3 2 polCero  ==  2*x^3
consPol 6 7 ejPol2   ==  7*x^6 + x^5 + 5*x^2 + 4*x
consPol 4 7 ejPol2   ==  x^5 + 7*x^4 + 5*x^2 + 4*x
consPol 5 7 ejPol2   ==  8*x^5 + 5*x^2 + 4*x
```

```
consPol :: Num a => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b p@(Pol xs)
    | esPolCero p = Pol (b:replicate n 0)
    | n > m       = Pol (b:(replicate (n-m-1) 0)++xs)
    | n < m       = consPol m c (consPol n b (restoPol p))
    | b+c == 0    = Pol (dropWhile (==0) (tail xs))
    | otherwise   = Pol ((b+c):tail xs)
    where
      c = coefLider p
      m = grado p
```

- `(grado p)` es el grado del polinomio `p`. Por ejemplo,

```
ejPol3        ==  6*x^4 + 2*x
grado ejPol3  ==  4
```

```
grado:: Polinomio a -> Int
grado (Pol []) = 0
grado (Pol xs) = length xs - 1
```

- `(coefLider p)` es el coeficiente líder del polinomio `p`. Por ejemplo,

```
ejPol3            ==  6*x^4 + 2*x
coefLider ejPol3  ==  6
```

```
coefLider:: Num t => Polinomio t -> t
coefLider (Pol [])    = 0
coefLider (Pol (a:_)) = a
```

- `(restoPol p)` es el resto del polinomio `p`. Por ejemplo,

```
ejPol3           ==  6*x^4 + 2*x
restoPol ejPol3  ==  2*x
ejPol2           ==  x^5 + 5*x^2 + 4*x
restoPol ejPol2  ==  5*x^2 + 4*x
```

```
restoPol :: Num t => Polinomio t -> Polinomio t
restoPol (Pol [])     = polCero
restoPol (Pol [_])    = polCero
restoPol (Pol (_:b:as))
    | b == 0    = Pol (dropWhile (==0) as)
    | otherwise = Pol (b:as)
```

10.1.4. Los polinomios como listas densas

Los polinomios se pueden representar mediante la lista de pares `(grado,coef)`, ordenados en orden decreciente según el grado. Por ejemplo, el polinomio $6x^4 - 5x^2 + 4x - 7$ se representa por `[(4,6),(2,-5),(1,4),(0,-7)]`. Dicha representación se llama listas densas.

En el módulo `PolRepDensa` se implementa el TAD de los polinomios como listas densas. La cabecera del módulo es

```
module PolRepDensa
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
    consPol,   -- Num a => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- Polinomio a -> Polinomio a
  ) where
```

La definición del tipo de los polinomios es

```
data Polinomio a = Pol [(Int,a)]
                   deriving Eq
```

Para facilitar la escritura de los polinomios se define

```
instance Num t => Show (Polinomio t) where
  show pol
      | esPolCero pol         = "0"
      | n == 0 && esPolCero p = show a
      | n == 0                = concat [show a, " + ", show p]
      | n == 1 && esPolCero p = concat [show a, "*x"]
      | n == 1                = concat [show a, "*x + ", show p]
      | a == 1 && esPolCero p = concat ["x^", show n]
      | esPolCero p           = concat [show a, "*x^", show n]
      | a == 1                = concat ["x^", show n, " + ", show p]
      | otherwise             = concat [show a, "*x^", show n, " + ", show p]
     where n = grado pol
           a = coefLider pol
           p = restoPol pol
```

Los siguientes ejemplos muestran polinomios con coeficientes enteros:

```
ejPol1, ejPol2, ejPol3:: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)
```

Se puede comprobar la función de escritura como sigue:

```
ghci> ejPol1
3*x^4 + -5*x^2 + 3
ghci> ejPol2
x^5 + 5*x^2 + 4*x
ghci> ejPol3
6*x^4 + 2*x
```

La implementación de la especificación es la siguiente:

- `polCero` es el polinomio cero. Por ejemplo,

```
ghci> polCero
0
```

```
polCero :: Num a => Polinomio a
polCero = Pol []
```

- `(esPolCero p)` se verifica si `p` es el polinomio cero. Por ejemplo,

```
esPolCero polCero  ==  True
esPolCero ejPol1   ==  False
```

```
esPolCero :: Num a => Polinomio a -> Bool
esPolCero (Pol []) = True
esPolCero _        = False
```

- `(consPol n b p)` es el polinomio $bx^n + p$. Por ejemplo,

```
ejPol2              ==  x^5 + 5*x^2 + 4*x
consPol 3 0 ejPol2  ==  x^5 + 5*x^2 + 4*x
consPol 3 2 polCero ==  2*x^3
consPol 6 7 ejPol2  ==  7*x^6 + x^5 + 5*x^2 + 4*x
consPol 4 7 ejPol2  ==  x^5 + 7*x^4 + 5*x^2 + 4*x
consPol 5 7 ejPol2  ==  8*x^5 + 5*x^2 + 4*x
```

```
consPol :: Num a => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b p@(Pol xs)
    | esPolCero p = Pol [(n,b)]
    | n > m       = Pol ((n,b):xs)
    | n < m       = consPol m c (consPol n b (Pol (tail xs)))
    | b+c == 0    = Pol (tail xs)
```

```
    | otherwise   = Pol ((n,b+c):(tail xs))
    where
      c = coefLider p
      m = grado p
```

- `(grado p)` es el grado del polinomio `p`. Por ejemplo,

```
ejPol3        ==  6*x^4 + 2*x
grado ejPol3  ==  4
```

```
grado:: Polinomio a -> Int
grado (Pol [])        = 0
grado (Pol ((n,_):_)) = n
```

- `(coefLider p)` es el coeficiente líder del polinomio `p`. Por ejemplo,

```
ejPol3            ==  6*x^4 + 2*x
coefLider ejPol3  ==  6
```

```
coefLider:: Num t => Polinomio t -> t
coefLider (Pol [])        = 0
coefLider (Pol ((_,b):_)) = b
```

- `(restoPol p)` es el resto del polinomio `p`. Por ejemplo,

```
ejPol3           ==  6*x^4 + 2*x
restoPol ejPol3  ==  2*x
ejPol2           ==  x^5 + 5*x^2 + 4*x
restoPol ejPol2  ==  5*x^2 + 4*x
```

```
restoPol :: Num t => Polinomio t -> Polinomio t
restoPol (Pol [])     = polCero
restoPol (Pol [_])    = polCero
restoPol (Pol (_:xs)) = Pol xs
```

10.1.5. Comprobación de las implementaciones con QuickCheck

En el módulo `polPropiedades` se comprueba con QuickCheck que las 3 implementaciones del TAD de los polinomios cumplen las propiedades de la especificación. Para ello, se define un generador de polinomios. Este generador se usará en las siguientes secciones para verificar propiedades de las operaciones con polinomios.

La cabecera del módulo es

```
{-# LANGUAGE FlexibleInstances #-}

module PolPropiedades where

import PolRepTDA
-- import PolRepDispersa
-- import PolRepDensa

import Test.QuickCheck
```

Nótese que hay que elegir (descomentando) una implementación del TAD de polinomios. Nosotros hemos elegido la primera.

Para la generación arbitraria de polinomios se define la función

```
genPol :: Int -> Gen (Polinomio Int)
```

tal que `(genPol n)` es un generador de polinomios. Por ejemplo,

```
ghci> sample (genPol 1)
7*x^9 + 9*x^8 + 10*x^7 + -14*x^5 + -15*x^2 + -10
-4*x^8 + 2*x
-8*x^9 + 4*x^8 + 2*x^6 + 4*x^5 + -6*x^4 + 5*x^2 + -8*x
-9*x^9 + x^5 + -7
8*x^10 + -9*x^7 + 7*x^6 + 9*x^5 + 10*x^3 + -1*x^2
7*x^10 + 5*x^9 + -5
-8*x^10 + -7
-5*x
5*x^10 + 4*x^4 + -3
3*x^3 + -4
10*x
```

```
genPol :: Int -> Gen (Polinomio Int)
genPol 0 = return polCero
genPol n = do n <- choose (0,10)
              b <- choose (-10,10)
              p <- genPol (div n 2)
              return (consPol n b p)
```

y se declara los polinomios como una clase arbitraria

```
instance Arbitrary (Polinomio Int) where
    arbitrary = sized genPol
```

La formalización de las propiedades del TAD de los polinomios es

1. `polCero` es el polinomio cero.

```
prop_polCero_es_cero :: Bool
prop_polCero_es_cero =
    esPolCero polCero
```

2. Si n es mayor que el grado de p y b no es cero, entonces `(consPol n b p)` es un polinomio distinto del cero.

```
prop_consPol_no_cero :: Int -> Int -> Polinomio Int -> Property
prop_consPol_no_cero n b p =
    n > grado p && b /= 0  ==>
      not (esPolCero (consPol n b p))
```

3. `(consPol (grado p) (coefLider p) (restoPol p))` es igual a `p`.

```
prop_consPol :: Polinomio Int -> Bool
prop_consPol p =
    consPol (grado p) (coefLider p) (restoPol p) == p
```

4. Si n es mayor que el grado de p y b no es cero, entonces el grado de `(consPol n b p)` es n.

```
prop_grado :: Int -> Int -> Polinomio Int -> Property
prop_grado n b p =
    n > grado p && b /= 0 ==>
      grado (consPol n b p) == n
```

5. Si n es mayor que el grado de p y b no es cero, entonces el coeficiente líder de `(consPol n b p)` es b.

```
prop_coefLider :: Int -> Int -> Polinomio Int -> Property
prop_coefLider n b p =
    n > grado p && b /= 0 ==>
      coefLider (consPol n b p) == b
```

6. Si n es mayor que el grado de p y b no es cero, entonces el resto de `(consPol n b p)` es `p`.

```
prop_restoPol :: Int -> Int -> Polinomio Int -> Property
prop_restoPol n b p =
    n > grado p && b /= 0 ==>
      restoPol (consPol n b p) == p
```

La comprobación de las propiedades es

```
ghci> quickCheck prop_polCero_es_cero
+++ OK, passed 100 tests.

ghci> quickCheck prop_consPol_no_cero
+++ OK, passed 100 tests.

ghci> quickCheck prop_consPol
+++ OK, passed 100 tests.

ghci> quickCheck prop_grado
+++ OK, passed 100 tests.

ghci> quickCheck prop_coefLider
+++ OK, passed 100 tests.

ghci> quickCheck prop_restoPol
+++ OK, passed 100 tests.
```

10.2. Operaciones con polinomios

En el módulo `PolOperaciones` se definen, usando el TAD de los polinomios, las operaciones básicas con polinomios. La cabecera del módulo es

```
module PolOperaciones (module Pol, module PolOperaciones) where

-- import PolRepTDA as Pol
-- import PolRepDispersa as Pol
import PolRepDensa as Pol

import Test.QuickCheck
```

Nótese que hay que elegir (descomentándola) una implementación del TAD de los polinomios. Nosotros hemos elegido la tercera.

En esta sección usaremos los siguientes ejemplos de polinomios:

```
ejPol1, ejPol2, ejPol3, ejTerm:: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)
ejTerm = consPol 1 4 polCero
```

Usamos el generador definido en la sección anterior

```
genPol :: (Arbitrary a, Num a) => Int -> Gen (Polinomio a)
genPol 0 = return polCero
genPol n = do n <- choose (0,10)
              b <- arbitrary
              p <- genPol (div n 2)
              return (consPol n b p)

instance (Arbitrary a, Num a) => Arbitrary (Polinomio a) where
    arbitrary = sized genPol
```

10.2.1. Funciones sobre términos

- `(creaTermino n a)` es el término $a * x^n$. Por ejemplo,

```
creaTermino 2 5 == 5*x^2
```

```
creaTermino:: Num t => Int -> t -> Polinomio t
creaTermino n a = consPol n a polCero
```

- `(termLider p)` es el término líder del polinomio `p`. Por ejemplo,

```
ejPol2            ==  x^5 + 5*x^2 + 4*x
termLider ejPol2  ==  x^5
```

```
termLider:: Num t => Polinomio t -> Polinomio t
termLider p = creaTermino (grado p) (coefLider p)
```

10.2.2. Suma de polinomios

- `(sumaPol p q)` es la suma de los polinomios `p` y `q`. Por ejemplo,

```
ejPol1                 ==  3*x^4 + -5*x^2 + 3
ejPol2                 ==  x^5 + 5*x^2 + 4*x
sumaPol ejPol1 ejPol2  ==  x^5 + 3*x^4 + 4*x + 3
```

```
sumaPol:: Num a => Polinomio a -> Polinomio a -> Polinomio a
sumaPol p q
    | esPolCero p = q
    | esPolCero q = p
    | n1 > n2      = consPol n1 a1 (sumaPol r1 q)
    | n1 < n2      = consPol n2 a2 (sumaPol p r2)
    | a1+a2 /= 0   = consPol n1 (a1+a2) (sumaPol r1 r2)
    | otherwise    = sumaPol r1 r2
    where n1 = grado p
          a1 = coefLider p
          r1 = restoPol p
          n2 = grado q
          a2 = coefLider q
          r2 = restoPol q
```

La suma verifica las siguientes propiedades:

1. El polinomio cero es el elemento neutro de la suma.

```
prop_neutroSumaPol :: Polinomio Int -> Bool
prop_neutroSumaPol p =
    sumaPol polCero p == p
```

   Su comprobación es

```
ghci> quickCheck prop_neutroSumaPol
OK, passed 100 tests.
```

2. La suma es conmutativa.

```
prop_conmutativaSuma :: Polinomio Int -> Polinomio Int -> Bool
prop_conmutativaSuma p q =
    sumaPol p q == sumaPol q p
```

   Su comprobación es

```
ghci> quickCheck prop_conmutativaSuma
OK, passed 100 tests.
```

10.2.3. Producto de polinomios

- `(multPorTerm t p)` es el producto del término `t` por el polinomio `p`. Por ejemplo,

```
ejTerm                    ==  4*x
ejPol2                    ==  x^5 + 5*x^2 + 4*x
multPorTerm ejTerm ejPol2 ==  4*x^6 + 20*x^3 + 16*x^2
```

```
multPorTerm :: Num t => Polinomio t -> Polinomio t -> Polinomio t
multPorTerm term pol
    | esPolCero pol = polCero
    | otherwise     = consPol (n+m) (a*b) (multPorTerm term r)
    where n = grado term
          a = coefLider term
          m = grado pol
          b = coefLider pol
          r = restoPol pol
```

- `(multPol p q)` es el producto de los polinomios `p` y `q`. Por ejemplo,

```
ghci> ejPol1
3*x^4 + -5*x^2 + 3
ghci> ejPol2
x^5 + 5*x^2 + 4*x
ghci> multPol ejPol1 ejPol2
3*x^9 + -5*x^7 + 15*x^6 + 15*x^5 + -25*x^4 + -20*x^3 + 15*x^2 + 12*x
```

```
multPol :: Num a => Polinomio a -> Polinomio a -> Polinomio a
multPol p q
    | esPolCero p = polCero
    | otherwise    = sumaPol (multPorTerm (termLider p) q)
                             (multPol (restoPol p) q)
```

El producto de polinomios verifica las siguientes propiedades

1. El producto de polinomios es conmutativo.

```
prop_conmutativaProducto :: Polinomio Int -> Polinomio Int -> Bool
prop_conmutativaProducto p q =
    multPol p q == multPol q p
```

La comprobación es

```
ghci> quickCheck prop_conmutativaProducto
OK, passed 100 tests.
```

2. El producto es distributivo respecto de la suma.

```
prop_distributivaProductoSuma :: Polinomio Int -> Polinomio Int
                                 -> Polinomio Int -> Bool
prop_distributivaProductoSuma p q r =
    multPol p (sumaPol q r) == sumaPol (multPol p q) (multPol p r)
```

La comprobación es:

```
ghci> quickCheck prop_distributivaProductoSuma
OK, passed 100 tests.
```

10.2.4. El polinomio unidad

- `polUnidad` es el polinomio unidad. Por ejemplo,

```
ghci> polUnidad
1
```

```
polUnidad:: Num t => Polinomio t
polUnidad = consPol 0 1 polCero
```

El polinomio unidad es el elemento neutro del producto.

```
prop_polUnidad :: Polinomio Int -> Bool
prop_polUnidad p =
    multPol p polUnidad == p
```

La comprobación es:

```
ghci> quickCheck prop_polUnidad
OK, passed 100 tests.
```

10.2.5. Resta de polinomios

- `(resta p q)` es la el polinomio obtenido restándole a `p` el `q`. Por ejemplo,

```
ejPol1                  ==  3*x^4 + -5*x^2 + 3
ejPol2                  ==  x^5 + 5*x^2 + 4*x
restaPol ejPol1 ejPol2  ==  -1*x^5 + 3*x^4 + -10*x^2 + -4*x + 3
```

```
restaPol :: (Num a) => Polinomio a -> Polinomio a -> Polinomio a
restaPol p q  =
    sumaPol p (multPorTerm (creaTermino 0 (-1)) q)
```

10.2.6. Valor de un polinomio en un punto

- `(valor p c)` es el valor del polinomio `p` al sustituir su variable por `c`. Por ejemplo,

```
ejPol1             ==  3*x^4 + -5*x^2 + 3
valor ejPol1 0     ==  3
valor ejPol1 1     ==  1
valor ejPol1 (-2)  ==  31
```

```
valor:: Num a => Polinomio a -> a -> a
valor p c
    | esPolCero p = 0
    | otherwise   =  b*c^n + valor r c
    where n = grado p
          b = coefLider p
          r = restoPol p
```

10.2.7. Verificación de raices de polinomios

- `(esRaiz c p)` se verifica si `c` es una raíz del polinomio `p`. Por ejemplo,

```
ejPol3           ==  6*x^4 + 2*x
esRaiz 1 ejPol3  ==  False
esRaiz 0 ejPol3  ==  True
```

```
esRaiz:: Num a => a -> Polinomio a -> Bool
esRaiz c p = valor p c == 0
```

10.2.8. Derivación de polinomios

- `(derivada p)` es la derivada del polinomio `p`. Por ejemplo,

```
ejPol2           ==  x^5 + 5*x^2 + 4*x
derivada ejPol2  ==  5*x^4 + 10*x + 4
```

```
derivada :: Polinomio Int -> Polinomio Int
derivada p
    | n == 0     = polCero
    | otherwise  = consPol (n-1) (n*b) (derivada r)
    where n = grado p
          b = coefLider p
          r = restoPol p
```

La derivada de la suma es la suma de las derivadas.

```
prop_derivada :: Polinomio Int -> Polinomio Int -> Bool
prop_derivada p q =
    derivada (sumaPol p q) == sumaPol (derivada p) (derivada q)
```

La comprobación es

```
ghci> quickCheck prop_derivada
OK, passed 100 tests.
```

10.3. Ejercicios sobre polinomios

El objetivo de esta sección es ampliar el conjunto de operaciones sobre polinomios definidas utilizando las implementaciones del TAD de polinomio. Además, en algunos ejemplos de usan polinomios con coeficientes racionales. En Haskell, el número racional $\frac{x}{y}$ se representa por `x%y`. El TAD de los números racionales está definido en el módulo `Data.Ratio`.

Nota. Se usarán las siguientes librerías

```
import PolOperaciones
import Test.QuickCheck
import Data.Ratio
```

10.3.1. Polinomio a partir de la representación dispersa

Ejercicio 10.3.1.1. *Definir la función*

```
creaPolDispersa :: Num 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
```

Solución:

```
creaPolDispersa :: Num a => [a] -> Polinomio a
creaPolDispersa []     = polCero
creaPolDispersa (x:xs) = consPol (length xs) x (creaPolDispersa xs)
```

10.3.2. Polinomio a partir de la representación densa

Ejercicio 10.3.2.1. *Definir la función*

```
creaPolDensa :: Num 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
```

Solución:

```
creaPolDensa :: Num 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 => 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)]
```

10.3.3. Representación densa de un polinomio

Ejercicio 10.3.3.1. *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)]
```

Solución:

```
densa :: Num a => Polinomio a -> [(Int,a)]
densa p | esPolCero p = []
        | otherwise   = (grado p, coefLider p) : densa (restoPol p)
```

10.3.4. Transformación de la representación densa a dispersa

Ejercicio 10.3.4.1. *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]
```

Solución:

```
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)
```

10.3.5. Representación dispersa de un polinomio

Ejercicio 10.3.5.1. *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]
```

Solución:

```
dispersa :: Num a => Polinomio a -> [a]
dispersa = densaAdispersa . densa
```

10.3.6. Coeficiente del término de grado *k*

Ejercicio 10.3.6.1. *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,*

```
pol1               ==  x^5 + 5*x^2 + 4*x
coeficiente 2 pol1 ==  5
coeficiente 3 pol1 ==  0
```

Solución:

```
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])
```

10.3.7. Lista de los coeficientes de un polinomio

Ejercicio 10.3.7.1. *Definir la función*

```
coeficientes :: Num a => Polinomio a -> [a]
```

tal que `(coeficientes p)` *es la lista de los coeficientes del polinomio* `p`*. Por ejemplo,*

```
pol1              ==  x^5 + 5*x^2 + 4*x
coeficientes pol1 ==  [1,0,0,5,4,0]
```

Solución:

```
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 10.3.7.2. *Comprobar con QuickCheck que, dado un polinomio* `p`*, el polinomio obtenido mediante* `creaPolDispersa` *a partir de la lista de coeficientes de* `p` *coincide con* `p`*.*

Solución: La propiedad es

```
prop_coef :: Polinomio Int -> Bool
prop_coef p =
    creaPolDispersa (coeficientes p) == p
```

La comprobación es

```
ghci> quickCheck prop_coef
+++ OK, passed 100 tests.
```

10.3.8. Potencia de un polinomio

Ejercicio 10.3.8.1. *Definir la función*

```
potencia :: Num 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
```

Solución:

```
potencia :: Num a => Polinomio a -> Int -> Polinomio a
potencia p 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))
```

Ejercicio 10.3.8.2. *Mejorar la definición de potencia definiendo la función*

```
potenciaM :: Num 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* $x^n = (x^2)^{\frac{n}{2}}$

- *Si n es impar, entonces* $x^n = x \times (x^2)^{\frac{n-1}{2}}$.

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
```

Solución:

```
potenciaM :: Num 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))
```

10.3.9. Integración de polinomios

Ejercicio 10.3.9.1. *Definir la función*

```
integral :: Fractional a => Polinomio a -> Polinomio a
```

tal que (`integral` *p) es la integral del polinomio* `p` *cuyos coefientes 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
```

Solución:

```
integral :: Fractional 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 10.3.9.2. *Definir la función*

```
integralDef :: Fractional t => Polinomio t -> t -> t -> t
```

tal que `(integralDef p a b)` *es la integral definida del polinomio* `p`*, cuyos coefientes son números racionales, entre* `a` *y* `b`*. Por ejemplo,*

```
ghci> integralDef pol3 0 1
2.9166666666666667
ghci> integralDef pol3 0 1 :: Rational
35 % 12
```

Solución:

```
integralDef :: Fractional t => Polinomio t -> t -> t -> t
integralDef p a b = (valor q b) - (valor q a)
    where q = integral p
```

10.3.10. Multiplicación de un polinomio por un número

Ejercicio 10.3.10.1. *Definir la función*

```
multEscalar :: Num 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
```

Solución:

```
multEscalar :: Num 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
```

10.3.11. División de polinomios

Ejercicio 10.3.11.1. *Definir la función*

```
cociente:: Fractional 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
```

Solución:

```
cociente :: Fractional 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 10.3.11.2. *Definir la función*

```
resto :: Fractional 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
```

Solución:

```
resto :: Fractional a => Polinomio a -> Polinomio a -> Polinomio a
resto p q = restaPol p (multPol (cociente p q) q)
```

10.3.12. Divisibilidad de polinomios

Ejercicio 10.3.12.1. *Definir la función*

```
divisiblePol :: Fractional 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
```

Solución:

```
divisiblePol :: Fractional a => Polinomio a -> Polinomio a -> Bool
divisiblePol p q = esPolCero (resto p q)
```

Ejercicio 10.3.12.2. *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* $ax^5 + b * x^4 + c * x^3 + d * x^2 + e * x + f$ *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
```

Solución:

```
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 defininición equivalente por plegado es

```
horner' :: Num a => Polinomio a -> a -> a
horner' p x = (foldr (\a b -> a + b*x) 0) (coeficientes p)
```

10.4. La regla de Ruffini

El objetivo de esta sección es implementar la regla de Ruffini y sus aplicaciones utilizando las implementaciones del TAD de polinomio.

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)))
```

10.4.1. Divisores de un número

Ejercicio 10.4.1.1. *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]
```

Solución:

```
divisores :: Int -> [Int]
divisores n = concat [[x,-x] | x <- [1..abs n], rem n x == 0]
```

10.4.2. Término independiente de un polinomio

Ejercicio 10.4.2.1. *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
```

Solución:

```
terminoIndep :: Num a => Polinomio  a -> a
terminoIndep p = coeficiente 0 p
```

10.4.3. Paso de la regla de Ruffini

Ejercicio 10.4.3.1. *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
```

Solución:

```
pRuffini :: Int -> [Int] -> [Int]
pRuffini r p@(c:cs) =
    c : [x+r*y | (x,y) <- zip cs (pRuffini r p)]
```

Otra forma de definirla es

```
pRuffini' :: Int -> [Int] -> [Int]
pRuffini' r = scanl1 (\s x -> s * r + x)
```

10.4.4. Cociente mediante la regla de Ruffini

Ejercicio 10.4.4.1. *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
```

Solución:

```
cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = creaPolDispersa (init (pRuffini r (coeficientes p)))
```

10.4.5. Resto mediante la regla de Ruffini

Ejercicio 10.4.5.1. *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
```

Solución:

```
restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (pRuffini r (coeficientes p))
```

Ejercicio 10.4.5.2. *Comprobar con QuickCheck que, dado un polinomio* `p` *y un número entero* `r`*, las funciones anteriores verifican la propiedad de la división euclídea.*

Solución: 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 = creaPolDispersa [1,-r]
          res = creaTermino 0 (restoRuffini r p)
```

La comprobación es

```
ghci> quickCheck prop_diviEuclidea
+++ OK, passed 100 tests.
```

10.4.6. Raíces mediante la regla de Ruffini

Ejercicio 10.4.6.1. *Definir la función*

```
esRaizRuffini:: Int -> Polinomio Int -> Bool
```

tal que `(esRaizRuffini r p)` *se verifica si* `r` *es una raiz de* `p`*, usando para ello el regla de Ruffini. Por ejemplo,*

```
esRaizRuffini 0 ejPol3 == True
esRaizRuffini 1 ejPol3 == False
```

Solución:

```
esRaizRuffini :: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0
```

Ejercicio 10.4.6.2. *Definir la función*

```
raicesRuffini :: Polinomio Int -> [Int]
```

tal que `(raicesRuffini p)` *es la lista de las raices 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 == []
```

Solución:

```
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
```

10.4.7. Factorización mediante la regla de Ruffini

Ejercicio 10.4.7.1. *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 (creaPolDispersa [1,0,0,0,-1]) == [1*x + -1,1*x + 1,x^2 + 1]
```

Solución:

```
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 =
              (creaPolDispersa [1,-r]) : factorizacion (cocienteRuffini r p)
          | otherwise = aux rs
```

Capítulo 11

Vectores y matrices

Contenido

11.1 Posiciones de un elemento en una matriz 244
11.2 Tipos de los vectores y de las matrices 244
11.3 Operaciones básicas con matrices 245
11.4 Suma de matrices 247
11.5 Producto de matrices 249
11.6 Traspuestas y simétricas 250
11.7 Diagonales de una matriz 251
11.8 Submatrices 252
11.9 Transformaciones elementales 252
11.10 Triangularización de matrices 255
11.11 Algoritmo de Gauss para triangularizar matrices 257
11.12 Determinante 260
11.13 Máximo de las sumas de elementos de una matriz en líneas distintas . 261

El objetivo de este capítulo 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 de [1].

Además, en algunos ejemplos de usan matrices con números racionales. En Haskell, el número racional $\frac{x}{y}$ se representa por `x%y`. El TAD de los números racionales está definido en el módulo `Data.Ratio`.

Nota. Se importan ambas librerías

```
import Data.Array
import Data.Ratio
```

11.1. Posiciones de un elemento en una matriz

Ejercicio 11.1.1. *Definir la función*

```
posiciones :: Eq a => a -> Array (Int,Int) a -> [(Int,Int)]
```

tal que `(posiciones x p)` *es la lista de las posiciones de la matriz* `p` *cuyo valor es* `x`*. Por ejemplo,*

```
ghci> let p = listArray ((1,1),(2,3)) [1,2,3,2,4,6]
ghci> p
array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
                     ((2,1),2),((2,2),4),((2,3),6)]
ghci> posiciones 2 p
[(1,2),(2,1)]
ghci> posiciones 6 p
[(2,3)]
ghci> posiciones 7 p
[]
```

Solución:

```
posiciones :: Eq a => a -> Array (Int,Int) a -> [(Int,Int)]
posiciones x p = [(i,j) | (i,j) <- indices p, p!(i,j) == x]
```

11.2. Tipos de los vectores y de las matrices

Nota. 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
```

11.3. Operaciones básicas con matrices

Ejercicio 11.3.1. *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)]
```

Solución:

```
listaVector :: Num a => [a] -> Vector a
listaVector xs = listArray (1,n) xs
    where n = length xs
```

Ejercicio 11.3.2. *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)]
```

Solución:

```
listaMatriz :: Num a => [[a]] -> Matriz a
listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
    where m = length xss
          n = length (head xss)
```

Ejercicio 11.3.3. *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
```

Solución:

```
numFilas :: Num a => Matriz a -> Int
numFilas = fst . snd . bounds
```

Ejercicio 11.3.4. *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
```

Solución:

```
numColumnas :: Num a => Matriz a -> Int
numColumnas = snd . snd . bounds
```

Ejercicio 11.3.5. *Definir la función*

```
dimension :: Num a => Matriz a -> (Int,Int)
```

tal que (`dimension m`) *es el número de filas y columnas de la matriz* `m`*. Por ejemplo,*

```
dimension (listaMatriz [[1,3,5],[2,4,7]]) == (2,3)
```

Solución:

```
dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas p)
```

Ejercicio 11.3.6. *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]]
```

Solución:

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

Ejercicio 11.3.7. *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]]
```

Solución:

```
matrizLista :: Num a => Matriz a -> [[a]]
matrizLista p = separa (numColumnas p) (elems p)
```

Ejercicio 11.3.8. *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]
```

Solución:

```
vectorLista :: Num a => Vector a -> [a]
vectorLista = elems
```

11.4. Suma de matrices

Ejercicio 11.4.1. *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]]
```

Solución:

```
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 11.4.2. *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]
```

Solución:

```
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 11.4.3. *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]
```

Solución:

```
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
```

11.5. Producto de matrices

Ejercicio 11.5.1. *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
```

Solución:

```
prodEscalar :: Num a => Vector a -> Vector a -> a
prodEscalar v1 v2 =
    sum [i*j | (i,j) <- zip (elems v1) (elems v2)]
```

Ejercicio 11.5.2. *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]]
```

Solución:

```
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
```

11.6. Traspuestas y simétricas

Ejercicio 11.6.1. *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]]
```

Solución:

```
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 11.6.2. *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
```

Solución:

```
esCuadrada :: Num a => Matriz a -> Bool
esCuadrada x = numFilas x == numColumnas x
```

Ejercicio 11.6.3. *Definir la función*

```
esSimetrica :: Num 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
```

Solución:

```
esSimetrica :: Num a => Matriz a -> Bool
esSimetrica x = x == traspuesta x
```

11.7. Diagonales de una matriz

Ejercicio 11.7.1. *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]
```

Solución:

```
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 11.7.2. *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 (diagonalPral p)
[5,2]
```

Solución:

```
diagonalSec :: Num a => Matriz a -> Vector a
diagonalSec p = array (1,n) [(i,p!(i,m+1-i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)
          m = numFilas p
```

11.8. Submatrices

Ejercicio 11.8.1. *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]]
```

Solución:

```
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)
```

11.9. Transformaciones elementales

Ejercicio 11.9.1. *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]]
```

Solución:

```
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 11.9.2. *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]]
```

Solución:

```
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 11.9.3. *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]]
```

Solución:

```
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 11.9.4. *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` *de 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]]
```

Solución:

```
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 11.9.5. *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]]
```

Solución:

```
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)
```

11.10. Triangularización de matrices

Ejercicio 11.10.1. *Definir la función*

```
buscaIndiceDesde :: Num 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
```

Solución:

```
buscaIndiceDesde :: Num 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 11.10.2. *Definir la función*

```
buscaPivoteDesde :: Num 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
```

Solución:

```
buscaPivoteDesde :: Num 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 11.10.3. *Definir la función*

```
anuladaColumnaDesde :: Num 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
```

Solución:

```
anuladaColumnaDesde :: Num a => Matriz a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
    buscaIndiceDesde p j (i+1) == Nothing
```

Ejercicio 11.10.4. *Definir la función*

```
anulaEltoColumnaDesde :: Fractional 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]]
```

Solución:

```
anulaEltoColumnaDesde :: Fractional 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 11.10.5. *Definir la función*

```
anulaColumnaDesde :: Fractional 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 de la posición* `(i,j)` *(se supone que el elemento* $p_{i,j}$ *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]]
```

Solución:

```
anulaColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
anulaColumnaDesde p j i
    | anuladaColumnaDesde p j i = p
    | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i
```

11.11. Algoritmo de Gauss para triangularizar matrices

Ejercicio 11.11.1. *Definir la función*

```
elementosNoNulosColDesde :: Num 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]
```

Solución:

```
elementosNoNulosColDesde :: Num a => Matriz a -> Int -> Int -> [a]
elementosNoNulosColDesde p j i =
    [x | ((k,j'),x) <- assocs p, x /= 0, j' == j, k >= i]
```

Ejercicio 11.11.2. *Definir la función*

```
existeColNoNulaDesde :: Num 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
```

Solución:

```
existeColNoNulaDesde :: Num 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 11.11.3. *Definir la función*

```
menorIndiceColNoNulaDesde
   :: Num 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
```

Solución:

```
menorIndiceColNoNulaDesde :: (Num 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 11.11.4. *Definir la función*

```
gaussAux :: Fractional 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*
   a) j' *la primera columna a partir de la* j *donde* p *tiene algún elemento no nulo a partir de la fila* i*,*
   b) p_1 *la matriz obtenida intercambiando las columnas* j *y* j' *de* p*,*
   c) i' *la primera fila a partir de la* i *donde la columna* j *de* p_1 *tiene un elemento no nulo,*
   d) p_2 *la matriz obtenida intercambiando las filas* i *e* i' *de la matriz* p_1 *y*
   e) p' *la matriz obtenida anulando todos los elementos de la columna* j *de* p_2 *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]]
```

Solución:

```
gaussAux :: Fractional 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 11.11.5. *Definir la función*

```
gauss :: Fractional 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]]
```

Solución:

```
gauss :: Fractional a => Matriz a -> Matriz a
gauss p = gaussAux p 1 1
```

11.12. Determinante

Ejercicio 11.12.1. *Definir la función*

```
determinante :: Fractional a => Matriz a -> a
```

tal que `(determinante p)` *es el determinante de la matriz* `p`*. Por ejemplo,*

```
ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
ghci> determinante p
0.0
ghci> let p = listaMatriz [[1.0,2,3],[1,3,4],[1,2,5]]
ghci> determinante p
2.0
```

Solución:

```
determinante :: Fractional a => Matriz a -> a
determinante p = product (elems (diagonalPral (gauss p)))
```

11.13. Máximo de las sumas de elementos de una matriz en líneas distintas

Ejercicio 11.13.1 (Problema 345 del proyecto Euler)**.** *Las matrices puede representarse mediante tablas cuyos índices son pares de números naturales:*

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

Definir la función

```
maximaSuma :: Matriz -> Int
```

tal que `(maximaSuma p)` *es el máximo de las sumas de las listas de elementos de la matriz* `p` *tales que cada elemento pertenece sólo a una fila y a una columna. Por ejemplo,*

```
ghci> maximaSuma (listArray ((1,1),(3,3)) [1,2,3,8,4,9,5,6,7])
17
```

ya que las selecciones, y sus sumas, de la matriz

$$\begin{pmatrix} 1 & 2 & 3 \\ 8 & 4 & 9 \\ 5 & 6 & 7 \end{pmatrix}$$

son

Selección	*Suma*
[1,4,7]	*12*
[1,9,6]	*16*
[2,8,7]	*17*
[2,9,5]	*16*
[3,8,6]	*17*
[3,4,5]	*12*

Hay dos selecciones con máxima suma: $[2,8,7]$ *y* $[3,8,6]$.

Solución:

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

maximaSuma :: Matriz -> Int
maximaSuma p = maximum [sum xs | xs <- selecciones p]
```

donde `(selecciones p)` es la lista de las selecciones en las que cada elemento pertenece a un única fila y a una única columna de la matriz `p`. Por ejemplo,

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

```
selecciones :: Matriz -> [[Int]]
selecciones p =
    [[p!(i,j) | (i,j) <- ijs] |
     ijs <- [zip [1..n] xs | xs <- permutations [1..n]]]
    where (_,(m,n)) = bounds p
```

Otra solución (mediante submatrices) es:

```
maximaSuma2 :: Matriz -> Int
maximaSuma2 p
    | (m,n) == (1,1) = p!(1,1)
    | otherwise = maximum [p!(1,j)
                  + maximaSuma2 (submatriz 1 j p) | j <- [1..n]]
    where (m,n) = dimension p
```

donde

- `(dimension p)` es la dimensión de la matriz `p`.

```
dimension :: Matriz -> (Int,Int)
dimension = snd . bounds
```

- `(submatriz i j p)` es la matriz obtenida a partir de la `p` eliminando la fila `i` y la columna `j`. Por ejemplo,

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

```
submatriz :: Int -> Int -> Matriz -> Matriz
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)
```

Capítulo 12

Relaciones binarias homogéneas

Contenido

12.1 Tipo de dato de las relaciones binarias . . . 266
12.2 Universo de una relación . . . 266
12.3 Grafo de una relación . . . 267
12.4 Relaciones reflexivas . . . 267
12.5 Relaciones simétricas . . . 267
12.6 Reconocimiento de subconjuntos . . . 268
12.7 Composición de relaciones . . . 268
12.8 Relación transitiva . . . 268
12.9 Relación de equivalencia . . . 269
12.10 Relación irreflexiva . . . 269
12.11 Relación antisimétrica . . . 269
12.12 Relación total . . . 270
12.13 Clausura reflexiva . . . 271
12.14 Clausura simétrica . . . 272
12.15 Clausura transitiva . . . 273

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 Relación binaria[1]

[1] http://es.wikipedia.org/wiki/Relación_binaria

Nota. Se usarán las siguientes librerías auxiliares

```
import Test.QuickCheck
import Data.List
```

12.1. Tipo de dato de las relaciones binarias

Ejercicio 12.1.1. *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* `\begin{sesion}` *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`*.*

Solución:

```
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)])
```

12.2. Universo de una relación

Ejercicio 12.2.1. *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]
```

Solución:

```
universo :: Eq a => Rel a -> [a]
universo (us,_) = us
```

12.3. Grafo de una relación

Ejercicio 12.3.1. *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)]
```

Solución:

```
grafo :: Eq a => ([a],[(a,a)]) -> [(a,a)]
grafo (_,ps) = ps
```

12.4. Relaciones reflexivas

Ejercicio 12.4.1. *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
```

Solución:

```
reflexiva :: Eq a => Rel a -> Bool
reflexiva (us,ps) = and [(x,x) `elem` ps | x <- us]
```

12.5. Relaciones simétricas

Ejercicio 12.5.1. *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
```

Solución:

```
simetrica :: Eq a => Rel a -> Bool
simetrica (us,ps) = and [(y,x) `elem` ps | (x,y) <- ps]
```

12.6. Reconocimiento de subconjuntos

Ejercicio 12.6.1. *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 [1,3] [3,1,5]  ==  True
subconjunto [3,1,5] [1,3]  ==  False
```

Solución:

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

12.7. Composición de relaciones

Ejercicio 12.7.1. *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)])
```

Solución:

```
composicion :: Eq a => Rel a -> Rel a -> Rel a
composicion (xs,ps) (_,qs) =
    (xs,[(x,z) | (x,y) <- ps, (y',z) <- qs, y == y'])
```

12.8. Relación transitiva

Ejercicio 12.8.1. *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
```

Solución:

```
transitiva :: Eq a => Rel a -> Bool
transitiva r@(xs,ps) =
    subconjunto (grafo (composicion r r)) ps
```

12.9. Relación de equivalencia

Ejercicio 12.9.1. *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
```

Solución:

```
esEquivalencia :: Eq a => Rel a -> Bool
esEquivalencia r = reflexiva r && simetrica r && transitiva r
```

12.10. Relación irreflexiva

Ejercicio 12.10.1. *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
```

Solución:

```
irreflexiva :: Eq a => Rel a -> Bool
irreflexiva (xs,ps) = and [(x,x) `notElem` ps | x <- xs]
```

12.11. Relación antisimétrica

Ejercicio 12.11.1. *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
```

Solución:

```
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.
```

12.12. Relación total

Ejercicio 12.12.1. *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` *etá 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
```

Solución:

```
total :: Eq a => Rel a -> Bool
total (xs,ps) =
    and [(x,y) `elem` ps || (y,x) `elem` ps | x <- xs, y <- xs]
```

Ejercicio 12.12.2. *Comprobar con QuickCheck que las relaciones totales son reflexivas.*

Solución: La propiedad es

```
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.
```

12.13. Clausura reflexiva

Ejercicio 12.13.1. *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)])
```

Solución:

```
clausuraReflexiva :: Eq a => Rel a -> Rel a
clausuraReflexiva (xs,ps) =
  (xs, ps `union` [(x,x) | x <- xs])
```

Ejercicio 12.13.2. *Comprobar con QuickCheck que* `(clausuraReflexiva r)` *es reflexiva.*

Solución: La propiedad es

```
prop_ClausuraReflexiva :: Rel Int -> Bool
prop_ClausuraReflexiva r =
    reflexiva (clausuraReflexiva r)
```

La comprobación es

```
ghci> quickCheck prop_ClausuraRefl
+++ OK, passed 100 tests.
```

12.14. Clausura simétrica

Ejercicio 12.14.1. *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)])
```

Solución:

```
clausuraSimetrica :: Eq a => Rel a -> Rel a
clausuraSimetrica (xs,ps) =
    (xs, ps `union` [(y,x) | (x,y) <- ps])
```

Ejercicio 12.14.2. *Comprobar con QuickCheck que* `(clausuraSimetrica r)` *es simétrica.*

Solución: La propiedad es

```
prop_ClausuraSimetrica :: Rel Int -> Bool
prop_ClausuraSimetrica r =
    simetrica (clausuraSimetrica r)
```

La comprobación es

```
ghci> quickCheck prop_ClausuraSimetrica
+++ OK, passed 100 tests.
```

12.15. Clausura transitiva

Ejercicio 12.15.1. *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)])
```

Solución:

```
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 12.15.2. *Comprobar con QuickCheck que* `(clausuraTransitiva r)` *es transitiva.*

Solución: La propiedad es

```
prop_ClausuraTransitiva :: Rel Int -> Bool
prop_ClausuraTransitiva r =
    transitiva (clausuraTransitiva r)
```

La comprobación es

```
ghci> quickCheck prop_ClausuraTransitiva
+++ OK, passed 100 tests.
```

Capítulo 13

Operaciones con conjuntos

Contenido

13.1 Representación de conjuntos y operaciones básicas **276**
13.1.1 El tipo de los conjuntos 276
13.1.2 El conjunto vacío 277
13.1.3 Reconocimiento del conjunto vacío 277
13.1.4 Pertenencia de un elemento a un conjunto 277
13.1.5 Inserción de un elemento en un conjunto 278
13.1.6 Eliminación de un elemento de un conjunto 278
13.2 Ejercicios sobre conjuntos **278**
13.2.1 Reconocimiento de subconjuntos 278
13.2.2 Reconocimiento de subconjunto propio 280
13.2.3 Conjunto unitario 281
13.2.4 Cardinal de un conjunto 281
13.2.5 Unión de conjuntos 281
13.2.6 Unión de una lista de conjuntos 282
13.2.7 Intersección de conjuntos 283
13.2.8 Intersección de una lista de conjuntos 284
13.2.9 Conjuntos disjuntos 284
13.2.10 Diferencia de conjuntos 285
13.2.11 Diferencia simétrica de conjuntos 285
13.2.12 Filtrado en conjuntos 285
13.2.13 Partición de un conjunto según una propiedad 286

13.2.14 División de un conjunto según un elemento 286
13.2.15 Aplicación de una función a un conjunto 286
13.2.16 Todos los elementos verifican una propiedad 287
13.2.17 Algunos elementos verifican una propiedad 287
13.2.18 Producto cartesiano 287
13.2.19 Orden en el tipo de los conjuntos 288
13.2.20 Conjunto potencia 288
13.2.21 Verificación de propiedades de conjuntos 288

El objetivo de este capítulo de ejercicios es definir operaciones entre conjuntos, representados mediante listas ordenadas sin repeticiones. Esta, y otras representaciones, se encuentran en el tema 17 de [1]. Pra hace el capítulo autocontenido mostramos a continuación las definiciones de dicha representación.

Nota. La cabecera del módulo es

```
{-# LANGUAGE FlexibleInstances #-}
import Test.QuickCheck
```

13.1. Representación de conjuntos y operaciones básicas

13.1.1. El tipo de los conjuntos

El tipo de los conjuntos (con elementos de tipo `a`) como listas ordenadas sin repeticiones es `Conj a`:

```
newtype Conj a = Cj [a]
    deriving Eq
```

Para facilitar la escritura de los conjuntos se define

```
instance (Show a) => Show (Conj a) where
    showsPrec _ (Cj s) cad = showConj s cad

showConj []     cad = showString "{}" cad
showConj (x:xs) cad = showChar '{' (shows x (showl xs cad))
     where showl []     cad = showChar '}' cad
           showl (x:xs) cad = showChar ',' (shows x (showl xs cad))
```

Usaremos los siguientes ejemplos de conjunto: c1, c2, c3, c4 :: Conj Int c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0] c2 = foldr inserta vacio [2,6,8,6,1,2,1,9,6] c3 = Cj [2..100000] c4 = Cj [1..100000]

Se puede comprobar la función de escritura:

```
ghci> c1
0,1,2,3,5,7,9
```

13.1.2. El conjunto vacío

- `vacio` es el conjunto vacío. Por ejemplo,

```
ghci> vacio
```

```
vacio :: Conj a
vacio = Cj []
```

13.1.3. Reconocimiento del conjunto vacío

- `(esVacio c)` se verifica si `c` es el conjunto vacío. Por ejemplo,

```
esVacio c1     ==  False
esVacio vacio  ==  True
```

```
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs
```

13.1.4. Pertenencia de un elemento a un conjunto

- `(pertenece x c)` se verifica si `x` pertenece al conjunto `c`. Por ejemplo,

```
c1              ==  0,1,2,3,5,7,9
pertenece 3 c1  ==  True
pertenece 4 c1  ==  False
```

```
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = x `elem` takeWhile (<= x) s
```

13.1.5. Inserción de un elemento en un conjunto

- `(inserta x c)` es el conjunto obtenido añadiendo el elemento `x` al conjunto `c`. Por ejemplo,

```
c1            ==  0,1,2,3,5,7,9
inserta 5 c1  ==  0,1,2,3,5,7,9
inserta 4 c1  ==  0,1,2,3,4,5,7,9
```

```
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (agrega x s)
    where agrega x []                    = [x]
          agrega x s@(y:ys) | x > y      = y : agrega x ys
                            | x < y      = x : s
                            | otherwise  = s
```

13.1.6. Eliminación de un elemento de un conjunto

- `(elimina x c)` es el conjunto obtenido eliminando el elemento `x` del conjunto `c`. Por ejemplo,

```
c1            ==  0,1,2,3,5,7,9
elimina 3 c1  ==  0,1,2,5,7,9
```

```
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (elimina x s)
    where elimina x []                   = []
          elimina x s@(y:ys) | x > y     = y : elimina x ys
                             | x < y     = s
                             | otherwise = ys
```

13.2. Ejercicios sobre conjuntos

13.2.1. Reconocimiento de subconjuntos

Ejercicio 13.2.1. *Definir la función*

```
subconjunto :: Ord a => Conj a -> Conj a -> Bool
```

tal que `(subconjunto c1 c2)` *se verifica si todos los elementos de* `c1` *pertenecen a* `c2`*. Por ejemplo,*

```
subconjunto (Cj [2..100000]) (Cj [1..100000]) == True
subconjunto (Cj [1..100000]) (Cj [2..100000]) == False
```

Solución: Se presentan distintas definiciones y se compara su eficiencia. La primera definición es

```
subconjunto1 :: Ord a => Conj a -> Conj a -> Bool
subconjunto1 (Cj xs) (Cj ys) = sublista xs ys
    where sublista [] _      = True
          sublista (x:xs) ys = elem x ys && sublista xs ys
```

La segunda definición es

```
subconjunto2 :: Ord a => Conj a -> Conj a -> Bool
subconjunto2 (Cj xs) c =
    and [pertenece x c | x <-xs]
```

La tercera definición

```
subconjunto3 :: Ord a => Conj a -> Conj a -> Bool
subconjunto3 (Cj xs) (Cj ys) = sublista' xs ys
    where sublista' [] _      = True
          sublista' _ []      = False
          sublista' (x:xs) ys@(y:zs) = x >= y && elem x ys &&
                                       sublista' xs zs
```

La cuarta definición es

```
subconjunto4 :: Ord a => Conj a -> Conj a -> Bool
subconjunto4 (Cj xs) (Cj ys) = sublista' xs ys
    where sublista' [] _      = True
          sublista' _ []      = False
          sublista' (x:xs) ys@(y:zs)
              | x < y  = False
              | x == y = sublista' xs zs
              | x > y  = elem x zs && sublista' xs zs
```

La comparación de la eficiencia es

```
ghci> subconjunto1 (Cj [2..100000]) (Cj [1..1000000])
  C-c C-cInterrupted.
ghci> subconjunto2 (Cj [2..100000]) (Cj [1..1000000])
  C-c C-cInterrupted.
```

```
ghci> subconjunto3 (Cj [2..100000]) (Cj [1..1000000])
True
(0.52 secs, 26097076 bytes)
ghci> subconjunto4 (Cj [2..100000]) (Cj [1..1000000])
True
(0.66 secs, 32236700 bytes)
ghci> subconjunto1 (Cj [2..100000]) (Cj [1..10000])
False
(0.54 secs, 3679024 bytes)
ghci> subconjunto2 (Cj [2..100000]) (Cj [1..10000])
False
(38.19 secs, 1415562032 bytes)
ghci> subconjunto3 (Cj [2..100000]) (Cj [1..10000])
False
(0.08 secs, 3201112 bytes)
ghci> subconjunto4 (Cj [2..100000]) (Cj [1..10000])
False
(0.09 secs, 3708988 bytes)
```

En lo que sigue, se usará la 3ª definición:

```
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto = subconjunto3
```

13.2.2. Reconocimiento de subconjunto propio

Ejercicio 13.2.2. *Definir la función*

```
subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
```

tal (`subconjuntoPropio c1 c2`) *se verifica si* `c1` *es un subconjunto propio de* `c2`. *Por ejemplo,*

```
subconjuntoPropio (Cj [2..5]) (Cj [1..7]) == True
subconjuntoPropio (Cj [2..5]) (Cj [1..4]) == False
subconjuntoPropio (Cj [2..5]) (Cj [2..5]) == False
```

Solución:

```
subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
subconjuntoPropio c1 c2 =
    subconjunto c1 c2 && c1 /= c2
```

13.2.3. Conjunto unitario

Ejercicio 13.2.3. *Definir la función*

```
unitario :: Ord a => a -> Conj a
```

tal que `(unitario x)` *es el conjunto* `{x}`*. Por ejemplo,*

```
unitario 5 == 5
```

Solución:

```
unitario :: Ord a => a -> Conj a
unitario x = inserta x vacio
```

13.2.4. Cardinal de un conjunto

Ejercicio 13.2.4. *Definir la función*

```
cardinal :: Conj a -> Int
```

tal que `(cardinal c)` *es el número de elementos del conjunto* `c`*. Por ejemplo,*

```
cardinal c1 == 7
cardinal c2 == 5
```

Solución:

```
cardinal :: Conj a -> Int
cardinal (Cj xs) = length xs
```

13.2.5. Unión de conjuntos

Ejercicio 13.2.5. *Definir la función*

```
union :: Ord a => Conj a -> Conj a -> Conj a
```

tal `(union c1 c2)` *es la unión de ambos conjuntos. Por ejemplo,*

```
union c1 c2             == 0,1,2,3,5,6,7,8,9
cardinal (union2 c3 c4) == 100000
```

Solución: Se considera distintas definiciones y se compara la eficiencia. La primera definición es

```
union1 :: Ord a => Conj a -> Conj a -> Conj a
union1 (Cj xs) (Cj ys) = foldr inserta (Cj ys) xs
```

Otra definión es

```
union2 :: Ord a => Conj a -> Conj a -> Conj a
union2 (Cj xs) (Cj ys) = Cj (unionL xs ys)
    where unionL [] ys = ys
          unionL xs [] = xs
          unionL l1@(x:xs) l2@(y:ys)
              | x < y  = x : unionL xs l2
              | x == y = x : unionL xs ys
              | x > y  = y : unionL l1 ys
```

La comparación de eficiencia es

```
ghci> :set +s
ghci> let c = Cj [1..1000]
ghci> cardinal (union1 c c)
1000
(1.04 secs, 56914332 bytes)
ghci> cardinal (union2 c c)
1000
(0.01 secs, 549596 bytes)
```

En lo que sigue se usará la segunda definición

```
union :: Ord a => Conj a -> Conj a -> Conj a
union = union2
```

13.2.6. Unión de una lista de conjuntos

Ejercicio 13.2.6. *Definir la función*

```
unionG :: Ord a => [Conj a] -> Conj a
```

tal `(unionG cs)` *calcule la unión de la lista de conjuntos* `cs`*. Por ejemplo,*

```
unionG [c1, c2] == 0,1,2,3,5,6,7,8,9
```

Solución:

```
unionG :: Ord a => [Conj a] -> Conj a
unionG []          = vacio
unionG (Cj xs:css) = Cj xs `union` unionG css
```

Se puede definir por plegados

```
unionG2 :: Ord a => [Conj a] -> Conj a
unionG2 = foldr union vacio
```

13.2.7. Intersección de conjuntos

Ejercicio 13.2.7. *Definir la función*

```
interseccion :: Eq a => Conj a -> Conj a -> Conj a
```

tal que `(interseccion c1 c2)` *es la intersección de los conjuntos* `c1` *y* `c2`*. Por ejemplo,*

```
interseccion (Cj [1..7]) (Cj [4..9])    == 4,5,6,7
interseccion (Cj [2..1000000]) (Cj [1]) ==
```

Solución: Se da distintas definiciones y se compara su eficiencia. La primera definición es

```
interseccion1 :: Eq a => Conj a -> Conj a -> Conj a
interseccion1 (Cj xs) (Cj ys) = Cj [x | x <- xs, x `elem` ys]
```

La segunda definición es

```
interseccion2 :: Ord a => Conj a -> Conj a -> Conj a
interseccion2 (Cj xs) (Cj ys) = Cj (interseccionL xs ys)
    where interseccionL l1@(x:xs) l2@(y:ys)
              | x > y   = interseccionL l1 ys
              | x == y  = x : interseccionL xs ys
              | x < y   = interseccionL xs l2
          interseccionL _ _ = []
```

La comparación de eficiencia es

```
ghci> interseccion1 (Cj [2..1000000]) (Cj [1])

(0.32 secs, 80396188 bytes)
ghci> interseccion2 (Cj [2..1000000]) (Cj [1])

(0.00 secs, 2108848 bytes)
```

En lo que sigue se usa la segunda definición:

```
interseccion :: Ord a => Conj a -> Conj a -> Conj a
interseccion = interseccion2
```

13.2.8. Intersección de una lista de conjuntos

Ejercicio 13.2.8. *Definir la función*

```
interseccionG :: Ord a => [Conj a] -> Conj a
```

tal que `(interseccionG cs)` *es la intersección de la lista de conjuntos* `cs`*. Por ejemplo,*

```
interseccionG [c1, c2] == 1,2,9
```

Solución:

```
interseccionG :: Ord a => [Conj a] -> Conj a
interseccionG [c] = c
interseccionG (cs:css) = interseccion cs (interseccionG css)
```

Se puede definir por plegado

```
interseccionG2 :: Ord a => [Conj a] -> Conj a
interseccionG2 = foldr1 interseccion
```

13.2.9. Conjuntos disjuntos

Ejercicio 13.2.9. *Definir la función*

```
disjuntos :: Ord a => Conj a -> Conj a -> Bool
```

tal que `(disjuntos c1 c2)` *se verifica si los conjuntos* `c1` *y* `c2` *son disjuntos. Por ejemplo,*

```
disjuntos (Cj [2..5]) (Cj [6..9]) == True
disjuntos (Cj [2..5]) (Cj [1..9]) == False
```

Solución:

```
disjuntos :: Ord a => Conj a -> Conj a -> Bool
disjuntos c1 c2 = esVacio (interseccion c1 c2)
```

13.2.10. Diferencia de conjuntos

Ejercicio 13.2.10. *Ejercicio 10. Definir la función*

```
diferencia :: Eq a => Conj a -> Conj a -> Conj a
```

tal que `(diferencia c1 c2)` *es el conjunto de los elementos de* `c1` *que no son elementos de* `c2`. *Por ejemplo,*

```
diferencia c1 c2 == 0,3,5,7
diferencia c2 c1 == 6,8
```

Solución:

```
diferencia :: Eq a => Conj a -> Conj a -> Conj a
diferencia (Cj xs) (Cj ys) = Cj zs
    where zs = [x | x <- xs, x `notElem` ys]
```

13.2.11. Diferencia simétrica de conjuntos

Ejercicio 13.2.11. *Definir la función*

```
diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
```

tal que `(diferenciaSimetrica c1 c2)` *es la diferencia simétrica de los conjuntos* `c1` *y* `c2`. *Por ejemplo,*

```
diferenciaSimetrica c1 c2 == 0,3,5,6,7,8
diferenciaSimetrica c2 c1 == 0,3,5,6,7,8
```

Solución:

```
diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
diferenciaSimetrica c1 c2 =
    diferencia (union c1 c2) (interseccion c1 c2)
```

13.2.12. Filtrado en conjuntos

Ejercicio 13.2.12. *Definir la función*

```
filtra :: (a -> Bool) -> Conj a -> Conj a
```

tal `(filtra p c)` *es el conjunto de elementos de* `c` *que verifican el predicado* `p`. *Por ejemplo,*

```
filtra even c1 == 0,2
filtra odd  c1  == 1,3,5,7,9
```

Solución:

```
filtra :: (a -> Bool) -> Conj a -> Conj a
filtra p (Cj xs) = Cj (filter p xs)
```

13.2.13. Partición de un conjunto según una propiedad

Ejercicio 13.2.13. *Definir la función*

```
particion :: (a -> Bool) -> Conj a -> (Conj a, Conj a)
```

tal que `(particion c)` *es el par formado por dos conjuntos: el de sus elementos que verifican* `p` *y el de los elementos que no lo verifica. Por ejemplo,*

```
particion even c1 == (0,2,1,3,5,7,9)
```

Solución:

```
particion :: (a -> Bool) -> Conj a -> (Conj a, Conj a)
particion p c = (filtra p c, filtra (not . p) c)
```

13.2.14. División de un conjunto según un elemento

Ejercicio 13.2.14. *Definir la función*

```
divide :: Ord a => a -> Conj a -> (Conj a, Conj a)
```

tal que `(divide x c)` *es el par formado por dos subconjuntos de* `c`*: el de los elementos menores o iguales que* `x` *y el de los mayores que* `x`*. Por ejemplo,*

```
divide 5 c1 == (0,1,2,3,5,7,9)
```

Solución:

```
divide :: Ord a => a-> Conj a -> (Conj a, Conj a)
divide x = particion (<= x)
```

13.2.15. Aplicación de una función a un conjunto

Ejercicio 13.2.15. *Definir la función*

```
mapC :: (a -> b) -> Conj a -> Conj b
```

tal que `(map f c)` *es el conjunto formado por las imágenes de los elementos de* `c`*, mediante* `f`*. Por ejemplo,*

```
mapC (*2) (Cj [1..4]) == 2,4,6,8
```

Solución:

```
mapC :: (a -> b) -> Conj a -> Conj b
mapC f (Cj xs) = Cj (map f xs)
```

13.2.16. Todos los elementos verifican una propiedad

Ejercicio 13.2.16. *Definir la función*

```
everyC :: (a -> Bool) -> Conj a -> Bool
```

tal que `(everyC p c)` *se verifica si todos los elementos de* `c` *verifican el predicado* `p`*. Por ejemplo,*

```
everyC even (Cj [2,4..10]) == True
everyC even (Cj [2..10])   == False
```

Solución:

```
everyC :: (a -> Bool) -> Conj a -> Bool
everyC p (Cj xs) = all p xs
```

13.2.17. Algunos elementos verifican una propiedad

Ejercicio 13.2.17. *Definir la función*

```
someC :: (a -> Bool) -> Conj a -> Bool
```

tal que `(someC p c)` *se verifica si algún elemento de* `c` *verifica el predicado* `p`*. Por ejemplo,*

```
someC even (Cj [1,4,7]) == True
someC even (Cj [1,3,7]) == False
```

Solución:

```
someC :: (a -> Bool) -> Conj a -> Bool
someC p (Cj xs) = any p xs
```

13.2.18. Producto cartesiano

Ejercicio 13.2.18. *Definir la función*

```
productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
```

tal que `(productoC c1 c2)` *es el producto cartesiano de los conjuntos* `c1` *y* `c2`*. Por ejemplo,*

```
productoC (Cj [1,3]) (Cj [2,4])== (1,2),(1,4),(3,2),(3,4)
```

Solución:

```
productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
productoC (Cj xs) (Cj ys) =
    foldr inserta vacio [(x,y) | x <- xs, y <- ys]
```

13.2.19. Orden en el tipo de los conjuntos

Ejercicio 13.2.19. *Especificar que, dado un tipo ordenado* `a`*, el orden entre los conjuntos con elementos en* `a` *es el orden inducido por el orden existente entre las listas con elementos en* `a`*.*

Solución:

```
instance Ord a => Ord (Conj a) where
    (Cj xs) <= (Cj ys) = xs <= ys
```

13.2.20. Conjunto potencia

Ejercicio 13.2.20. *Definir la función*

```
potencia :: Ord a => Conj a -> Conj (Conj a)
```

tal que `(potencia c)` *es el conjunto potencia de* `c`*; es decir, el conjunto de todos los subconjuntos de* `c`*. Por ejemplo,*

```
potencia (Cj [1,2])  == ,1,1,2,2
potencia (Cj [1..3]) == ,1,1,2,1,2,3,1,3,2,2,3,3
```

Solución:

```
potencia :: Ord a => Conj a -> Conj (Conj a)
potencia (Cj []) = unitario vacio
potencia (Cj (x:xs)) = mapC (inserta x) pr `union` pr
    where pr = potencia (Cj xs)
```

13.2.21. Verificación de propiedades de conjuntos

Generador de conjuntos

Para verificar las propidades con QuickCheck se define `genConjunto` que es un generador de conjuntos. Por ejemplo,

```
ghci> sample genConjunto



3,-2,-2,-3,-2,4
-8,0,4,6,-5,-2
12,-2,-1,-10,-2,2,15,15
2
```

```
-42,55,55,-11,23,23,-11,27,-17,-48,16,-15,-7,5,41,43
-124,-66,-5,-47,58,-88,-32,-125
49,-38,-231,-117,-32,-3,45,227,-41,54,169,-160,19
```

```
genConjunto :: Gen (Conj Int)
genConjunto = do xs <- listOf arbitrary
                 return (foldr inserta vacio xs)
```

y se declara que los conjuntos son concreciones de los arbitrarios:

```
instance Arbitrary (Conj Int) where
    arbitrary = genConjunto
```

Ejercicio 13.2.21. *Comprobar con QuickCheck que la relación de subconjunto es un orden parcial. Es decir, es una relación reflexiva, antisimétrica y transitiva.*

Solución: La propiedad reflexiva es

```
propSubconjuntoReflexiva:: Conj Int -> Bool
propSubconjuntoReflexiva c = subconjunto c c
```

La comprobación es

```
ghci> quickCheck propSubconjuntoReflexiva
+++ OK, passed 100 tests.
```

La propiedad antisimétrica es

```
propSubconjuntoAntisimetrica:: Conj Int -> Conj Int -> Property
propSubconjuntoAntisimetrica c1 c2 =
    subconjunto c1 c2 && subconjunto c2 c1 ==> c1 == c2
```

La comprobación es

```
ghci> quickCheck propSubconjuntoAntisimetrica
*** Gave up! Passed only 13 tests.
```

La propiedad transitiva es

```
propSubconjuntoTransitiva :: Conj Int -> Conj Int -> Conj Int -> Property
propSubconjuntoTransitiva c1 c2 c3 =
    subconjunto c1 c2 && subconjunto c2 c3 ==> subconjunto c1 c3
```

La comprobación es

```
ghci> quickCheck propSubconjuntoTransitiva
*** Gave up! Passed only 7 tests.
```

Ejercicio 13.2.22. *Comprobar con QuickCheck que el conjunto vacío está contenido en cualquier conjunto.*

Solución: La propiedad es

```
propSubconjuntoVacio:: Conj Int -> Bool
propSubconjuntoVacio c = subconjunto vacio c
```

La comprobación es

```
ghci> quickCheck propSubconjuntoVacio
+++ OK, passed 100 tests.
```

Ejercicio 13.2.23. *Comprobar con QuickCheck las siguientes propiedades de la unión de conjuntos:*

Idempotente:	$A \cup A = A$
Neutro:	$A \cup \emptyset = A$
Commutativa:	$A \cup B = B \cup A$
Asociativa:	$A \cup (B \cup C) = (A \cup B) \cup C$
Subconjunto:	$A \subseteq (A \cup B)$, $B \subseteq (A \cup B)$
Diferencia:	$A \cup B = A \cup (B\ A)$

Solución: Las propiedades son

```
propUnionIdempotente :: Conj Int -> Bool
propUnionIdempotente c =
    union c c == c

propVacioNeutroUnion:: Conj Int -> Bool
propVacioNeutroUnion c =
    union c vacio == c

propUnionCommutativa:: Conj Int -> Conj Int -> Bool
propUnionCommutativa c1 c2 =
    union c1 c2 == union c2 c1

propUnionAsociativa:: Conj Int -> Conj Int -> Conj Int -> Bool
propUnionAsociativa c1 c2 c3 =
```

```
    union c1 (union c2 c3) == union (union c1 c2) c3

propUnionSubconjunto :: Conj Int -> Conj Int -> Bool
propUnionSubconjunto c1 c2 =
    subconjunto c1 c3 && subconjunto c2 c3
    where c3 = union c1 c2

propUnionDiferencia :: Conj Int -> Conj Int -> Bool
propUnionDiferencia c1 c2 =
    union c1 c2 == union c1 (diferencia c2 c1)
```

Sus comprobaciones son

```
ghci> quickCheck propUnionIdempotente
+++ OK, passed 100 tests.

ghci> quickCheck propVacioNeutroUnion
+++ OK, passed 100 tests.

ghci> quickCheck propUnionCommutativa
+++ OK, passed 100 tests.

ghci> quickCheck propUnionAsociativa
+++ OK, passed 100 tests.

ghci> quickCheck propUnionSubconjunto
+++ OK, passed 100 tests.

ghci> quickCheck propUnionDiferencia
+++ OK, passed 100 tests.
```

Ejercicio 13.2.24. *Comprobar con QuickCheck las siguientes propiedades de la intersección de conjuntos:*

Idempotente:	$A \cap A = A$
VacioInterseccion:	$A \cap \emptyset = \emptyset$
Commutativa:	$A \cap B = B \cap A$
Asociativa:	$A \cap (B \cap C) = (A \cap B) \cap C$
InterseccionSubconjunto:	$(A \cap B) \subseteq A$, $(A \cap B) \subseteq A$
DistributivaIU:	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
DistributivaUI:	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

Solución: Las propiedades son

```
propInterseccionIdempotente :: Conj Int -> Bool
propInterseccionIdempotente c =
    interseccion c c == c

propVacioInterseccion:: Conj Int -> Bool
propVacioInterseccion c =
    interseccion c vacio == vacio

propInterseccionCommutativa:: Conj Int -> Conj Int -> Bool
propInterseccionCommutativa c1 c2 =
    interseccion c1 c2 == interseccion c2 c1

propInterseccionAsociativa:: Conj Int -> Conj Int -> Conj Int -> Bool
propInterseccionAsociativa c1 c2 c3 =
    interseccion c1 (interseccion c2 c3) == interseccion (interseccion c1 c2) c3

propInterseccionSubconjunto :: Conj Int -> Conj Int -> Bool
propInterseccionSubconjunto c1 c2 =
    subconjunto c3 c1 && subconjunto c3 c2
        where c3 = interseccion c1 c2

propDistributivaIU:: Conj Int -> Conj Int -> Conj Int -> Bool
propDistributivaIU c1 c2 c3 =
    interseccion c1 (union c2 c3) == union (interseccion c1 c2)
                                           (interseccion c1 c3)

propDistributivaUI:: Conj Int -> Conj Int -> Conj Int -> Bool
propDistributivaUI c1 c2 c3 =
    union c1 (interseccion c2 c3) == interseccion (union c1 c2)
                                                  (union c1 c3)
```

Sus comprobaciones son

```
ghci> quickCheck propInterseccionIdempotente
+++ OK, passed 100 tests.

ghci> quickCheck propVacioInterseccion
+++ OK, passed 100 tests.

ghci> quickCheck propInterseccionCommutativa
```

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

ghci> quickCheck propInterseccionAsociativa
+++ OK, passed 100 tests.

ghci> quickCheck propInterseccionSubconjunto
+++ OK, passed 100 tests.

ghci> quickCheck propDistributivaIU
+++ OK, passed 100 tests.

ghci> quickCheck propDistributivaUI
+++ OK, passed 100 tests.
```

Ejercicio 13.2.25. *Comprobar con QuickCheck las siguientes propiedades de la diferencia de conjuntos:*

DiferenciaVacio1: $A \setminus \emptyset = A$
DiferenciaVacio2: $\emptyset \setminus A = \emptyset$
DiferenciaDif1: $(A \setminus B) \setminus C = A \setminus (B \cup C)$
DiferenciaDif2: $A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$
DiferenciaSubc: $(A \setminus B) \subseteq A$
DiferenciaDisj: *A y $B \setminus A$ son disjuntos*
DiferenciaUI: $(A \cup B) \setminus A = B \setminus (A \cap B)$

Solución: Las propiedades son

```
propDiferenciaVacio1 :: Conj Int -> Bool
propDiferenciaVacio1 c = diferencia c vacio == c

propDiferenciaVacio2 :: Conj Int -> Bool
propDiferenciaVacio2 c = diferencia vacio c == vacio

propDiferenciaDif1 :: Conj Int -> Conj Int -> Conj Int -> Bool
propDiferenciaDif1 c1 c2 c3 =
  diferencia (diferencia c1 c2) c3 == diferencia c1 (union c2 c3)

propDiferenciaDif2 :: Conj Int -> Conj Int -> Conj Int -> Bool
propDiferenciaDif2 c1 c2 c3 =
  diferencia c1 (diferencia c2 c3) == union (diferencia c1 c2)
                                            (interseccion c1 c3)

propDiferenciaSubc:: Conj Int -> Conj Int -> Bool
```

```
propDiferenciaSubc c1 c2 =
  subconjunto (diferencia c1 c2) c1

propDiferenciaDisj:: Conj Int -> Conj Int -> Bool
propDiferenciaDisj c1 c2 =
  disjuntos c1 (diferencia c2 c1)

propDiferenciaUI:: Conj Int -> Conj Int -> Bool
propDiferenciaUI c1 c2 =
  diferencia (union c1 c2) c1 == diferencia c2 (interseccion c1 c2)
```

Sus comprobaciones son

```
ghci> quickCheck propDiferenciaVacio2
+++ OK, passed 100 tests.

ghci> quickCheck propDiferenciaVacio2
+++ OK, passed 100 tests.

ghci> quickCheck propDiferenciaDif1
+++ OK, passed 100 tests.

ghci> quickCheck propDiferenciaDif2
+++ OK, passed 100 tests.

ghci> quickCheck propDiferenciaSubc
+++ OK, passed 100 tests.

ghci> quickCheck propDiferenciaDisj
+++ OK, passed 100 tests.

ghci> quickCheck propDiferenciaUI
+++ OK, passed 100 tests.
```

Capítulo 14

Grafos

En este capítulo se proponen ejercicios con el tipo abstracto de datos (TAD) de los grafos presentados en el tema 22 de [1]. Para hacerlo autocontenido se recuerdan el TAD y sus implementaciones.

Contenido

14.1 El TAD de los grafos **296**
14.1.1 Especificación del TAD de los grafos 296
14.1.2 Los grafos como vectores de adyacencia 297
14.1.3 Los grafos como matrices de adyacencia 300
14.1.4 Los grafos como listas 304
14.2 Ejercicios sobre grafos **309**
14.2.1 Generador de grafos 310
14.2.2 El grafo completo de orden n 312
14.2.3 El ciclo de orden n 312
14.2.4 Número de vértices 313
14.2.5 Reconocimiento de grafos no dirigidos 313
14.2.6 Vértices incidentes 314
14.2.7 Vértices contiguos 314
14.2.8 Lazos 314
14.2.9 Número de lazos 315
14.2.10 Número de aristas 315
14.2.11 Grado positivo de un vértice 316
14.2.12 Grado negativo de un vértice 317
14.2.13 Grado de un vértice 317

14.2.14 Grafos regulares 319
14.2.15 Grafos *k*–regulares 320

14.1. El TAD de los grafos

14.1.1. Especificación del TAD de los grafos

La signatura del TAD de los grafos es

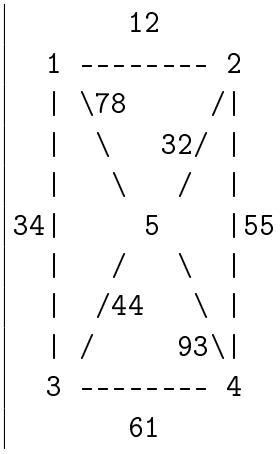
```
creaGrafo   :: (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                               Grafo v p
dirigido    :: (Ix v,Num p) => (Grafo v p) -> Bool
adyacentes  :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
nodos       :: (Ix v,Num p) => (Grafo v p) -> [v]
aristas     :: (Ix v,Num p) => (Grafo 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
```

donde el significado de las operaciones es

- `(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 trío formado por los dos vértices y su peso). Por ejemplo,

```
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)]
```

  crea el grafo



- `(dirigido g)` se verifica si `g` es dirigido.
- `(nodos g)` es la lista de todos los nodos del grafo `g`.
- `(aristas g)` es la lista de las aristas del grafo `g`.
- `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`.
- `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`.
- `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo `g`.

14.1.2. Los grafos como vectores de adyacencia

En el módulo `GrafoConVectorDeAdyacencia` se implementa el TAD de los grafos mediante vectores de adyacencia. La cabecera del módulo es

```
module GrafoConVectorDeAdyacencia
    (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
```

Se usa la librería `Array`

```
import Data.Array
```

La implementación del TAD es la siguiente:

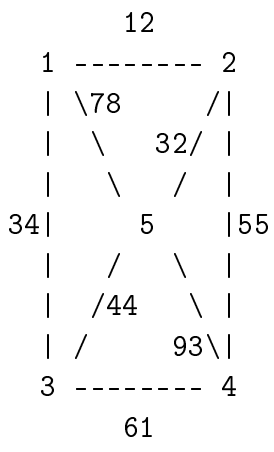
- `Orientacion` es `D` (dirigida) ó `ND` (no dirigida).

```
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 (Array v [(v,p)])
                 deriving (Eq, Show)
```

- `(creaGrafo o 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 trío formado por los dos vértices y su peso). Por ejemplo,
  - El grafo no dirigido correspondiente a



    se define por

```
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)]
```

    y su valor es

```
ghci> ejGrafoND
G ND array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                  (2,[(1,12),(4,55),(5,32)]),
                  (3,[(1,34),(4,61),(5,44)]),
                  (4,[(2,55),(3,61),(5,93)]),
                  (5,[(1,78),(2,32),(3,44),(4,93)])])
```

  - El grafo dirigido correspondiente a la figura anterior se define por

```
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)]
```

su valor es

```
ghci> ejGrafoD
G D array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                 (2,[(4,55),(5,32)]),
                 (3,[(4,61),(5,44)]),
                 (4,[(5,93)]),
                 (5,[])]
```

La definición de `creaGrafo` es

```
creaGrafo :: (Ix v, Num p) => Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs vs =
    G o (accumArray
         (\xs x -> xs++[x]) [] cs
         ((if o == D then []
           else [(x2,(x1,p))|(x1,x2,p) <- vs, x1 /= x2]) ++
          [(x1,(x2,p)) | (x1,x2,p) <- vs]))
```

- `(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
```

- `(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 _ g) = indices g
```

- `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`. Por ejemplo,

```
adyacentes ejGrafoND 4  ==  [2,3,5]
adyacentes ejGrafoD  4  ==  [5]
```

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes (G _ g) v = map fst (g!v)
```

- `(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` adyacentes g x
```

- `(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 _ g) = head [c | (a,c) <- g!x , a == y]
```

- `(aristas 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 o g) = [(v1,v2,w) | v1 <- nodos (G o g) , (v2,w) <- g!v1]
```

14.1.3. Los grafos como matrices de adyacencia

En el módulo `GrafoConMatrizDeAdyacencia` se implementa el TAD de los grafos mediante matrices de adyacencia. La cabecera del módulo es

```
module GrafoConMatrizDeAdyacencia
    (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
```

Se usa la librería `Array`

```
import Data.Array
```

La implementación del TAD es la siguiente:

- `Orientacion` 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 (Array (v,v) (Maybe p))
                 deriving (Eq, Show)
  ```

- `(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 trío formado por los dos vértices y su peso). Por ejemplo,

  - El grafo no dirigido correspondiente a

    ```
          12
      1 -------- 2
      | \78     /|
      |  \   32/ |
      |   \   /  |
    34|     5    |55
      |   /   \  |
      |  /44   \ |
      | /     93\|
      3 -------- 4
          61
    ```

se define por

```
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)]
```

y su valor es

```
ghci> ejGrafoND
G ND array ((1,1),(5,5))
           [((1,1),Nothing),((1,2),Just 12),((1,3),Just 34),
            ((1,4),Nothing),((1,5),Just 78),((2,1),Just 12),
            ((2,2),Nothing),((2,3),Nothing),((2,4),Just 55),
            ((2,5),Just 32),((3,1),Just 34),((3,2),Nothing),
            ((3,3),Nothing),((3,4),Just 61),((3,5),Just 44),
            ((4,1),Nothing),((4,2),Just 55),((4,3),Just 61),
            ((4,4),Nothing),((4,5),Just 93),((5,1),Just 78),
            ((5,2),Just 32),((5,3),Just 44),((5,4),Just 93),
            ((5,5),Nothing)]
```

- El grafo dirigido correspondiente a la figura anterior se define por

```
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)]
```

su valor es

```
ghci> ejGrafoD
G D (array ((1,1),(5,5))
           [((1,1),Nothing),((1,2),Just 12),((1,3),Just 34),
            ((1,4),Nothing),((1,5),Just 78),((2,1),Nothing),
            ((2,2),Nothing),((2,3),Nothing),((2,4),Just 55),
            ((2,5),Just 32),((3,1),Nothing),((3,2),Nothing),
            ((3,3),Nothing),((3,4),Just 61),((3,5),Just 44),
            ((4,1),Nothing),((4,2),Nothing),((4,3),Nothing),
            ((4,4),Nothing),((4,5),Just 93),((5,1),Nothing),
            ((5,2),Nothing),((5,3),Nothing),((5,4),Nothing),
            ((5,5),Nothing)])
```

La definición de `creaGrafo` es

```
creaGrafo :: (Ix v, Num p) => Orientacion -> (v,v) -> [(v,v,p)] -> (Grafo v p)
creaGrafo o cs@(l,u) as
```

```
    = G o (matrizVacia //
           ([((x1,x2),Just w) | (x1,x2,w) <- as] ++
            if o == D then []
            else [((x2,x1),Just w) | (x1,x2,w) <- as, x1 /= x2]))
      where
      matrizVacia = array ((l,l),(u,u))
                          [((x1,x2),Nothing) | x1 <- range cs,
                                               x2 <- range cs]
```

- `(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
```

- `(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 _ g) = range (l,u)
    where ((l,_),(u,_)) = bounds g
```

- `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`. Por ejemplo,

```
adyacentes ejGrafoND 4  ==  [2,3,5]
adyacentes ejGrafoD  4  ==  [5]
```

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes (G o g) v =
    [v' | v' <- nodos (G o g), (g!(v,v')) /= Nothing]
```

- `(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 :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn (G _o g) (x,y)= (g!(x,y)) /= Nothing
```

- `(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 _ g)  = w where (Just w) = g!(x,y)
```

- `(aristas 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 o e) = [(v1,v2,extrae(e!(v1,v2)))
                     | v1 <- nodos g,
                       v2 <- nodos g,
                       aristaEn g (v1,v2)]
    where extrae (Just w) = w
```

14.1.4. Los grafos como listas

El objetivo de esta sección es implementar el TAD de los grafos mediante listas, de manera análoga a las implementaciones anteriores.

La cabecera del módulo es

```
module GrafoConLista
    (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
```

Se usan las librerías `Array` y `List`:

```
import Data.Array
import Data.List
```

`Orientacion` es `D` (dirigida) ó `ND` (no dirigida).

```
data Orientacion = D | ND
                   deriving (Eq, Show)
```

El tipo `(Grafo v p)` representa los grafos con vértices de tipo `v` y pesos de tipo `p`.

```
data Grafo v p = G Orientacion ([v],[((v,v),p)])
                 deriving (Eq, Show)
```

Ejercicio 14.1.1. *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 trío 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)])
```

Solución:

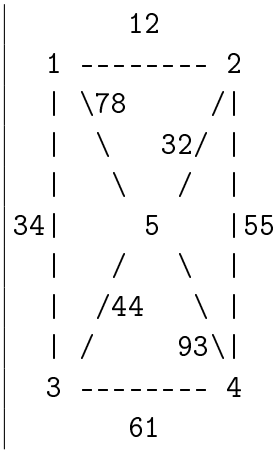
```
creaGrafo :: (Ix v, Num p) =>
             Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs as =
```

```
    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 14.1.2. *Definir, con creaGrafo, la constante*

```
ejGrafoND :: Grafo Int Int
```

para representar el siguiente grafo no dirigido




```
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)])
```

Solución:

```
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 14.1.3. *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)])
```

Solución:

```
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 14.1.4. *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
```

Solución:

```
dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

Ejercicio 14.1.5. *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]
```

Solución:

```
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos (G _ (ns,_)) = ns
```

Ejercicio 14.1.6. *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,*

```
adyacentes ejGrafoND 4  ==  [5,2,3]
adyacentes ejGrafoD  4  ==  [5]
```

Solución:

```
adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adyacentes (G _ (_,e)) v = nub [u | ((w,u),_) <- e, w == v]
```

Ejercicio 14.1.7. *Definir la función*

```
aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
```

tal que `(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
```

Solución:

```
aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x
```

Ejercicio 14.1.8. *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
```

Solución:

```
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 14.1.9. *Definir la función*

```
aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
```

tal que `(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)]
```

Solución:

```
aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
aristas (G _ (_,g)) = [(v1,v2,p) | ((v1,v2),p) <- g]
```

14.2. Ejercicios sobre grafos

El objetivo de esta sección es definir funciones sobre el TAD de los grafos, utilizando las implementaciones anteriores.

La cabecera del módulo es

```
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

import Data.Array
import Data.List (nub)
import Test.QuickCheck

import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

Obsérvese que hay que seleccionar una implementación del TAD de los grafos.

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)]
```

14.2.1. Generador de grafos

Para comprobar propiedades de grafos con QuickCheck se definen las siguientes funciones:

- `(generaGND n ps)` es el grafo completo de orden `n` tal que los pesos están determinados por `\begin{sesion}`. 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 `\begin{sesion}`. 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
```

14.2.2. El grafo completo de orden n

Ejercicio 14.2.1. *El* grafo completo de orden n, $K(n)$, *es un grafo no dirigido cuyos conjunto de vértices es* $\{1,\ldots,n\}$ *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)])])
```

Solución:

```
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]
```

Una definición equivalente es

```
completo' :: Int -> Grafo Int Int
completo' n = creaGrafo ND (1,n) [(a,b,0)|a<-[1..n],b<-[1..a-1]]
```

14.2.3. El ciclo de orden n

Ejercicio 14.2.2. *El* ciclo de orden n, $C(n)$, *es un grafo no dirigido cuyo conjunto de vértices es* $\{1,\ldots,n\}$ *y las aristas son* $(1,2)$, $(2,3)$, $\ldots$, $(n-1,n)$, $(n,1)$.

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)])])
```

Solución:

```
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)]
```

14.2.4. Número de vértices

Ejercicio 14.2.3. *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
```

Solución:

```
nVertices :: (Ix v,Num p) => Grafo v p ->  Int
nVertices = length . nodos
```

14.2.5. Reconocimiento de grafos no dirigidos

Ejercicio 14.2.4. *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
```

Solución:

```
noDirigido :: (Ix v,Num p) => Grafo v p ->  Bool
noDirigido = not . dirigido
```

14.2.6. Vértices incidentes

Ejercicio 14.2.5. *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]
```

Solución:

```
incidentes :: (Ix v,Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]
```

14.2.7. Vértices contiguos

Ejercicio 14.2.6. *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]
```

Solución:

```
contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
contiguos g v = nub (adyacentes g v ++ incidentes g v)
```

14.2.8. Lazos

Ejercicio 14.2.7. *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
[]
```

Solución:

```
lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
lazos g = [(x,x) | x <- nodos g, aristaEn g (x,x)]
```

14.2.9. Número de lazos

Ejercicio 14.2.8. *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
```

Solución:

```
nLazos :: (Ix v,Num p) => Grafo v p ->  Int
nLazos = length . lazos
```

14.2.10. Número de aristas

Ejercicio 14.2.9. *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
```

Solución:

```
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 14.2.10. *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* $\frac{n(n-1)}{2}$ *y, usando la función, comprobar que la propiedad se cumple para* `n` *de 1 a 20.*

Solución: La propiedad es

```
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
```

14.2.11. Grado positivo de un vértice

Ejercicio 14.2.11. *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
```

Solución:

```
gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)
```

14.2.12. Grado negativo de un vértice

Ejercicio 14.2.12. *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
```

Solución:

```
gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)
```

14.2.13. Grado de un vértice

Ejercicio 14.2.13. *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
```

Solución:

```
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 14.2.14. *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`*.*

Solución: 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 14.2.15. *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.*

Solución: La propiedad es

```
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 14.2.16. *Comprobar con QuickCheck que en todo grafo, el número de nodos de grado impar es par.*

Solución: La propiedad es

```
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 14.2.17. *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.*

Solución: La propiedad es

```
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
```

14.2.14. Grafos regulares

Ejercicio 14.2.18. *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
```

Solución:

```
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 14.2.19. *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.*

Solución: La propiedad es

```
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
```

14.2.15. Grafos *k*–regulares

Ejercicio 14.2.20. *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
```

Solución:

```
regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
regularidad g | regular g = Just (grado g (head (nodos g)))
              | otherwise = Nothing
```

Ejercicio 14.2.21. *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.

Solución: La propiedad es

```
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 14.2.22. *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.

Solución: La propiedad es

```
prop_cicloRegular :: Int -> Bool
prop_cicloRegular n =
   regularidad (grafoCiclo n) == Just 2
```

La comprobación es

```
ghci> and [prop_cicloRegular n | n <- [3..20]]
True
```

Parte III

Casos de estudio

Capítulo 15

El cifrado César

En el tema 5 del curso ([1]) se estudió, como aplicación de las definiciones por comprensión, el cifrado César. En el cifrado César cada letra en el texto original es reemplazada por otra letra que se encuentra 3 posiciones más adelante en el alfabeto. Por ejemplo, la codificación de "en todo la medida" es "hq wrgr od phglgd". Se puede generalizar desplazando cada letra n posiciones. Por ejemplo, la codificación con un desplazamiento 5 de "en todo la medida" es "js ytit qf rjinif".

La descodificación de un texto codificado con un desplazamiento n se obtiene codificándolo con un desplazamiento $-n$.

El objetivo de esta relación es modificar el programa de cifrado César para que pueda utilizar también letras mayúsculas. Por ejemplo,

```
ghci> descifra "Ytit Ufwf Sfif"
"Todo Para Nada"
```

Para ello, se propone la modificación de las funciones correspondientes del tema 5.

Nota. Se usará librería `Data.Char`.

```
import Data.Char
```

15.1. Codificación y descodificación

Ejercicio 15.1.1. *Redefinir la función*

```
minuscula2int :: Char -> Int
```

tal que `(minuscula2int c)` *es el entero correspondiente a la letra minúscula* `c`*. Por ejemplo,*

```
minuscula2int 'a'  ==  0
minuscula2int 'd'  ==  3
minuscula2int 'z'  ==  25
```

Solución:

```
minuscula2int :: Char -> Int
minuscula2int c = ord c - ord 'a'
```

Ejercicio 15.1.2. *Redefinir la función*

```
mayuscula2int :: Char -> Int
```

tal que `(mayuscula2int c)` *es el entero correspondiente a la letra mayúscula* `c`*. Por ejemplo,*

```
mayuscula2int 'A'  ==  0
mayuscula2int 'D'  ==  3
mayuscula2int 'Z'  ==  25
```

Solución:

```
mayuscula2int :: Char -> Int
mayuscula2int c = ord c - ord 'A'
```

Ejercicio 15.1.3. *Redefinir la función*

```
int2minuscula :: Int -> Char
```

tal que `(int2minuscula n)` *es la letra minúscula correspondiente al entero* `n`*. Por ejemplo,*

```
int2minuscula 0   ==  'a'
int2minuscula 3   ==  'd'
int2minuscula 25  ==  'z'
```

Solución:

```
int2minuscula :: Int -> Char
int2minuscula n = chr (ord 'a' + n)
```

Ejercicio 15.1.4. *Redefinir la función*

```
int2mayuscula :: Int -> Char
```

tal que `(int2mayuscula n)` *es la letra minúscula correspondiente al entero* `n`*. Por ejemplo,*

```
int2mayuscula 0   ==  'A'
int2mayuscula 3   ==  'D'
int2mayuscula 25  ==  'Z'
```

Solución:

```
int2mayuscula :: Int -> Char
int2mayuscula n = chr (ord 'A' + n)
```

Ejercicio 15.1.5. *Redefinir la función*

```
desplaza :: Int -> Char -> Char
```

tal que `(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'
```

Solución:

```
desplaza :: Int -> Char -> Char
desplaza n c
    | elem c ['a'..'z'] = int2minuscula ((minuscula2int c+n) `mod` 26)
    | elem c ['A'..'Z'] = int2mayuscula ((mayuscula2int c+n) `mod` 26)
    | otherwise         = c
```

Ejercicio 15.1.6. *Redefinir la función*

```
codifica :: Int -> String -> String
```

tal que `(codifica n xs)` *es el resultado de codificar el texto* `xs` *con un desplazamiento* `n`*. Por ejemplo,*

```
ghci> codifica   3  "En Todo La Medida"
"Hq Wrgr Od Phglgd"
ghci> codifica (-3) "Hq Wrgr Od Phglgd"
"En Todo La Medida"
```

Solución:

```
codifica :: Int -> String -> String
codifica n xs = [desplaza n x | x <- xs]
```

15.2. Análisis de frecuencias

Para descifrar mensajes se parte de la frecuencia de aparición de letras.

Ejercicio 15.2.1. *Redefinir la constante*

```
tabla :: [Float]
```

tal que `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 %.*

Solución:

```
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]
```

Ejercicio 15.2.2. *Redefinir la función*

```
porcentaje :: Int -> Int -> Float
```

tal que `(porcentaje n m)` *es el porcentaje de* `n` *sobre* `m`*. Por ejemplo,*

```
porcentaje 2 5 == 40.0
```

Solución:

```
porcentaje :: Int -> Int -> Float
porcentaje n m = (fromIntegral n / fromIntegral m) * 100
```

Ejercicio 15.2.3. *Redefinir la función*

```
letras :: String -> String
```

tal que `(letras xs)` *es la cadena formada por las letras de la cadena* `xs`*. Por ejemplo,*

```
letras "Esto Es Una Prueba" == "EstoEsUnaPrueba"
```

Solución:

```
letras :: String -> String
letras xs = [x | x <- xs, elem x (['a'..'z']++['A'..'Z'])]
```

Ejercicio 15.2.4. *Redefinir la función*

```
ocurrencias :: Char -> String -> Int
```

tal que `(ocurrencias x xs)` *es el número de veces que ocurre el carácter* `x` *en la cadena* `xs`. *Por ejemplo,*

```
ocurrencias 'a' "Salamanca" == 4
```

Solución:

```
ocurrencias :: Char -> String -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']
```

Ejercicio 15.2.5. *Redefinir la función*

```
frecuencias :: String -> [Float]
```

tal que `(frecuencias xs)` *es la frecuencia de cada una de las letras de la cadena* `xs`. *Por ejemplo,*

```
ghci> 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]
```

Solución:

```
frecuencias :: String -> [Float]
frecuencias xs =
    [porcentaje (ocurrencias x xs') n | x <- ['a'..'z']]
    where xs' = [toLower x | x <- xs]
          n   = length (letras xs)
```

15.3. Descifrado

Ejercicio 15.3.1. *Redefinir la función*

```
chiCuad :: [Float] -> [Float] -> Float
```

tal que `chiCuad` *os es) es la medida* χ^2 *de las distribuciones* `os` *y* `es`. *Por ejemplo,*

```
chiCuad [3,5,6] [3,5,6]  ==  0.0
chiCuad [3,5,6] [5,6,3]  ==  3.9666667
```

Solución:

```
chiCuad :: [Float] -> [Float] -> Float
chiCuad os es = sum [((o-e)^2)/e | (o,e) <- zip os es]
```

Ejercicio 15.3.2. *Redefinir la función*

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

tal que `rota` *n xs) es la lista obtenida rotando* `n` *posiciones los elementos de la lista* `xs`*. Por ejemplo,*

```
rota 2 "manolo" == "noloma"
```

Solución:

```
rota :: Int -> [a] -> [a]
rota n xs = drop n xs ++ take n xs
```

Ejercicio 15.3.3. *Redefinir la función*

```
descifra :: String -> String
```

tal que `descifra` *xs) es la cadena obtenida descodificando la cadena* `xs` *por el anti–desplazamiento que produce una distribución de letras con la menor deviación* χ^2 *respecto de la tabla de distribución de las letras en castellano. Por ejemplo,*

```
ghci> codifica 5 "Todo Para Nada"
"Ytit Ufwf Sfif"
ghci> descifra "Ytit Ufwf Sfif"
"Todo Para Nada"
```

Solución:

```
descifra :: String -> String
descifra xs =  codifica (-factor) xs
  where
   factor = head (posiciones (minimum tabChi) tabChi)
   tabChi = [chiCuad (rota n tabla') tabla | n <- [0..25]]
   tabla' = frecuencias xs
```

donde `(posiciones x xs)` es la lista de las posiciones del elemento `x` en la lista `xs`

```
posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
    [i | (x',i) <- zip xs [0..], x == x']
```

Capítulo 16

Codificación y transmisión de mensajes

Contenido

16.1 Cambios de bases 331
16.2 Codificación 333
16.3 Descodificación 335
16.4 Transmisión 336

En esta relación se va a modificar el programa de transmisión de cadenas, presentado en el capítulo 7 de [1], para detectar errores de transmisión sencillos usando bits de paridad. Es decir, cada octeto de ceros y unos generado durante la codificación se extiende con un bit de paridad que será un uno si el número contiene un número impar de unos y cero en caso contrario. En la decodificación, en cada número binario de 9 cifras debe comprobarse que la paridad es correcta, en cuyo caso se descarta el bit de paridad. En caso contrario, debe generarse un mensaje de error en la paridad.

Esta relación es una aplicación del uso de la funciones de orden superior y de plegados.

Nota. Se usará la librería de caracteres.

```
import Data.Char
```

Los bits se representán mediante enteros.

```
type Bit = Int
```

16.1. Cambios de bases

Ejercicio 16.1.1. *Definir, por recursión, la función*

```
bin2intR :: [Bit] -> Int
```

tal que `(bin2intR x)` *es el número decimal correspondiente al número binario* `x`*. Por ejemplo,*

```
bin2intR [1,0,1,1] == 13
```

Solución:

```
bin2intR :: [Bit] -> Int
bin2intR [] = 0
bin2intR (x:xs) = x + 2 * (bin2intR xs)
```

Ejercicio 16.1.2. *Definir, por plegado, la función*

```
bin2int :: [Bit] -> Int
```

tal que `(bin2int x)` *es el número decimal correspondiente al número binario* `x`*. Por ejemplo,*

```
bin2int [1,0,1,1] == 13
```

Solución:

```
bin2int :: [Bit] -> Int
bin2int =  foldr (\x y -> x + 2*y) 0
```

Ejercicio 16.1.3. *Definir, por comprensión, la función*

```
bin2intC :: [Bit] -> Int
```

tal que `(bin2intC x)` *es el número decimal correspondiente al número binario* `x`*. Por ejemplo,*

```
bin2intC [1,0,1,1] == 13
```

Solución:

```
bin2intC :: [Bit] -> Int
bin2intC xs = sum [x*2^n | (x,n) <- zip xs [0..]]
```

Ejercicio 16.1.4. *Definir la función*

```
int2bin :: Int -> [Bit]
```

tal que `(int2bin x)` *es el número binario correspondiente al número decimal* `x`*. Por ejemplo,*

```
int2bin 13 == [1,0,1,1]
```

Solución:

```
int2bin :: Int -> [Bit]
int2bin n | n < 2     = [n]
          | otherwise = n `rem` 2 : int2bin (n `div` 2)
```

Ejercicio 16.1.5. *Comprobar con QuickCheck que al pasar un número natural a binario con* `int2bin` *y el resultado a decimal con* `bin2int` *se obtiene el número inicial.*

Solución: La propiedad es

```
prop_int_bin :: Int -> Bool
prop_int_bin x =
    bin2int (int2bin y) == y
    where y = abs x
```

La comprobación es

```
ghci> quickCheck prop_int_bin
+++ OK, passed 100 tests.
```

16.2. Codificación

Nota. Un octeto es un grupo de ocho bits.

Ejercicio 16.2.1. *Definir la función*

```
creaOcteto :: [Bit] -> [Bit]
```

tal que `(creaOcteto bs)` *es el octeto correspondiente a la lista de bits* `bs`*; es decir, los 8 primeros elementos de* `bs` *si su longitud es mayor o igual que 8 y la lista de 8 elemento añadiendo ceros al final de* `bs` *en caso contrario. Por ejemplo,*

```
creaOcteto [1,0,1,1,0,0,1,1,1,0,0,0]  ==  [1,0,1,1,0,0,1,1]
creaOcteto [1,0,1,1]                  ==  [1,0,1,1,0,0,0,0]
```

Solución:

```
creaOcteto :: [Bit] -> [Bit]
creaOcteto bs =  take 8 (bs ++ repeat 0)
```

La definición anterior puede simplificarse a

```
creaOcteto' :: [Bit] -> [Bit]
creaOcteto' =  take 8 . (++ repeat 0)
```

Ejercicio 16.2.2. *Definir la función*

```
paridad :: [Bit] -> Bit
```

tal que `(paridad bs)` *es el bit de paridad de* `bs`*; es decir, 1 si* `bs` *contiene un número impar de unos y 0 en caso contrario. Por ejemplo,*

```
paridad [0,1,1]      ==  0
paridad [0,1,1,0,1]  ==  1
```

Solución:

```
paridad :: [Bit] -> Bit
paridad bs | odd (sum bs)  = 1
           | otherwise     = 0
```

Ejercicio 16.2.3. *Definir la función*

```
agregaParidad :: [Bit] -> [Bit]
```

tal que `(agregaParidad bs)` *es la lista obtenida añadiendo al principio de* `bs` *su paridad. Por ejemplo,*

```
agregaParidad [0,1,1]      ==  [0,0,1,1]
agregaParidad [0,1,1,0,1]  ==  [1,0,1,1,0,1]
```

Solución:

```
agregaParidad :: [Bit] -> [Bit]
agregaParidad bs = (paridad bs) : bs
```

Ejercicio 16.2.4. *Definir la función*

```
codifica :: String -> [Bit]
```

tal que `(codifica c)` *es la codificación de la cadena 1c como una lista de bits obtenida convirtiendo cada carácter en un número Unicode, convirtiendo cada uno de dichos números en un octeto con su paridad y concatenando los octetos con paridad para obtener una lista de bits. Por ejemplo,*

```
ghci> codifica "abc"
[1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
```

Solución:

```
codifica :: String -> [Bit]
codifica = concat . map (agregaParidad . creaOcteto . int2bin . ord)
```

16.3. Descodificación

Ejercicio 16.3.1. *Definir la función*

```
separa9 :: [Bit] -> [[Bit]]
```

tal que `(separa9 bs)` *es la lista obtenida separando la lista de bits* `bs` *en listas de 9 elementos. Por ejemplo,*

```
ghci> separa9 [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
[[1,1,0,0,0,0,1,1,0],[1,0,1,0,0,0,1,1,0],[0,1,1,0,0,0,1,1,0]]
```

Solución:

```
separa9 :: [Bit] -> [[Bit]]
separa9 []   = []
separa9 bs = take 9 bs : separa9 (drop 9 bs)
```

Ejercicio 16.3.2. *Definir la función*

```
compruebaParidad :: [Bit] -> [Bit ]
```

tal que `(compruebaParidad bs)` *es el resto de* `bs` *si el primer elemento de* `bs` *es el bit de paridad del resto de* `bs` *y devuelve error de paridad en caso contrario. Por ejemplo,*

```
ghci> compruebaParidad [1,1,0,0,0,0,1,1,0]
[1,0,0,0,0,1,1,0]
ghci> compruebaParidad [0,1,0,0,0,0,1,1,0]
*** Exception: paridad erronea
```

Nota: *Usar la función del preludio*

```
error :: String -> a
```

tal que `(error c)` *devuelve la cadena* `c`.

Solución:

```
compruebaParidad :: [Bit] -> [Bit ]
compruebaParidad (b:bs)
    | b == paridad bs = bs
    | otherwise       = error "paridad erronea"
```

Ejercicio 16.3.3. *Definir la función*

```
descodifica :: [Bit] -> String
```

tal que `(descodifica bs)` *es la cadena correspondiente a la lista de bits con paridad* `bs`*. Para ello, en cada número binario de 9 cifras debe comprobarse que la paridad es correcta, en cuyo caso se descarta el bit de paridad. En caso contrario, debe generarse un mensaje de error en la paridad. Por ejemplo,*

```
descodifica [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
==  "abc"
descodifica [1,0,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
==  "*** Exception: paridad erronea
```

Solución:

```
descodifica :: [Bit] -> String
descodifica = map (chr . bin2int . compruebaParidad) . separa9
```

16.4. Transmisión

Ejercicio 16.4.1. *Se define la función*

```
transmite :: ([Bit] -> [Bit]) -> String -> String
transmite canal =  descodifica . canal . codifica
```

tal que `(transmite c t)` *es la cadena obtenida transmitiendo la cadena* `t` *a través del canal* `c`*. Calcular el reultado de trasmitir la cadena* `"Conocete a ti mismo"` *por el canal identidad (*`id`*) y del canal que olvida el primer bit (*`tail`*).*

Solución: El cálculo es

```
ghci> transmite id "Conocete a ti mismo"
"Conocete a ti mismo"
ghci> transmite tail "Conocete a ti mismo"
"*** Exception: paridad erronea
```

Capítulo 17

Resolución de problemas matemáticos

En este capítulo se presentan ejercicios para resolver problemas matemáticos. Se corresponden a los 7 primeros temas de [1].

Contenido

17.1 El problema de Ullman sobre la existencia de subconjunto del tamaño dado y con su suma acotada 338

17.2 Descomposiciones de un número como suma de dos cuadrados 339

17.3 Números reversibles 340

17.4 Grafo de una función sobre los elementos que cumplen una propiedad 341

17.5 Números semiperfectos 342

17.6 Decidir el carácter funcional de una relación 344

17.7 La identidad de Bézout 344

17.8 Distancia entre dos conjuntos de números 346

17.9 Expresables como suma de números consecutivos 346

17.10 Solución de una ecuación diofántica 348

Nota. En esta relación se usan las librerías `List y QuickCheck`.

```
import Data.List
import Test.QuickCheck
```

17.1. El problema de Ullman sobre la existencia de subconjunto del tamaño dado y con su suma acotada

Ejercicio 17.1.1. *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
```

Solución: Se presentan dos soluciones y se compara su eficiencia.

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] /= []
```

donde `(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 eficencia:

```
*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
```

17.2. Descomposiciones de un número como suma de dos cuadrados

Ejercicio 17.2.1. *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)]
```

Solución: Se consideran 3 soluciones y se compara su eficiencia.

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
```

La comparación de las tres definiciones es

n	1ª definición	2ª definición	3ª definición
999	2.17 segs	0.02 segs	0.01 segs
48612265		140.38 segs	0.13 segs

17.3. Números reversibles

Ejercicio 17.3.1 (Basado en el problema 145 del Proyecto Euler[1]). *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
```

Solución:

```
reversiblesMenores :: Int -> Int
reversiblesMenores n = length [x | x <- [1..n-1], esReversible x]
```

En la definición se usan las siguientes funciones auxiliares:

- `(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
```

[1] http://projecteuler.net/problem=145

```
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]
```

17.4. Grafo de una función sobre los elementos que cumplen una propiedad

Ejercicio 17.4.1. *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)]
```

Solución:

```
grafoReducido :: Eq a => (a -> b) -> (a -> Bool) -> [a] -> [(a,b)]
grafoReducido f p xs = [(x,f x) | x <- nub xs, p x]
```

17.5. Números semiperfectos

Ejercicio 17.5.1. *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
```

Solución:

```
esSemiPerfecto :: Int -> Bool
esSemiPerfecto n =
    or [sum ys == n | ys <- subconjuntos (divisores n)]
```

donde se usan las siguientes funciones auxiliares.

- `(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
```

- `(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 17.5.2. *Definir la constante* `primerSemiPerfecto` *tal que su valor es el primer número semiperfecto.*

Solución:

```
primerSemiPerfecto :: Int
primerSemiPerfecto = head [n | n <- [1..], esSemiPerfecto n]
```

La evaluación es

```
ghci> primerSemiPerfecto
6
```

Ejercicio 17.5.3. *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
```

Solución:

```
semiPerfecto :: Int -> Int
semiPerfecto n = semiPerfectos !! n
```

donde `semiPerfectos` es la lista de los números semiPerfectos. Por ejemplo,

```
take 4 semiPerfectos == [6,12,18,20]
```

```
semiPerfectos :: [Int]
semiPerfectos = [n | n <- [1..], esSemiPerfecto n]
```

17.6. Decidir el carácter funcional de una relación

Ejercicio 17.6.1. *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
```

Solución:

```
esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
esFuncion [] = True
esFuncion ((x,y):r) =
    null [y' | (x',y') <- r, x == x', y /= y'] && esFuncion r
```

17.7. La identidad de Bézout

Ejercicio 17.7.1. *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`*.*

Solución: Un ejemplo del cálculo es el siguiente

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,

$$\begin{aligned} 3 &= 15 - 6 * 2 && \text{[por (3)]} \\ &= 15 - (21 - 15 * 1) * 2 && \text{[por (2)]} \\ &= 21 * (-2) + 15 * 3 \\ &= 21 * (-2) + (36 - 21 * 1) * 3 && \text{[por (1)]} \\ &= 36 * 3 + 21 * (-5) \end{aligned}$$

Sean q y r el cociente y el resto de a entre b, d el máximo común múltiplo de a y b y (x, y) el valor de `(bezout b r)`. Entonces,

$$\begin{aligned} a &= bp + r \\ d &= bx + ry \end{aligned}$$

Por tanto,

$$\begin{aligned} d &= bx + (a - bp)y \\ &= ay + b(x - qy) \end{aligned}$$

Luego,

$$\text{bezout}(a, b) = (y, x - qy)$$

La definición de `bezout` es

```
bezout :: Integer -> Integer -> (Integer, Integer)
bezout _ 0 = (1,0)
bezout _ 1 = (0,1)
bezout a b = (y, x-q*y)
    where (x,y) = bezout b r
          (q,r) = quotRem a b
```

Ejercicio 17.7.2. *Comprobar con QuickCheck que si* `a` *y* `b` *son positivos 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`*.*

Solución: 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

```
ghci> quickCheck prop_Bezout
OK, passed 100 tests.
```

17.8. Distancia entre dos conjuntos de números

Ejercicio 17.8.1. *El enunciado del problema 1 de la Olimpiada Iberoamericana de Matemática Universitaria del 2006 es el siguiente:*

> *Sean m y n números enteros mayores que 1. Se definen los conjuntos $P(m) = \{\frac{1}{m}, \frac{2}{m}, \ldots, \frac{m-1}{m}\}$ y $P(n) = \{\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}\}$. Encontrar la distancia entre $P(m)$ y $P(n)$, que se define como* $\min\{|a-b| : a \in P(m), b \in P(n)\}$.

Definir la función

```
distancia :: Float -> Float -> Float
```

tal que `(distancia m n)` *es la distancia entre $P(m)$ y $P(n)$. Por ejemplo,*

```
distancia 2 7 == 7.142857e-2
distancia 2 8 == 0.0
```

Solución:

```
distancia :: Float -> Float -> Float
distancia m n =
    minimum [abs (i/m - j/n) | i <- [1..m-1], j <- [1..n-1]]
```

17.9. Expresables como suma de números consecutivos

El enunciado del problema 580[2] de "Números y algo más..." es el siguiente:

> ¿Cuál es el menor número que puede expresarse como la suma de 9, 10 y 11 números consecutivos?

A lo largo de los distintos apartados de este ejercicio se resolverá el problema.

Ejercicio 17.9.1. *Definir la función*

```
consecutivosConSuma :: Int -> Int -> [[Int]]
```

tal que `(consecutivosConSuma x n)` *es la lista de listas de* `n` *números consecutivos cuya suma es* `x`*. Por ejemplo,*

```
consecutivosConSuma 12 3  ==  [[3,4,5]]
consecutivosConSuma 10 3  ==  []
```

Solución:

[2] http://goo.gl/1K3t7

```
consecutivosConSuma :: Int -> Int -> [[Int]]
consecutivosConSuma x n =
    [[y..y+n-1] | y <- [1..x], sum [y..y+n-1] == x]
```

Se puede hacer una definición sin búsqueda, ya que por la fórmula de la suma de progresiones aritméticas, la expresión `sum [y..y+n-1] == x` se reduce a

$$\frac{(y+(y+n-1))n}{2} = x$$

De donde se puede despejar la y, ya que

$$2yn + n^2 - n = 2x$$
$$y = \frac{2x-n^2+n}{2n}$$

De la anterior anterior se obtiene la siguiente definición de `consecutivosConSuma` que no utiliza búsqueda.

```
consecutivosConSuma' :: Int -> Int -> [[Int]]
consecutivosConSuma' x n
    | z >= 0 && mod z (2*n) == 0 = [[y..y+n-1]]
    | otherwise                  = []
    where z = 2*x-n^2+n
          y = div z (2*n)
```

Ejercicio 17.9.2. *Definir la función*

```
esSuma :: Int -> Int -> Bool
```

tal que `(esSuma x n)` *se verifica si* `x` *es la suma de* `n` *números naturales consecutivos. Por ejemplo,*

```
esSuma 12 3  ==  True
esSuma 10 3  ==  False
```

Solución:

```
esSuma :: Int -> Int -> Bool
esSuma x n = consecutivosConSuma x n /= []
```

También puede definirse directamente sin necesidad de `consecutivosConSuma` como se muestra a continuación.

```
esSuma' :: Int -> Int -> Bool
esSuma' x n = or [sum [y..y+n-1] == x | y <- [1..x]]
```

Ejercicio 17.9.3. *Definir la función*

```
menorQueEsSuma :: [Int] -> Int
```

tal que `(menorQueEsSuma ns)` *es el menor número que puede expresarse como suma de tantos números consecutivos como indica* `ns`*. Por ejemplo,*

```
menorQueEsSuma [3,4] == 18
```

Lo que indica que 18 es el menor número se puede escribir como suma de 3 y de 4 números consecutivos. En este caso, las sumas son 18 = 5+6+7 y 18 = 3+4+5+6.

Solución:

```
menorQueEsSuma :: [Int] -> Int
menorQueEsSuma ns =
    head [x | x <- [1..], and [esSuma x n | n <- ns]]
```

Ejercicio 17.9.4. *Usando la función menorQueEsSuma calcular el menor número que puede expresarse como la suma de 9, 10 y 11 números consecutivos.*

Solución: La solución es

```
ghci> menorQueEsSuma [9,10,11]
495
```

17.10. Solución de una ecuación diofántica

Ejercicio 17.10.1. *En este ejercicio vamos a comprobar que la ecuación diofántica*

$$\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n} = 1$$

tiene solución; es decir, que para todo $n \geq 1$ *se puede construir una lista de números enteros de longitud n tal que la suma de sus inversos es 1. Para ello, basta observar que si* $[x_1, x_2, \ldots, x_n]$ *es una solución, entonces* $[2, 2x_1, 2x_2, \ldots.., 2x_n]$ *también lo es. Definir la función* `solucion` *tal que* `(solucion n)` *es la solución de longitud* `n` *construida mediante el método anterior. Por ejemplo,*

```
solucion 1  ==  [1]
solucion 2  ==  [2,2]
solucion 3  ==  [2,4,4]
solucion 4  ==  [2,4,8,8]
solucion 5  ==  [2,4,8,16,16]
```

Solución:

```
solucion 1 = [1]
solucion n = 2 : [2*x | x <- solucion (n-1)]
```

Ejercicio 17.10.2. *Definir la función* `esSolucion` *tal que* `(esSolucion xs)` *se verifica si la suma de los inversos de* `xs` *es 1. Por ejemplo,*

```
esSolucion [4,2,4]       ==  True
esSolucion [2,3,4]       ==  False
esSolucion (solucion 5)  ==  True
```

Solución:

```
esSolucion xs = sum [1/x | x <- xs] == 1
```

Capítulo 18

El 2011 y los números primos

Cada comienzo de año se suelen buscar propiedades numéricas del número del año. En el 2011 se han buscado propiedades que relacionan el 2011 y los números primos. En este ejercicio vamos a realizar la búsqueda de dichas propiedades con Haskell.

Nota. Se usará la librería de `Listas`

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

18.1. La criba de Eratótenes

La criba de Eratótenes es un método para calcular números primos. Se comienza escribiendo todos los números desde 2 hasta (supongamos) 100. El primer número (el 2) es primo. Ahora eliminamos todos los múltiplos de 2. El primero de los números restantes (el 3) también es primo. Ahora eliminamos todos los múltiplos de 3. El primero de los números restantes (el 5) también es primo ... y así sucesivamente. Cuando no quedan números, se han encontrado todos los números primos en el rango fijado.

Ejercicio 18.1.1. *Definir, por comprensión, la función*

```
elimina :: Int -> [Int] -> [Int]
```

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]
```

Solución:

```
elimina :: Int -> [Int] -> [Int]
elimina n xs = [ x | x <- xs, x `rem` n /= 0 ]
```

Ejercicio 18.1.2. *Definir, por recursión, la función*

```
eliminaR :: Int -> [Int] -> [Int]
```

tal que `(eliminaR n xs)` *es la lista obtenida eliminando en la lista* `xs` *los múltiplos de* `n`*. Por ejemplo,*

```
eliminaR 3 [2,3,8,9,5,6,7] == [2,8,5,7]
```

Solución:

```
eliminaR :: Int -> [Int] -> [Int]
eliminaR n [] = []
eliminaR n (x:xs) | rem x n == 0 = eliminaR n xs
                  | otherwise    = x : eliminaR n xs
```

Ejercicio 18.1.3. *Definir, por plegado, la función*

```
eliminaP :: Int -> [Int] -> [Int]
```

tal que `(eliminaP n xs)` *es la lista obtenida eliminando en la lista* `xs` *los múltiplos de* `n`*. Por ejemplo,*

```
eliminaP 3 [2,3,8,9,5,6,7] == [2,8,5,7]
```

Solución:

```
eliminaP :: Int -> [Int] -> [Int]
eliminaP n = foldr f []
             where f x y | rem x n == 0 = y
                         | otherwise    = x:y
```

Ejercicio 18.1.4. *Definir la función*

```
criba :: [Int] -> [Int]
```

tal que `(criba xs)` *es la lista obtenida cribando la lista* `xs` *con el método descrito anteriormente. 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]
```

Solución:

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

Ejercicio 18.1.5. *Definir la función*

```
primos :: [Int]
```

cuyo valor es la lista de los números primos. Por ejemplo,

```
take 10 primos == [2,3,5,7,11,13,17,19,23,29]
```

Solución:

```
primos :: [Int]
primos = criba [2..]
```

Ejercicio 18.1.6. *Definir la función*

```
esPrimo :: Int -> Bool
```

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

```
esPrimo 7  ==  True
esPrimo 9  ==  False
```

Solución:

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

18.2. 2011 es primo

Ejercicio 18.2.1. *Comprobar que 2011 es primo.*

Solución: La comprobación es

```
ghci> esPrimo 2011
True
```

18.3. Primera propiedad del 2011

Ejercicio 18.3.1. *Definir la función*

```
prefijosConSuma :: [Int] -> Int -> [[Int]]
```

tal que `(prefijosConSuma xs n)` *es la lista de los prefijos de* `xs` *cuya suma es* `n`*. Por ejemplo,*

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

Solución:

```
prefijosConSuma :: [Int] -> Int -> [[Int]]
prefijosConSuma [] 0 = [[]]
prefijosConSuma [] n = []
prefijosConSuma (x:xs) n
    | x < n  = [x:ys | ys <- prefijosConSuma xs (n-x)]
    | x == n = [[x]]
    | x > n  = []
```

Ejercicio 18.3.2. *Definir la función*

```
consecutivosConSuma :: [Int] -> Int -> [[Int]]
```

tal que (`consecutivosConSuma xs n`) *es la lista de los elementos consecutivos de* `xs` *cuya suma es* `n`. *Por ejemplo,*

```
consecutivosConSuma [1..10] 9 == [[2,3,4],[4,5],[9]]
```

Solución:

```
consecutivosConSuma :: [Int] -> Int -> [[Int]]
consecutivosConSuma [] 0 = [[]]
consecutivosConSuma [] n = []
consecutivosConSuma (x:xs) n =
    (prefijosConSuma (x:xs) n) ++ (consecutivosConSuma xs n)
```

Ejercicio 18.3.3. *Definir la función*

```
primosConsecutivosConSuma :: Int -> [[Int]]
```

tal que (`primosConsecutivosConSuma n`) *es la lista de los números primos consecutivos cuya suma es* `n`. *Por ejemplo,*

```
ghci> primosConsecutivosConSuma 41
[[2,3,5,7,11,13],[11,13,17],[41]]
```

Solución:

```
primosConsecutivosConSuma :: Int -> [[Int]]
primosConsecutivosConSuma n =
    consecutivosConSuma (takeWhile (<=n) primos) n
```

Ejercicio 18.3.4. *Calcular las descomposiciones de 2011 como sumas de primos consecutivos.*

Solución: El cálculo es

```
ghci> primosConsecutivosConSuma 2011
[[157,163,167,173,179,181,191,193,197,199,211],[661,673,677],[2011]]
```

Ejercicio 18.3.5. *Definir la función*

```
propiedad1 :: Int -> Bool
```

tal que `(propiedad1 n)` *se verifica si* `n` *sólo se puede expresar como sumas de 1, 3 y 11 primos consecutivos. Por ejemplo,*

```
propiedad1 2011  ==  True
propiedad1 2010  ==  False
```

Solución:

```
propiedad1 :: Int -> Bool
propiedad1 n =
    sort (map length (primosConsecutivosConSuma n)) == [1,3,11]
```

Ejercicio 18.3.6. *Calcular los años hasta el 3000 que cumplen la propiedad1.*

Solución: El cálculo es

```
ghci> [n | n <- [1..3000], propiedad1 n]
[883,2011]
```

18.4. Segunda propiedad del 2011

Ejercicio 18.4.1. *Definir la función*

```
sumaCifras :: Int -> Int
```

tal que `(sumaCifras x)` *es la suma de las cifras del número* `x`*. Por ejemplo,*

```
sumaCifras 254 == 11
```

Solución:

```
sumaCifras :: Int -> Int
sumaCifras x = sum [read [y] | y <- show x]
```

Ejercicio 18.4.2. *Definir, por comprensión, la función*

```
sumaCifrasLista :: [Int] -> Int
```

tal que `(sumaCifrasLista xs)` *es la suma de las cifras de la lista de números* `xs`*. Por ejemplo,*

```
sumaCifrasLista [254, 61] == 18
```

Solución:

```
sumaCifrasLista :: [Int] -> Int
sumaCifrasLista xs = sum [sumaCifras y | y <- xs]
```

Ejercicio 18.4.3. *Definir, por recursión, la función*

```
sumaCifrasListaR :: [Int] -> Int
```

tal que `(sumaCifrasListaR xs)` *es la suma de las cifras de la lista de números* `xs`*. Por ejemplo,*

```
sumaCifrasListaR [254, 61] == 18
```

Solución:

```
sumaCifrasListaR :: [Int] -> Int
sumaCifrasListaR [] = 0
sumaCifrasListaR (x:xs) = sumaCifras x + sumaCifrasListaR xs
```

Ejercicio 18.4.4. *Definir, por plegado, la función*

```
sumaCifrasListaP :: [Int] -> Int
```

tal que `(sumaCifrasListaP xs)` *es la suma de las cifras de la lista de números* `xs`*. Por ejemplo,*

```
sumaCifrasListaP [254, 61] == 18
```

Solución:

```
sumaCifrasListaP :: [Int] -> Int
sumaCifrasListaP = foldr f 0
                   where f x y = sumaCifras x + y
```

Ejercicio 18.4.5. *Definir la función*

```
propiedad2 :: Int -> Bool
```

tal que `(propiedad2 n)` *se verifica si* `n` *puede expresarse como suma de 11 primos consecutivos y la suma de las cifras de los 11 sumandos es un número primo. Por ejemplo,*

```
propiedad2 2011  ==  True
propiedad2 2000  ==  False
```

Solución:

```
propiedad2 :: Int -> Bool
propiedad2 n = [xs | xs <- primosConsecutivosConSuma n,
                     length xs == 11,
                     esPrimo (sumaCifrasLista xs)]
               /= []
```

Ejercicio 18.4.6. *Calcular el primer año que cumple la* `propiedad1` *y la* `propiedad2`*.*

Solución: El cálculo es

```
ghci> head [n | n <- [1..], propiedad1 n, propiedad2 n]
2011
```

18.5. Tercera propiedad del 2011

Ejercicio 18.5.1. *Definir la función*

```
propiedad3 :: Int -> Bool
```

tal que `(propiedad3 n)` *se verifica si* `n` *puede expresarse como suma de tantos números primos consecutivos como indican sus dos últimas cifras. Por ejemplo,*

```
propiedad3 2011  ==  True
propiedad3 2000  ==  False
```

Solución:

```
propiedad3 :: Int -> Bool
propiedad3 n = [xs | xs <- primosConsecutivosConSuma n,
                     length xs == mod n 100]
               /= []
```

Ejercicio 18.5.2. *Calcular el primer año que cumple la* `propiedad1` *y la* `propiedad3`*.*

Solución: El cálculo es

```
ghci> head [n | n <- [1..], propiedad1 n, propiedad3 n]
2011
```

Nota. Hemos comprobado que 2011 es el menor número que cumple las propiedades 1 y 2 y también es el menor número que cumple las propiedades 1 y 3.

Capítulo 19

Combinatoria

El objetivo de este capítulo es estudiar la generación y el número de las principales operaciones de la combinatoria.

Contenido

19.1 Reconocimiento y generación de subconjuntos 359
19.2 Permutaciones 361
19.3 Combinaciones sin repetición 364
19.4 Combinaciones con repetición 367
19.5 Variaciones sin repetición 369
19.6 Variaciones con repetición 370
19.7 El triángulo de Pascal 372

19.1. Reconocimiento y generación de subconjuntos

Ejercicio 19.1.1. *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
```

Solución:

```
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto []     _ = True
subconjunto (x:xs) ys = elem x ys && subconjunto xs ys
```

Ejercicio 19.1.2. *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
```

Solución:

```
subconjunto' :: Eq a => [a] -> [a] -> Bool
subconjunto' xs ys = all (`elem` ys) xs
```

Ejercicio 19.1.3. *Comprobar con QuickCheck que las funciones* `subconjunto` *y* `subconjunto'` *son equivalentes.*

Solución: 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 19.1.4. *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
```

Solución:

```
igualConjunto :: Eq a => [a] -> [a] -> Bool
igualConjunto xs ys = subconjunto xs ys && subconjunto ys xs
```

Ejercicio 19.1.5. *Definir la función*

```
subconjuntos :: [a] -> [[a]]
```

tal que `(subconjuntos xs)` *es la lista de las 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], []]
```

Solución:

```
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
```

19.2. Permutaciones

Ejercicio 19.2.1. *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]]
```

Solución:

```
intercala :: a -> [a] -> [[a]]
intercala x []     = [[x]]
intercala x (y:ys) = (x:y:ys) : [y:zs | zs <- intercala x ys]
```

Ejercicio 19.2.2. *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"]
```

Solución:

```
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat [intercala x ys | ys <- permutaciones xs]
```

Ejercicio 19.2.3. *Definir la función*

```
permutacionesN :: Int -> [[Int]]
```

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]]
```

Solución:

```
permutacionesN :: Int -> [[Int]]
permutacionesN n = permutaciones [1..n]
```

Ejercicio 19.2.4. *Definir, usando permutacionesN, la función*

```
numeroPermutacionesN :: Int -> Int
```

tal que `(numeroPermutacionesN n)` *es el número de permutaciones de un conjunto con* `n` *elementos. Por ejemplo,*

```
numeroPermutacionesN 3  ==  6
numeroPermutacionesN 4  ==  24
```

Solución:

```
numeroPermutacionesN :: Int -> Int
numeroPermutacionesN = length . permutacionesN
```

Ejercicio 19.2.5. *Definir la función*

```
fact :: Int -> Int
```

tal que (`fact n`) *es el factorial de* `n`. *Por ejemplo,*

```
fact 3 == 6
```

Solución:

```
fact :: Int -> Int
fact n = product [1..n]
```

Ejercicio 19.2.6. *Definir, usando* `fact`, *la función*

```
numeroPermutacionesN' :: Int -> Int
```

tal que (`numeroPermutacionesN' n`) *es el número de permutaciones de un conjunto con* `n` *elementos. Por ejemplo,*

```
numeroPermutacionesN' 3  ==  6
numeroPermutacionesN' 4  ==  24
```

Solución:

```
numeroPermutacionesN' :: Int -> Int
numeroPermutacionesN' = fact
```

Ejercicio 19.2.7. *Definir la función*

```
prop_numeroPermutacionesN :: Int -> 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
```

Solución:

```
prop_numeroPermutacionesN :: Int -> Bool
prop_numeroPermutacionesN n =
    and [numeroPermutacionesN x == numeroPermutacionesN' x | x <- [1..n]]
```

19.3. Combinaciones sin repetición

Ejercicio 19.3.1. *Definir, por recursión, la función*

```
combinaciones :: Int -> [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"]
```

Solución:

```
combinaciones :: Int -> [a] -> [[a]]
combinaciones 0 _          = [[]]
combinaciones _ []         = []
combinaciones k (x:xs) =
    [x:ys | ys <- combinaciones (k-1) xs] ++ combinaciones k xs
```

Ejercicio 19.3.2. *Definir, usando* `subconjuntos`*, la función*

```
combinaciones' :: Int -> [a] -> [[a]]
```

tal que `combinaciones'` *sea equivalente a* `combinaciones`*.*

Solución:

```
combinaciones' :: Int -> [a] -> [[a]]
combinaciones' n xs =
    [ys | ys <- subconjuntos xs, length ys == n]
```

Ejercicio 19.3.3. *Comparar la eficiencia de* `combinaciones` *y* `combinaciones'` *y decidir cuál es la más eficiente.*

Solución: 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)
```

Ejercicio 19.3.4. *Definir la función*

```
combinacionesN :: Int -> Int -> [[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]]
```

Solución:

```
combinacionesN :: Int -> Int -> [[Int]]
combinacionesN n k = combinaciones k [1..n]
```

Ejercicio 19.3.5. *Definir, usando* `combinacionesN`*, la función*

```
numeroCombinaciones :: Int -> Int -> Int
```

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
```

Solución:

```
numeroCombinaciones :: Int -> Int -> Int
numeroCombinaciones n k = length (combinacionesN n k)
```

Puede definirse por composición

```
numeroCombinaciones_2 :: Int -> Int -> Int
numeroCombinaciones_2 = (length .) . 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 :: Int -> Int -> Int
numeroCombinaciones_3 = length .: combinacionesN
```

Ejercicio 19.3.6. *Definir la función*

```
comb :: Int -> Int -> Int
```

tal que `(comb n k)` *es el número combinatorio* `n` *sobre* `k`*; es decir,* `(comb n k)` $= \frac{n!}{k!(n-k)!}$*. Por ejemplo,*

```
comb 4 2  ==  6
comb 4 3  ==  4
```

Solución:

```
comb :: Int -> Int -> Int
comb n k = (fact n) `div` ((fact k) * (fact (n-k)))
```

Ejercicio 19.3.7. *Definir, usando* `comb`*, la función*

```
numeroCombinaciones' :: Int -> Int -> Int
```

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
```

Solución:

```
numeroCombinaciones' :: Int -> Int -> Int
numeroCombinaciones' = comb
```

Ejercicio 19.3.8. *Definir la función*

```
prop_numeroCombinaciones :: Int -> 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
```

Solución:

```
prop_numeroCombinaciones :: Int -> Bool
prop_numeroCombinaciones n =
    and [numeroCombinaciones n k == numeroCombinaciones' n k | k <- [1..n]]
```

19.4. Combinaciones con repetición

Ejercicio 19.4.1. *Definir la función*

```
combinacionesR :: Int -> [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"]
```

Solución:

```
combinacionesR :: Int -> [a] -> [[a]]
combinacionesR _ [] = []
combinacionesR 0 _  = [[]]
combinacionesR k (x:xs) =
    [x:ys | ys <- combinacionesR (k-1) (x:xs)] ++ combinacionesR k xs
```

Ejercicio 19.4.2. *Definir la función*

```
combinacionesRN :: Int -> Int -> [[Int]]
```

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]]
```

Solución:

```
combinacionesRN :: Int -> Int -> [[Int]]
combinacionesRN n k = combinacionesR k [1..n]
```

Ejercicio 19.4.3. *Definir, usando combinacionesRN, la función*

```
numeroCombinacionesR :: Int -> Int -> Int
```

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
```

Solución:

```
numeroCombinacionesR :: Int -> Int -> Int
numeroCombinacionesR n k = length (combinacionesRN n k)
```

Ejercicio 19.4.4. *Definir, usando* `comb`*, la función*

```
numeroCombinacionesR' :: Int -> Int -> Int
```

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
```

Solución:

```
numeroCombinacionesR' :: Int -> Int -> Int
numeroCombinacionesR' n k = comb (n+k-1) k
```

Ejercicio 19.4.5. *Definir la función*

```
prop_numeroCombinacionesR :: Int -> 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
```

Solución:

```
prop_numeroCombinacionesR :: Int -> Bool
prop_numeroCombinacionesR n =
  and [numeroCombinacionesR n k == numeroCombinacionesR' n k |
       k <- [1..n]]
```

19.5. Variaciones sin repetición

Ejercicio 19.5.1. *Definir la función*

```
variaciones :: Int -> [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"]
```

Solución:

```
variaciones :: Int -> [a] -> [[a]]
variaciones k xs =
    concat (map permutaciones (combinaciones k xs))
```

Ejercicio 19.5.2. *Definir la función*

```
variacionesN :: Int -> Int -> [[Int]]
```

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]]
```

Solución:

```
variacionesN :: Int -> Int -> [[Int]]
variacionesN n k = variaciones k [1..n]
```

Ejercicio 19.5.3. *Definir, usando* `variacionesN`*, la función*

```
numeroVariaciones :: Int -> Int -> Int
```

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
```

Solución:

```
numeroVariaciones :: Int -> Int -> Int
numeroVariaciones n k = length (variacionesN n k)
```

Ejercicio 19.5.4. *Definir, usando* `product`*, la función*

```
numeroVariaciones' :: Int -> Int -> Int
```

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
```

Solución:

```
numeroVariaciones' :: Int -> Int -> Int
numeroVariaciones' n k = product [(n-k+1)..n]
```

Ejercicio 19.5.5. *Definir la función*

```
prop_numeroVariaciones :: Int -> Bool
```

tal que `(prop_numeroVariaciones n)` *se verifica si las funciones* `numeroVariaciones` *y* `numeroVariacior` *son equivalentes para los* `n` *primeros números y todo* `k` *entre* `1` *y* `n`*. Por ejemplo,*

```
prop_numeroVariaciones 5 == True
```

Solución:

```
prop_numeroVariaciones :: Int -> Bool
prop_numeroVariaciones n =
    and [numeroVariaciones n k == numeroVariaciones' n k | k <- [1..n]]
```

19.6. Variaciones con repetición

Ejercicio 19.6.1. *Definir la función*

```
variacionesR :: Int -> [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"]
```

Solución:

```
variacionesR :: Int -> [a] -> [[a]]
variacionesR _ [] = [[]]
variacionesR 0 _  = [[]]
variacionesR k xs =
    [z:ys | z <- xs, ys <- variacionesR (k-1) xs]
```

Ejercicio 19.6.2. *Definir la función*

```
variacionesRN :: Int -> Int -> [[Int]]
```

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]]
```

Solución:

```
variacionesRN :: Int -> Int -> [[Int]]
variacionesRN n k = variacionesR k [1..n]
```

Ejercicio 19.6.3. *Definir, usando* `variacionesR`*, la función*

```
numeroVariacionesR :: Int -> Int -> Int
```

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
```

Solución:

```
numeroVariacionesR :: Int -> Int -> Int
numeroVariacionesR n k = length (variacionesRN n k)
```

Ejercicio 19.6.4. *Definir, usando* `(^)`*, la función*

```
numeroVariacionesR' :: Int -> Int -> Int
```

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
```

Solución:

```
numeroVariacionesR' :: Int -> Int -> Int
numeroVariacionesR' n k = n^k
```

Ejercicio 19.6.5. *Definir la función*

```
prop_numeroVariacionesR :: Int -> 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
```

Solución:

```
prop_numeroVariacionesR :: Int -> Bool
prop_numeroVariacionesR n =
    and [numeroVariacionesR n k == numeroVariacionesR' n k |
         k <- [1..n]]
```

19.7. El triángulo de Pascal

Ejercicio 19.7.1. *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 :: Int -> [Int]
```

tal que `(pascal n)` *es la* `n`*–ésima fila del triángulo de Pascal. Por ejemplo,*

```
pascal 6 == [1,5,10,10,5,1]
```

Solución:

```
pascal :: Int -> [Int]
pascal 1 = [1]
pascal n = [1] ++ [x+y | (x,y) <- pares (pascal (n-1))] ++ [1]
```

donde `(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 como se expresa en

```
prop_pares :: [Int] -> Bool
prop_pares xs =
    pares xs == pares' xs
```

y se comprueba con

```
ghci> quickCheck prop_pares
+++ OK, passed 100 tests.
```

Ejercicio 19.7.2. *Comprobar con QuickCheck, que la fila n-ésima del triángulo de Pascal tiene n elementos.*

Solución: La propiedad es

```
prop_Pascal :: Int -> Property
prop_Pascal n =
    n >= 1 ==> length (pascal n) == n
```

La comprobación es

```
ghci> quickCheck prop_Pascal
OK, passed 100 tests.
```

Ejercicio 19.7.3. *Comprobar con QuickCheck, que la suma de los elementos de la fila n–ésima del triángulo de Pascal es igual a* 2^{n-1}.

Solución: La propiedad es

```
prop_sumaPascal :: Int -> Property
prop_sumaPascal n =
    n >= 1 ==> sum (pascal n) == 2^(n-1)
```

La comprobación es

```
ghci> quickCheck prop_sumaPascal
OK, passed 100 tests.
```

Ejercicio 19.7.4. *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)`.

Solución: La propiedad es

```
prop_Combinaciones :: Int -> 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.
```

Capítulo 20

Cálculo numérico

Contenido

20.1 Diferenciación numérica 375
20.2 Cálculo de la raíz cuadrada mediante el método de Herón 377
20.3 Cálculo de los ceros de una función por el método de Newton 379
20.4 Cálculo de funciones inversas 380

Nota. En este capítulo se usa la librería `QuickCheck`.

```
import Test.QuickCheck
```

20.1. Diferenciación numérica

Ejercicio 20.1.1. *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
```

Solución:

```
derivada :: Double -> (Double -> Double) -> Double -> Double
derivada a f x = (f(x+a)-f(x))/a
```

Ejercicio 20.1.2. *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.*
- `(derivadauperBurda 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
```

Solución:

```
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 20.1.3. *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
```

Solución:

```
derivadaFinaDelSeno :: Double -> Double
derivadaFinaDelSeno = derivadaFina sin
```

20.2. Cálculo de la raíz cuadrada mediante el método de Herón

En los ejercicios de esta sección 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 $\frac{y+\frac{x}{y}}{2}$ es una aproximación mejor.
- El límite de la sucesión definida por $x_0 = 1$, $x_{n+1} = \frac{x_n+\frac{x}{x_n}}{2}$ es la raíz cuadrada de x.

Ejercicio 20.2.1. *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. Por ejemplo,*

```
raiz 9 == 3.000000001396984
```

Solución:

```
raiz :: Double -> Double
raiz x = raiz' 1
    where raiz' y | aceptable y = y
                  | otherwise   = raiz' (mejora y)
          mejora y    = 0.5*(y+x/y)
          aceptable y = abs(y*y-x) < 0.00001
```

Ejercicio 20.2.2. *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
```

Solución:

```
infix 5 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs(x-y) < 0.001
```

Ejercicio 20.2.3. *Comprobar con QuickCheck que si* `x` *es positivo, entonces* `(raiz x)^2 ~= x`.

Solución: La propiedad es

```
prop_raiz :: Double -> Bool
prop_raiz x =
    (raiz x')^2 ~= x'
    where x' = abs x
```

La comprobación es

```
ghci> quickCheck prop_raiz
OK, passed 100 tests.
```

Ejercicio 20.2.4. *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: *La función* `until'` *es equivalente a la predefinida* `until`.

Solución:

```
until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f x  | p x       = x
              | otherwise = until' p f (f x)
```

Ejercicio 20.2.5. *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
```

Solución:

```
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 20.2.6. *Comprobar con QuickCheck que si* `x` *es positivo, entonces* `(raizI x)^2 ~= x`.

Solución: La propiedad es

```
prop_raizI :: Double -> Bool
prop_raizI x =
    (raizI x')^2 ~= x'
    where x' = abs x
```

La comprobación es

```
ghci> quickCheck prop_raizI
OK, passed 100 tests.
```

20.3. Cálculo de los ceros de una función por el método de Newton

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 $b - \frac{f(b)}{f'(b)}$ es una mejor aproximación.
- el límite de la sucesión x_n definida por $x_0 = 1$, $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ es un cero de f.

Ejercicio 20.3.1. *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
```

Solución:

```
puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCero' f 1
    where puntoCero' f x | aceptable x = x
                         | otherwise   = puntoCero' f (mejora x)
          mejora b    = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001
```

Ejercicio 20.3.2. *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
```

Solución:

```
puntoCeroI :: (Double -> Double) -> Double
puntoCeroI f = until aceptable mejora 1
    where mejora b    = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001
```

20.4. Cálculo de funciones inversas

En esta sección se usará la función `puntoCero` para definir la inversa de distintas funciones.

Ejercicio 20.4.1. *Definir, usando* `puntoCero`*, la función*

```
raizCuadrada :: Double -> Double
```

tal que `(raizCuadrada x)` *es la raíz cuadrada de* `x`*. Por ejemplo,*

```
raizCuadrada 9 == 3.000000002941184
```

Solución:

```
raizCuadrada :: Double -> Double
raizCuadrada a = puntoCero f
    where f x = x*x-a
```

Ejercicio 20.4.2. *Comprobar con QuickCheck que si* `x` *es positivo, entonces* `(raizCuadrada x)^2 ~= x`*.*

Solución: La propiedad es

```
prop_raizCuadrada :: Double -> Bool
prop_raizCuadrada x =
    (raizCuadrada x')^2 ~= x'
    where x' = abs x
```

La comprobación es

```
ghci> quickCheck prop_raizCuadrada
OK, passed 100 tests.
```

Ejercicio 20.4.3. *Definir, usando* `puntoCero`*, la función*

```
raizCubica :: Double -> Double
```

tal que `(raizCubica x)` *es la raíz cuadrada de* `x`*. Por ejemplo,*

```
raizCubica 27 == 3.0000000000196048
```

Solución:

```
raizCubica :: Double -> Double
raizCubica a = puntoCero f
    where f x = x*x*x-a
```

Ejercicio 20.4.4. *Comprobar con QuickCheck que si* `x` *es positivo, entonces* `(raizCubica x)^3 ~= x`*.*

Solución: La propiedad es

```
prop_raizCubica :: Double -> Bool
prop_raizCubica x =
    (raizCubica x)^3 ~= x
    where x' = abs x
```

La comprobación es

```
ghci> quickCheck prop_raizCubica
OK, passed 100 tests.
```

Ejercicio 20.4.5. *Definir, usando* `puntoCero`*, la función*

```
arcoseno :: Double -> Double
```

tal que `(arcoseno x)` *es el arcoseno de* `x`*. Por ejemplo,*

```
arcoseno 1 == 1.5665489428306574
```

Solución:

```
arcoseno :: Double -> Double
arcoseno a = puntoCero f
    where f x = sin x - a
```

Ejercicio 20.4.6. *Comprobar con QuickCheck que si* `x` *está entre 0 y 1, entonces* `sin (arcoseno x) ~= x`.

Solución: La propiedad es

```
prop_arcoseno :: Double -> Bool
prop_arcoseno x =
    sin (arcoseno x') ~= x'
    where x' = abs (x - fromIntegral (truncate x))
```

La comprobación es

```
ghci> quickCheck prop_arcoseno
OK, passed 100 tests.
```

Ejercicio 20.4.7. *Definir, usando* `puntoCero`*, la función*

```
arcocoseno :: Double -> Double
```

tal que `(arcoseno x)` *es el arcoseno de* `x`*. Por ejemplo,*

```
arcocoseno 0 == 1.5707963267949576
```

Solución:

```
arcocoseno :: Double -> Double
arcocoseno a = puntoCero f
    where f x = cos x - a
```

Ejercicio 20.4.8. *Comprobar con QuickCheck que si* `x` *está entre 0 y 1, entonces* `cos (arcocoseno x) ~= x`.

Solución: La propiedad es

```
prop_arcocoseno :: Double -> Bool
prop_arcocoseno x =
    cos (arcocoseno x') ~= x'
    where x' = abs (x - fromIntegral (truncate x))
```

La comprobación es

```
ghci> quickCheck prop_arcocoseno
OK, passed 100 tests.
```

Ejercicio 20.4.9. *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 == 3.000000002941184
```

Solución:

```
inversa :: (Double -> Double) -> Double -> Double
inversa g a = puntoCero f
    where f x = g x - a
```

Ejercicio 20.4.10. *Redefinir, usando* `inversa`*, las funciones* `raizCuadrada`*,* `raizCubica`*,* `arcoseno` *y* `arcocoseno`*.*

Solución:

```
raizCuadrada' = inversa (^2)
raizCubica'   = inversa (^3)
arcoseno'     = inversa sin
arcocoseno'   = inversa cos
```

Capítulo 21

Ecuación con factoriales

Contenido

21.1 Cálculo de factoriales 385
21.2 Decisión de si un número es un factorial 386
21.3 Inversa del factorial 386
21.4 Enumeración de los pares de números naturales 387
21.5 Solución de la ecuación con factoriales 388

El objetivo de esta relación de ejercicios es resolver la ecuación

$$a! \times b! = a! + b! + c!$$

donde a, b y c son números naturales.

Nota. Se usará la librería `QuickCheck`:

```
import Test.QuickCheck
```

21.1. Cálculo de factoriales

Ejercicio 21.1.1. *Definir la función*

```
factorial :: Integer -> Integer
```

tal que `(factorial n)` *es el factorial de* `n`*. Por ejemplo,*

```
factorial 5 == 120
```

Solución:

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

Ejercicio 21.1.2. *Definir la constante*

```
factoriales :: [Integer]
```

tal que `factoriales` *es la lista de los factoriales de los números naturales. Por ejemplo,*

```
take 7 factoriales == [1,1,2,6,24,120,720]
```

Solución:

```
factoriales :: [Integer]
factoriales = [factorial n | n <- [0..]]
```

21.2. Decisión de si un número es un factorial

Ejercicio 21.2.1. *Definir, usando factoriales, la función*

```
esFactorial :: Integer -> Bool
```

tal que `(esFactorial n)` *se verifica si existe un número natural* `m` *tal que* `n` *es* `m!`*. Por ejemplo,*

```
esFactorial 120  ==  True
esFactorial  20  ==  False
```

Solución:

```
esFactorial :: Integer -> Bool
esFactorial n = n == head (dropWhile (<n) factoriales)
```

21.3. Inversa del factorial

Ejercicio 21.3.1. *Definir la constante*

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

tal que `posicionesFactoriales` *es la lista de los factoriales con su posición. Por ejemplo,*

```
ghci> take 7 posicionesFactoriales
[(0,1),(1,1),(2,2),(3,6),(4,24),(5,120),(6,720)]
```

Solución:

```
posicionesFactoriales :: [(Integer,Integer)]
posicionesFactoriales = zip [0..] factoriales
```

Ejercicio 21.3.2. *Definir la función*

```
invFactorial :: Integer -> Maybe Integer
```

tal que `(invFactorial x)` *es* `(Just n)` *si el factorial de* `n` *es* `x` *y es* `Nothing`*, en caso contrario. Por ejemplo,*

```
invFactorial 120  == Just 5
invFactorial 20   == Nothing
```

Solución:

```
invFactorial :: Integer -> Maybe Integer
invFactorial x
    | esFactorial x = Just (head [n | (n,y) <- posicionesFactoriales, y==x])
    | otherwise     = Nothing
```

21.4. Enumeración de los pares de números naturales

Ejercicio 21.4.1. *Definir la constante*

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

tal que `pares` *es la lista de todos los pares de números naturales. Por ejemplo,*

```
ghci> take 11 pares
[(0,0),(0,1),(1,1),(0,2),(1,2),(2,2),(0,3),(1,3),(2,3),(3,3),(0,4)]
```

Solución:

```
pares :: [(Integer,Integer)]
pares = [(x,y) | y <- [0..], x <- [0..y]]
```

21.5. Solución de la ecuación con factoriales

Ejercicio 21.5.1. *Definir la constante*

```
solucionFactoriales :: (Integer,Integer,Integer)
```

tal que `solucionFactoriales` *es una terna* `(a,b,c)` *que es una solución de la ecuación*

```
a! * b! = a! + b! + c!
```

Calcular el valor de `solucionFactoriales`*.*

Solución:

```
solucionFactoriales :: (Integer,Integer,Integer)
solucionFactoriales = (a,b,c)
    where (a,b)  = head [(x,y) | (x,y) <- pares,
                                 esFactorial (f x * f y - f x - f y)]
          f      = factorial
          Just c = invFactorial (f a * f b - f a - f b)
```

El cálculo es

```
ghci> solucionFactoriales
(3,3,4)
```

Ejercicio 21.5.2. *Comprobar con QuickCheck que* `solucionFactoriales` *es la única solución de la ecuación*

$$a! \times b! = a! + b! + c!$$

con a, b y c números naturales

Solución: La propiedad es

```
prop_solucionFactoriales :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales x y z =
    x >= 0 && y >= 0 && z >= 0 && (x,y,z) /= solucionFactoriales
    ==> not (f x * f y == f x + f y + f z)
    where f = factorial
```

La comprobación es

```
ghci> quickCheck prop_solucionFactoriales
*** Gave up! Passed only 86 tests.
```

También se puede expresar como

```
prop_solucionFactoriales' :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales' x y z =
    x >= 0 && y >= 0 && z >= 0 &&
    f x * f y == f x + f y + f z
    ==> (x,y,z) == solucionFactoriales
    where f = factorial
```

La comprobación es

```
ghci> quickCheck prop_solucionFactoriales
*** Gave up! Passed only 0 tests.
```

Nota. El ejercicio se basa en el artículo Ecuación con factoriales[1] del blog Gaussianos.

[1] http://gaussianos.com/ecuacion-con-factoriales

Capítulo 22

Cuadrados mágicos

Contenido

22.1 Reconocimiento de los cuadrados mágicos **392**
22.1.1 Traspuesta de una matriz 392
22.1.2 Suma de las filas de una matriz 393
22.1.3 Suma de las columnas de una matriz 393
22.1.4 Diagonal principal de una matriz 393
22.1.5 Diagonal secundaria de una matriz 394
22.1.6 Lista con todos los elementos iguales 394
22.1.7 Reconocimiento de matrices cuadradas 394
22.1.8 Elementos de una lista de listas 395
22.1.9 Eliminación de la primera ocurrencia de un elemento 395
22.1.10 Reconocimiento de permutaciones 395
22.1.11 Reconocimiento de cuadrados mágicos 396
22.2 Cálculo de los cuadrados mágicos **396**
22.2.1 Matriz cuadrada correspondiente a una lista de elementos . . . 396
22.2.2 Cálculo de cuadrados mágicos por permutaciones 397
22.2.3 Cálculo de los cuadradros mágicos mediante generación y poda 397

Una matriz cuadrada representa un cuadrado mágico de orden n si el conjunto de sus elementos es $\{1, 2, \ldots, n^2\}$ y las sumas de cada una de sus filas, columnas y dos diagonales principales coinciden. Por ejemplo,

$$\begin{pmatrix} 2 & 9 & 4 \\ 7 & 5 & 3 \\ 6 & 1 & 8 \end{pmatrix}$$

es un cuadrado mágico de orden 3, ya que el conjunto de sus elementos es $\{1,2,\ldots,9\}$ y 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 capítulo se definirán funciones cuyo objetivo es decidir si una matriz representa un cuadrado mágico y construirlos.

Nota. Se usan la siguiente librería

```
import Data.List
```

22.1. Reconocimiento de los cuadrados mágicos

22.1.1. Traspuesta de una matriz

Ejercicio 22.1.1. *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]]
```

Solución:

```
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
```

22.1.2. Suma de las filas de una matriz

Ejercicio 22.1.2. *Definir la función*

```
sumasDeFilas :: Num a => [[a]] -> [a]
```

tal que `(sumasDeFilas xss)` *es la lista de las sumas de las filas de la matriz* `xss`*. Por ejemplo,*

```
sumasDeFilas [[2,4,0],[7,1,3],[6,1,8]] == [6,11,15]
```

Solución:

```
sumasDeFilas :: Num a => [[a]] -> [a]
sumasDeFilas = map sum
```

22.1.3. Suma de las columnas de una matriz

Ejercicio 22.1.3. *Definir la función*

```
sumasDeColumnas :: Num a => [[a]] -> [a]
```

tal que `(sumasDeColumnas xss)` *es la lista de las sumas de las columnas de la matriz* `xss`*. Por ejemplo,*

```
sumasDeFilas [[2,4,0],[7,1,3],[6,1,8]] == [6,11,15]
```

Solución:

```
sumasDeColumnas :: Num a => [[a]] -> [a]
sumasDeColumnas = sumasDeFilas . traspuesta
```

22.1.4. Diagonal principal de una matriz

Ejercicio 22.1.4. *Definir la función*

```
diagonalPral :: [[a]] -> [a]
```

tal que `(diagonalPral m)` *es la diagonal principal de la matriz* `m`*. Por ejemplo,*

```
diagonalPral [[3,5,2],[4,7,1],[6,9,0]]  ==  [3,7,0]
diagonalPral [[3,5,2],[4,7,1]]          ==  [3,7]
```

Solución:

```
diagonalPral :: [[a]] -> [a]
diagonalPral ((x1:_):xs) = x1 : diagonalPral [tail x | x <- xs]
diagonalPral _ = []
```

22.1.5. Diagonal secundaria de una matriz

Ejercicio 22.1.5. *Definir la función*

```
diagonalSec :: [[a]] -> [a]
```

tal que `(diagonalSec m)` *es la diagonal secundaria de la matriz* `m` *Por ejemplo,*

```
diagonalSec [[3,5,2],[4,7,1],[6,9,0]]  ==  [6,7,2]
diagonalSec [[3,5,2],[4,7,1]]          ==  [4,5]
```

Solución:

```
diagonalSec :: [[a]] -> [a]
diagonalSec = diagonalPral . reverse
```

22.1.6. Lista con todos los elementos iguales

Ejercicio 22.1.6. *Definir la función*

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

tal que `(todosIguales xs)` *se verifica si todos los elementos de* `xs` *son iguales. Por ejemplo,*

```
todosIguales [2,2,2]  ==  True
todosIguales [2,3,2]  ==  False
```

Solución:

```
todosIguales :: Eq a => [a] -> Bool
todosIguales (x:y:ys) = x == y && todosIguales (y:ys)
todosIguales _ = True
```

22.1.7. Reconocimiento de matrices cuadradas

Ejercicio 22.1.7. *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
```

Solución:

```
matrizCuadrada :: [[Int]] -> Bool
matrizCuadrada xss =
    and [length xs == n | xs <- xss]
    where n = length xss
```

22.1.8. Elementos de una lista de listas

Ejercicio 22.1.8. *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]
```

Solución:

```
elementos :: [[a]] -> [a]
elementos = concat
```

22.1.9. Eliminación de la primera ocurrencia de un elemento

Ejercicio 22.1.9. *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]
```

Solución:

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

22.1.10. Reconocimiento de permutaciones

Ejercicio 22.1.10. *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
```

Solución:

```
esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     []     = True
esPermutacion []     (y:ys) = False
esPermutacion (x:xs) ys     = elem x ys && esPermutacion xs (borra x ys)
```

22.1.11. Reconocimiento de cuadrados mágicos

Ejercicio 22.1.11. *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
```

Solución:

```
cuadradoMagico xss =
    matrizCuadrada xss &&
    esPermutacion (elementos xss) [1..(length xss)^2]  &&
    todosIguales (sumasDeFilas xss ++
                  sumasDeColumnas xss ++
                  [sum (diagonalPral xss),
                   sum (diagonalSec xss)])
```

22.2. Cálculo de los cuadrados mágicos

22.2.1. Matriz cuadrada correspondiente a una lista de elementos

Ejercicio 22.2.1. *Definir la función*

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

tal que `(matriz n xs)` *es la matriz cuadrada de orden* $n \times n$ *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]]
```

Solución:

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

22.2.2. Cálculo de cuadrados mágicos por permutaciones

Ejercicio 22.2.2. *Definir la función*

```
cuadradosMagicos :: Int -> [[[Int]]]
```

tal que `(cuadradosMagicos n)` *es la lista de los cuadrados mágicos de orden* $n \times n$*. 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]]]
```

Solución:

```
cuadradosMagicos :: Int -> [[[Int]]]
cuadradosMagicos n =
    [m | xs <- permutations [1..n^2],
         let m = matriz n xs,
         cuadradoMagico m]
```

22.2.3. Cálculo de los cuadradros mágicos mediante generación y poda

Ejercicio 22.2.3. *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

```
cuadradosMagicos3 :: [[[Int]]]
```

tal que `cuadradosMagicos3` *es la lista de los cuadrados mágicos de orden 3 construidos usando el proceso anterior. Por ejemplo,*

```
ghci> take 2 cuadradosMagicos3
[[[2,7,6],[9,5,1],[4,3,8]],[[2,9,4],[7,5,3],[6,1,8]]]
```

Solución:

```
cuadradosMagicos3 :: [[[Int]]]
cuadradosMagicos3 =
    [[[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 22.2.4. *Comprobar que* `cuadradosMagicos3` *es el mismo conjunto que* `(cuadradosMagicos 3)`.

Solución: La comprobación es

```
ghci> esPermutacion cuadradosMagicos3 (cuadradosMagicos 3)
True
```

Ejercicio 22.2.5. *Comparar los tiempos utilizados en calcular* `cuadradosMagicos3` *y* `(cuadradosMagicos 3`

Solución: La comparación es

```
ghci> :set +s
ghci> cuadradosMagicos3
[[[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)
```

Capítulo 23

Enumeraciones de los números racionales

Contenido

23.1 Numeración de los racionales mediante representaciones hiperbinarias **402**
23.1.1 Lista de potencias de dos 402
23.1.2 Determinación si los dos primeros elementos son iguales a uno dado 402
23.1.3 Lista de las representaciones hiperbinarias de n 403
23.1.4 Número de representaciones hiperbinarias de n 403
23.1.5 Sucesiones hiperbinarias 404
23.2 Numeraciones mediante árboles de Calkin–Wilf **406**
23.2.1 Hijos de un nodo en el árbol de Calvin–Wilf 406
23.2.2 Niveles del árbol de Calvin–Wilf 407
23.2.3 Sucesión de Calvin–Wilf 408
23.3 Número de representaciones hiperbinarias mediante la función `fusc` . **408**
23.3.1 La función `fusc` 408

El objetivo de este caítulo es construir dos enumeraciones de los números racionales. Concretamente,

- una enumeración basada en las representaciones hiperbinarias y
- una enumeración basada en los los árboles de Calkin–Wilf.

También se incluye la comprobación de la igualdad de las dos sucesiones y una forma alternativa de calcular el número de representaciones hiperbinarias mediante la función fucs.

Esta relación se basa en los siguientes artículos:

- Gaussianos "Sorpresa sumando potencias de 2"[1]
- N. Calkin y H.S. Wilf "Recounting the rationals"[2]
- Wikipedia "Calkin–Wilf tree"[3]

Nota. Se usan las librerías `List` y `QuickCheck`:

```
import Data.List
import Test.QuickCheck
```

23.1. Numeración de los racionales mediante representaciones hiperbinarias

23.1.1. Lista de potencias de dos

Ejercicio 23.1.1. *Definir la constante*

```
potenciasDeDos :: [Integer]
```

tal que `potenciasDeDos` *es la lista de las potencias de 2. Por ejemplo,*

```
take 10 potenciasDeDos == [1,2,4,8,16,32,64,128,256,512]
```

Solución:

```
potenciasDeDos :: [Integer]
potenciasDeDos = [2^n | n <- [0..]]
```

23.1.2. Determinación si los dos primeros elementos son iguales a uno dado

Ejercicio 23.1.2. *Definir la función*

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

tal que `(empiezaConDos x ys)` *se verifica si los dos primeros elementos de* `ys` *son iguales a* `x`*. Por ejemplo,*

[1] http://gaussianos.com/sorpresa-sumando-potencias-de-2
[2] http://www.math.clemson.edu/~calkin/Papers/calkin_wilf_recounting_rationals.pdf
[3] http://en.wikipedia.org/wiki/Calkin-Wilf_tree

```
empiezaConDos 5 [5,5,3,7]  ==  True
empiezaConDos 5 [5,3,5,7]  ==  False
empiezaConDos 5 [5,5,5,7]  ==  True
```

Solución:

```
empiezaConDos x (y1:y2:ys) = y1 == x && y2 == x
empiezaConDos x _          = False
```

23.1.3. Lista de las representaciones hiperbinarias de *n*

Ejercicio 23.1.3. *Definir la función*

```
representacionesHB :: Integer -> [[Integer]]
```

tal que `(representacionesHB n)` *es la lista de las representaciones hiperbinarias del número* `n` *como suma de potencias de 2 donde cada sumando aparece como máximo 2 veces. Por ejemplo*

```
representacionesHB 5  ==  [[1,2,2],[1,4]]
representacionesHB 6  ==  [[1,1,2,2],[1,1,4],[2,4]]
```

Solución:

```
representacionesHB :: Integer -> [[Integer]]
representacionesHB n = representacionesHB' n potenciasDeDos
representacionesHB' n (x:xs)
    | n == 0    = [[]]
    | x == n    = [[x]]
    | x <  n    = [x:ys | ys <- representacionesHB' (n-x) (x:xs),
                          not (empiezaConDos x ys)] ++
                  representacionesHB' n xs
    | otherwise = []
```

23.1.4. Número de representaciones hiperbinarias de *n*

Ejercicio 23.1.4. *Definir la función*

```
nRepresentacionesHB :: Integer -> Integer
```

tal que `(nRepresentacionesHB n)` *es el número de las representaciones hiperbinarias del número* `n` *como suma de potencias de 2 donde cada sumando aparece como máximo 2 veces. Por ejemplo,*

```
ghci> [nRepresentacionesHB n | n <- [0..20]]
[1,1,2,1,3,2,3,1,4,3,5,2,5,3,4,1,5,4,7,3,8]
```

Solución:

```
nRepresentacionesHB :: Integer -> Integer
nRepresentacionesHB = genericLength . representacionesHB
```

23.1.5. Sucesiones hiperbinarias

Ejercicio 23.1.5. *Definir la función*

```
termino :: Integer -> (Integer,Integer)
```

tal que `(termino n)` *es el par formado por el número de representaciones hiperbinarias de* `n` *y de* `n+1` *(que se interpreta como su cociente). Por ejemplo,*

```
termino 4 == (3,2)
```

Solución:

```
termino :: Integer -> (Integer,Integer)
termino n = (nRepresentacionesHB n, nRepresentacionesHB (n+1))
```

Ejercicio 23.1.6. *Definir la función*

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

`sucesionHB` *es la la sucesión cuyo témino* `n`*–ésimo es* `(termino n)`*; es decir, el par formado por el número de representaciones hiperbinarias de* `n` *y de* `n+1`*. Por ejemplo,*

```
ghci> take 10 sucesionHB
[(1,1),(1,2),(2,1),(1,3),(3,2),(2,3),(3,1),(1,4),(4,3),(3,5)]
```

Solución:

```
sucesionHB :: [(Integer,Integer)]
sucesionHB = [termino n | n <- [0..]]
```

Ejercicio 23.1.7. *Comprobar con QuickCheck que, para todo* `n`*,* `(nRepresentacionesHB n)` *y* `(nRepresentacionesHB (n+1))` *son primos entre sí.*

Solución: La propiedad es

```
prop_irreducibles :: Integer -> Property
prop_irreducibles n =
    n >= 0 ==>
    gcd (nRepresentacionesHB n) (nRepresentacionesHB (n+1)) == 1
```

La comprobación es

```
ghci> quickCheck prop_irreducibles
+++ OK, passed 100 tests.
```

Ejercicio 23.1.8. *Comprobar con QuickCheck que todos los elementos de la* `sucesionHB` *son distintos.*

Solución: La propiedad es

```
prop_distintos :: Integer -> Integer -> Bool
prop_distintos n m =
    termino n' /= termino m'
    where n' = abs n
          m' = n' + abs m + 1
```

La comprobación es

```
ghci> quickCheck prop_distintos
+++ OK, passed 100 tests.
```

Ejercicio 23.1.9. *Definir la función*

```
contenido :: Integer -> Integer -> Bool
```

tal que `(contenido n)` *se verifica si la expresiones reducidas de todas las fracciones* `x/y`*, con* `x` *e* `y` *entre* `1` *y* `n`*, pertenecen a la* `sucesionHB`*. Por ejemplo,*

```
contenidos 5 == True
```

Solución:

```
contenido :: Integer -> Bool
contenido n =
    and [pertenece (reducida (x,y)) sucesionHB |
         x <- [1..n], y <- [1..n]]
    where pertenece x (y:ys) = x == y || pertenece x ys
          reducida (x,y) = (x `div` z, y `div` z)
              where z = gcd x y
```

Ejercicio 23.1.10. *Definir la función*

```
indice :: (Integer,Integer) -> Integer
```

tal que `(indice (a,b))` *es el índice del par* `(a,b)` *en la sucesión de los racionales. Por ejemplo,*

```
indice (3,2) == 4
```

Solución:

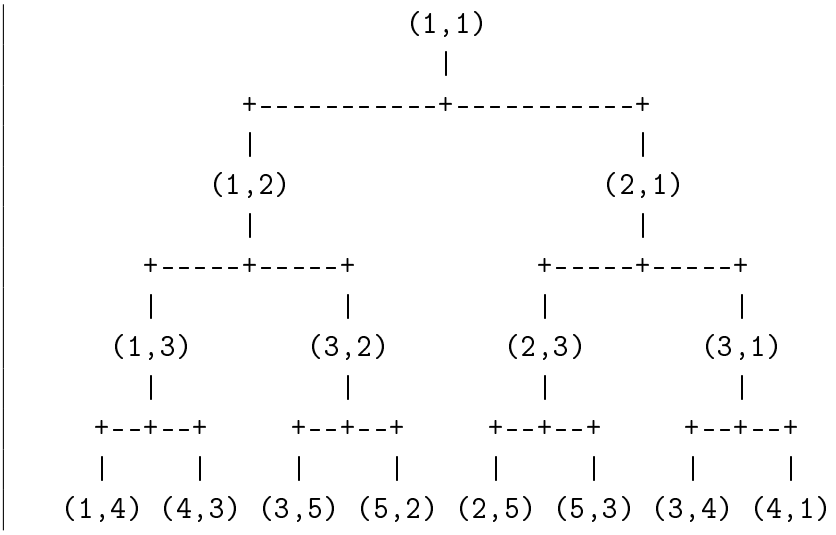
```
indice :: (Integer,Integer) -> Integer
indice (a,b) = head [n | (n,(x,y)) <- zip [0..] sucesionHB,
                         (x,y) == (a,b)]
```

23.2. Numeraciones mediante árboles de Calkin–Wilf

El *árbol de Calkin–Wilf* es el árbol definido por las siguientes reglas:

- El nodo raíz es el $(1,1)$
- Los hijos del nodo (x,y) son $(x,x+y)$ y $(x+y,y)$

Por ejemplo, los 4 primeros niveles del árbol de Calkin-Wilf son



23.2.1. Hijos de un nodo en el árbol de Calvin–Wilf

Ejercicio 23.2.1. *Definir la función*

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

tal que `(sucesores (x,y))` *es la lista de los hijos del par* `(x,y)` *en el árbol de Calkin–Wilf. Por ejemplo,*

```
sucesores (3,2) == [(3,5),(5,2)]
```

Solución:

```
sucesores :: (Integer,Integer) -> [(Integer,Integer)]
sucesores (x,y) = [(x,x+y),(x+y,y)]
```

Ejercicio 23.2.2. *Definir la función*

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

tal que `(siguiente xs)` *es la lista formada por los hijos de los elementos de* `xs` *en el árbol de Calkin–Wilf. Por ejemplo,*

```
ghci> siguiente [(1,3),(3,2),(2,3),(3,1)]
[(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]
```

Solución:

```
siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
siguiente xs = [p | x <- xs, p <- sucesores x]
```

23.2.2. Niveles del árbol de Calvin–Wilf

Ejercicio 23.2.3. *Definir la constante*

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

tal que `nivelesCalkinWilf` *es la lista de los niveles del árbol de Calkin–Wilf. Por ejemplo,*

```
ghci> take 4 nivelesCalkinWilf
[[(1,1)],
 [(1,2),(2,1)],
 [(1,3),(3,2),(2,3),(3,1)],
 [(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]]
```

Solución:

```
nivelesCalkinWilf :: [[(Integer,Integer)]]
nivelesCalkinWilf = iterate siguiente [(1,1)]
```

23.2.3. Sucesión de Calvin–Wilf

Ejercicio 23.2.4. *Definir la constante*

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

tal que `sucesionCalkinWilf` *es la lista correspondiente al recorrido en anchura del árbol de Calkin–Wilf. Por ejemplo,*

```
ghci> take 10 sucesionCalkinWilf
[(1,1),(1,2),(2,1),(1,3),(3,2),(2,3),(3,1),(1,4),(4,3),(3,5)]
```

Solución:

```
sucesionCalkinWilf :: [(Integer,Integer)]
sucesionCalkinWilf = concat nivelesCalkinWilf
```

Ejercicio 23.2.5. *Definir la función*

```
igual_sucesion_HB_CalkinWilf :: Int -> Bool
```

tal que `(igual_sucesion_HB_CalkinWilf n)` *se verifica si los* `n` *primeros términos de la sucesión HB son iguales que los de la sucesión de Calkin–Wilf. Por ejemplo,*

```
igual_sucesion_HB_CalkinWilf 20 == True
```

Solución:

```
igual_sucesion_HB_CalkinWilf :: Int -> Bool
igual_sucesion_HB_CalkinWilf n =
    take n sucesionCalkinWilf == take n sucesionHB
```

23.3. Número de representaciones hiperbinarias mediante la función `fusc`

23.3.1. La función `fusc`

Ejercicio 23.3.1. *Definir la función*

```
fusc :: Integer -> Integer
```

tal que

```
fusc(0)    = 1
fusc(2n+1) = fusc(n)
fusc(2n+2) = fusc(n+1)+fusc(n)
```

Por ejemplo,

```
fusc 4 == 3
```

Solución:

```
fusc :: Integer -> Integer
fusc 0 = 1
fusc  n | odd n     = fusc ((n-1) `div` 2)
        | otherwise = fusc(m+1) + fusc m
                      where m = (n-2) `div` 2
```

Ejercicio 23.3.2. *Comprobar con QuickCheck que, para todo* `n`*,* `(fusc n)` *es el número de las representaciones hiperbinarias del número* `n` *como suma de potencias de 2 donde cada sumando aparece como máximo 2 veces; es decir, que las funciones* `fusc` *y* `nRepresentacionesHB` *son equivalentes.*

Solución: La propiedad es

```
prop_fusc :: Integer -> Bool
prop_fusc n = nRepresentacionesHB n' == fusc n'
              where n' = abs n
```

La comprobación es

```
ghci> quickCheck prop_fusc
+++ OK, passed 100 tests.
```

Parte IV

Apéndices

Apéndice A

Resumen de funciones predefinidas de Haskell

1. `x + y` es la suma de `x` e `y`.
2. `x - y` es la resta de `x` e `y`.
3. `x / y` es el cociente de `x` entre `y`.
4. `x ^ y` es `x` elevado a `y`.
5. `x == y` se verifica si `x` es igual a `y`.
6. `x /= y` se verifica si `x` es distinto de `y`.
7. `x < y` se verifica si `x` es menor que `y`.
8. `x <= y` se verifica si `x` es menor o igual que `y`.
9. `x > y` se verifica si `x` es mayor que `y`.
10. `x >= y` se verifica si `x` es mayor o igual que `y`.
11. `x && y` es la conjunción de `x` e `y`.
12. `x || y` es la disyunción de `x` e `y`.
13. `x:ys` es la lista obtenida añadiendo `x` al principio de `ys`.
14. `xs ++ ys` es la concatenación de `xs` e `ys`.
15. `xs !! n` es el elemento `n`–ésimo de `xs`.
16. `f . g` es la composición de `f` y `g`.
17. `abs x` es el valor absoluto de `x`.
18. `and xs` es la conjunción de la lista de booleanos `xs`.
19. `ceiling x` es el menor entero no menor que `x`.
20. `chr n` es el carácter cuyo código ASCII es `n`.
21. `concat xss` es la concatenación de la lista de listas `xss`.
22. `const x y` es `x`.

23. `curry f` es la versión curryficada de la función `f`.
24. `div x y` es la división entera de `x` entre `y`.
25. `drop n xs` borra los `n` primeros elementos de `xs`.
26. `dropWhile p xs` borra el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
27. `elem x ys` se verifica si `x` pertenece a `ys`.
28. `even x` se verifica si `x` es par.
29. `filter p xs` es la lista de elementos de la lista `xs` que verifican el predicado `p`.
30. `flip f x y` es `f y x`.
31. `floor x` es el mayor entero no mayor que `x`.
32. `foldl f e xs` pliega `xs` de izquierda a derecha usando el operador f y el valor inicial `e`.
33. `foldr f e xs` pliega `xs` de derecha a izquierda usando el operador f y el valor inicial `e`.
34. `fromIntegral x` transforma el número entero `x` al tipo numérico correspondiente.
35. `fst p` es el primer elemento del par `p`.
36. `gcd x y` es el máximo común divisor de de `x` e `y`.
37. `head xs` es el primer elemento de la lista `xs`.
38. `init xs` es la lista obtenida eliminando el último elemento de `xs`.
39. `isSpace x` se verifica si `x` es un espacio.
40. `isUpper x` se verifica si `x` está en mayúscula.
41. `isLower x` se verifica si `x` está en minúscula.
42. `isAlpha x` se verifica si `x` es un carácter alfabético.
43. `isDigit x` se verifica si `x` es un dígito.
44. `isAlphaNum x` se verifica si `x` es un carácter alfanumérico.
45. `iterate f x` es la lista `[x, f(x), f(f(x)), ...]`.
46. `last xs` es el último elemento de la lista `xs`.
47. `length xs` es el número de elementos de la lista `xs`.
48. `map f xs` es la lista obtenida aplicado `f` a cada elemento de `xs`.
49. `max x y` es el máximo de `x` e `y`.
50. `maximum xs` es el máximo elemento de la lista `xs`.
51. `min x y` es el mínimo de `x` e `y`.
52. `minimum xs` es el mínimo elemento de la lista `xs`.
53. `mod x y` es el resto de `x` entre `y`.
54. `not x` es la negación lógica del booleano `x`.

55. `noElem x ys` se verifica si `x` no pertenece a `ys`.
56. `null xs` se verifica si `xs` es la lista vacía.
57. `odd x` se verifica si `x` es impar.
58. `or xs` es la disyunción de la lista de booleanos `xs`.
59. `ord c` es el código ASCII del carácter `c`.
60. `product xs` es el producto de la lista de números `xs`.
61. `rem x y` es el resto de `x` entre `y`.
62. `repeat x` es la lista infinita `[x, x, x, ...]`.
63. `replicate n x` es la lista formada por `n` veces el elemento `x`.
64. `reverse xs` es la inversa de la lista `xs`.
65. `round x` es el redondeo de `x` al entero más cercano.
66. `scanr f e xs` es la lista de los resultados de plegar `xs` por la derecha con `f` y `e`.
67. `show x` es la represantación de `x` como cadena.
68. `signum x` es 1 si `x` es positivo, 0 si `x` es cero y -1 si `x` es negativo.
69. `snd p` es el segundo elemento del par `p`.
70. `splitAt n xs` es `(take n xs, drop n xs)`.
71. `sqrt x` es la raíz cuadrada de `x`.
72. `sum xs` es la suma de la lista numérica `xs`.
73. `tail xs` es la lista obtenida eliminando el primer elemento de `xs`.
74. `take n xs` es la lista de los `n` primeros elementos de `xs`.
75. `takeWhile p xs` es el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
76. `uncurry f` es la versión cartesiana de la función `f`.
77. `until p f x` aplica `f` a `x` hasta que se verifique `p`.
78. `zip xs ys` es la lista de pares formado por los correspondientes elementos de `xs` e `ys`.
79. `zipWith f xs ys` se obtiene aplicando `f` a los correspondientes elementos de `xs` e `ys`.

Apéndice B

Método de Pólya para la resolución de problemas

B.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?
- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.

- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejercutar tu plan de la solución, comprueba cada uno de los pasos
- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya "Cómo plantear y resolver problemas" (Ed. Trillas, 1978) p. 19

B.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?

- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.

- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson *How to program it*, basado en G. Polya *Cómo plantear y resolver problemas*.

Bibliografía

[1] J. A. Alonso. Temas de programación funcional. Technical report, Univ. de Sevilla, 2011.

[2] R. Bird. *Introducción a la programación funcional con Haskell*. Prentice–Hall, 1999.

[3] H. C. Cunningham. Notes on functional programming with Haskell. Technical report, University of Mississippi, 2010.

[4] H. Daumé. Yet another Haskell tutorial. Technical report, University of Utah, 2006.

[5] A. Davie. *An introduction to functional programming systems using Haskell*. Cambridge University Press, 1992.

[6] K. Doets and J. van Eijck. *The Haskell road to logic, maths and programming*. King's College Publications, 2004.

[7] J. Fokker. Programación funcional. Technical report, Universidad de Utrech, 1996.

[8] P. Hudak. *The Haskell school of expression: Learning functional programming through multimedia*. Cambridge University Press, 2000.

[9] P. Hudak. The Haskell school of music (From signals to symphonies). Technical report, Yale University, 2012.

[10] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[11] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real world Haskell*. O'Reilly, 2008.

[12] G. Pólya. *Cómo plantear y resolver problemas*. Editorial Trillas, 1965.

[13] F. Rabhi and G. Lapalme. *Algorithms: A functional programming approach*. Addison–Wesley, 1999.

[14] B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. Gallardo. *Razonando con Haskell (Un curso sobre programación funcional)*. Thompson, 2004.

[15] S. Thompson. *Haskell: The craft of functional programming*. Addison–Wesley, third edition, 2011.

Índice alfabético

Carta, 192
Color, 189
Mano, 194
Palo, 189
Valor, 190
adyacentes, 308
agarradoC, 91
agarradoR, 92
agregaParidad, 334
agrupa', 147
agrupa, 114, 123, 146
and', 60
antisimetrica, 270
anulaColumnaDesde, 257
anulaEltoColumnaDesde, 257
anuladaColumnaDesde, 256
aproxE', 45
aproxE, 44
aproxLimSeno, 46
aproximaPiC, 90
aproximaPiR, 90
arbolBalanceado, 177
arcocoseno', 383
arcocoseno, 382
arcoseno', 383
arcoseno, 381
areaDeCoronaCircular, 24
area, 36
aristaEn, 308
aristas, 309
balanceado, 176
bezout, 345
bin2intC, 332
bin2intR, 332
bin2int, 332
borraP, 135
borraR, 135
borra, 64, 395
buscaCrucigramaR, 108
buscaCrucigrama, 107
buscaIndiceDesde, 255
buscaPivoteDesde, 256
busca, 50, 187
cabezasP, 139
cabezasS, 139
cabezas, 138
calculaPi, 46
cantidadHammingMenores, 159
capicua, 79
cardinal, 281
cercanos, 172
ceros, 115
chiCuad, 329
ciclo, 34
circulo, 44
clausuraReflexiva, 271
clausuraSimetrica, 272
clausuraTransitiva, 273
clave, 115
cocienteRuffini, 239
cociente, 236
codifica, 327, 334
coeficientes, 232
coeficiente, 232
colas, 138
collatz', 149

collatz, 149
color, 190
columnaMat, 248
combinacionesN, 365
combinacionesRN, 368
combinacionesR, 367
combinaciones, 364
comb, 366
completo, 312
composicion, 268
compruebaParidad, 335
concat', 61
concatP, 119
concatR, 119
conjetura, 48
conjugado, 33
consecutivosConSuma, 346, 354
contenido, 405
contieneR, 110
contiene, 110
contiguos, 314
copia, 203
creaGrafo, 305
creaOcteto, 333
creaPolDensa, 230
creaPolDispersa, 230
criba, 352
cuadradoMagico, 396
cuadradosC, 82
cuadradosMagicos3, 398
cuadradosMagicos, 397
cuadradosR, 82
cuadrante, 31
cuatroIguales, 29
dec2entP, 130
dec2entR, 130
densaAdispersa, 231
densa, 51, 231
derivadaBurda, 376
derivadaFinaDelSeno, 376
derivadaFina, 376
derivadaSuper, 376
derivada, 375
descifra, 330
descodifica, 336
desplaza, 327
determinante, 261
diagonalPral, 251, 393
diagonalSec, 252, 394
diferenciaP, 136
diferenciaR, 135
diferenciaSimetrica, 285
diferencia, 285
digitosC, 72
digitosDeFactorizacion, 96
digitosR, 72
dimension, 246
dirigido, 307
disjuntos, 284
dispersa, 231
distanciaC, 99
distanciaR, 100
distancia, 32, 346
divideMedia, 119
divide, 112, 286
divisiblePol, 237
divisionSegura, 29
divisoresEn, 158
divisores, 42, 162, 238, 342
dobleFactorial, 57
dropWhile', 118
ecoC, 144
ecoR, 144
ejGrafoD, 307
ejGrafoND, 306
elem', 60
elementosNoNulosColDesde, 258
elementos, 395
eligeCarta, 195
eliminaP, 352

eliminaR, 352
elimina, 351
empiezaConDos, 403
enRangoC, 88
enRangoR, 88
enteros', 156
enteros, 155
entreL, 86
entreR, 86
equivalentes, 81
errorAproxE, 45
errorE', 45
errorLimSeno, 46
errorPi, 47
esCuadrada, 250
esDigito, 74
esEquivalencia, 269
esFactorial, 386
esFuncion, 344
esMuyCompuesto, 152
esPermutacion, 65, 395
esPrimo, 353
esProductoDeDosPrimos, 152
esRaizRuffini, 241
esSemiPerfecto, 342
esSimetrica, 251
esSolucion, 349
esSuma, 347
esTautologia, 187
eslabones, 167
especial, 81, 98
espejo, 181
euler12, 161
euler16, 79
euler1, 43
euler5, 58
euler9, 49
everyC, 287
existeColNoNulaDesde, 258
expansionC, 95
expansionR, 95
extremos, 27
e, 45
factoresPrimos, 93
factores, 42, 92
factoriales, 386
factorial, 385
factorizacion, 94, 242
fact, 363
filaMat, 248
filterP, 133
filterR, 132
filtraAplica, 125
filtra, 285
finales, 27
formaReducida, 36
frase, 114
frecuencias, 329
fusc, 409
ganaCarta, 193
ganaMano, 194
gaussAux, 259
gauss, 260
golomb, 168
gradoNeg, 317
gradoPos, 316
grado, 317
grafoCiclo, 313
grafoReducido, 342
grafoSumaImpares, 200
grafo, 267
hamming', 159
hamming, 157
horner, 237
huecoHamming, 160
igualConjunto, 360
igual_sucesion_HB_CalkinWilf, 408
igualdadRacional, 37
imparesCuadradosC, 84
imparesCuadradosR, 84

imparesC, 83
imparesR, 83
incidentes, 314
indice, 406
int2bin, 332
int2mayuscula, 326
int2minuscula, 326
integralDef, 235
integral, 234
intercala, 34, 361
intercambiaColumnas, 253
intercambiaFilas, 253
intercambia, 32
interior, 26
interpretaciones', 189
interpretacionesVar', 188
interpretacionesVar, 187
interpretaciones, 187
interseccionG, 284
interseccion, 283
invFactorial, 387
inversaP', 129
inversaP, 128
inversaR', 128
inversaR, 128
inversas, 114
inversa, 383
inverso', 78
inverso, 78
irreflexiva, 269
itera, 146
last', 60
lazos, 315
letras, 328
linea, 41
listaMatriz, 245
listaNumeroC, 75
listaNumeroR, 75
listaVector, 245
longitudes, 113
mapC, 286
mapP, 132
mapR, 132
matrizCuadrada, 394
matrizLista, 247
matriz, 396
maxTres, 24
maximaSuma, 262
maximumP, 127
maximumR, 126
mayorExponenteC, 94
mayorExponenteR, 94
mayorRectanglo, 31
mayor, 191
mayuscula2int, 326
mayusculaInicialR, 105
mayusculaInicial, 105
mcd, 58
media3, 23
mediano', 28
mediano, 28
menorCollatzMayor, 149
menorCollatzSupera, 150
menorDivisible, 58
menorIndiceColNoNulaDesde, 259
menorQueEsSuma, 348
mezcla, 62
minimumP, 127
minuscula2int, 326
mitadParesC, 87
mitadParesR, 87
mitades, 63, 177
modulo, 30
multEscalar, 235
multFilaPor, 254
multiplosRestringidos, 145
musicos', 53
musicos, 52
muyCompuesto, 153
nAristas, 315

nDivisores, 162
nHojas, 176, 178
nLazos, 315
nNodos, 178
nRepresentacionesHB, 404
nVertices, 313
nivelesCalkinWilf, 407
noDirigido, 313
nodos, 307
nombres, 52
numColumnas, 246
numFilas, 245
numPasosHanoi, 59
numVars, 197
numeroAbundante, 42
numeroBloquesC, 70
numeroBloquesR, 70
numeroCombinacionesR', 368
numeroCombinacionesR, 368
numeroCombinaciones, 365
numeroDeDigitos, 74
numeroDePares, 48
numeroDeRaices, 35
numeroMayor, 34
numeroPermutacionesN, 363
numeroVariaciones', 370
numeroVariacionesR', 372
numeroVariacionesR, 371
numeroVariaciones, 370
numeroVueltas, 167
numerosAbundantesMenores, 42
ocurrencias, 329
ocurre, 174
ordMezcla, 63
ordenada, 64
pRuffini, 239
palabras, 112
palindromo, 26
palo, 192
pares', 51
paresOrdenados, 164
pares, 387
paridad, 334
particion, 286
pascal, 373
pegaNumerosNR, 76
pegaNumerosR, 76
perfectos, 42
permutacionesN, 362
permutaciones, 362
perteneceRango, 163
peso, 308
pitagoricas, 47
porcentaje, 328
posiciones', 50
posicionesFactoriales, 387
posicionesR, 109
posiciones, 50, 108, 244
posicion, 156
postorden, 180
potenciaFunc, 165
potenciaM, 234
potenciasDeDos, 402
potenciasMenores, 145
potencia, 56, 233, 288
prefijosConSuma, 354
preordenIt, 180
preorden, 179
primerAbundanteImpar, 43
primerDigitoNR, 77
primerDigitoR, 77
primerSemiPerfecto, 343
primitivo, 80
primoPermutable, 155
primoTruncable, 154
primosConsecutivosConSuma, 354
primosEquivalentes, 163
primos, 150, 353
primo, 93, 151
prodEscalar, 249

prodMatrices, 249
productoComplejos, 33
productoC, 287
productoEscalar, 49
productoPos, 137
productoPred, 137
productoRacional, 37
producto, 80, 137, 173
profundidad, 179
propiedad1, 355
propiedad2, 357
propiedad3, 357
puntoCeroI, 380
puntoCero, 379
puntoMedio, 32
raicesRuffini, 241
raices_1, 35
raices_2, 35
raizCuadrada', 383
raizCuadrada, 380
raizCubica', 383
raizCubica, 381
raizI, 378
raiz, 377
ramaIzquierda, 183
rango, 26
reagrupa, 113
refinada, 62
reflexiva, 267
regularidad, 320
regular, 319
relacionados, 122
repeatArbol, 182
repiteC, 142
repiteFinita', 143
repiteFinitaC, 143
repiteFinita, 143
repite, 142
replicate', 56
replicateArbol, 183
replica, 40
representacionesHB, 403
restoRuffini, 240
resto, 236
reversiblesMenores, 340
rota1, 25
rota, 25, 330
segmentos, 122
segmento, 27
selecciona, 61
seleccion, 53
semiPerfecto, 343
separa9, 335
separa, 246
siguienteHamming, 160
siguiente, 149, 407
simetrica, 267
simetricoH, 32
solucionFactoriales, 388
solucion, 349
someC, 287
subSucGolomb, 168
subconjuntoPropio, 280
subconjuntos, 342, 361
subconjunto, 268, 279, 359
submatriz, 252
sucGolomb, 168
sucesionCalkinWilf, 408
sucesionHB, 404
sucesores, 407
suma', 41
sumaCifrasListaP, 356
sumaCifrasListaR, 356
sumaCifrasLista, 356
sumaCifras, 355
sumaComplejos, 33
sumaConsecutivos, 50
sumaCuadradosC, 69
sumaCuadradosImparesC, 71, 85
sumaCuadradosImparesR, 71, 85

sumaCuadradosR, 69
sumaDeCuadrados, 40
sumaDeDosPrimos, 151
sumaDeDos, 165
sumaDigitosC, 104
sumaDigitosNR, 73
sumaDigitosR, 73, 104
sumaEspecialesR, 99
sumaEspeciales, 99
sumaFilaFila, 254
sumaFilaPor, 255
sumaImpares', 200
sumaImparesIguales, 200
sumaImpares, 199
sumaMatrices, 247
sumaMonedas, 23
sumaPositivosC, 89
sumaPositivosR, 89
sumaPotenciasDeDosMasUno', 202
sumaPotenciasDeDosMasUno, 201
sumaPrimoMenores, 161
sumaPrimosTruncables, 154
sumaP, 131
sumaRacional, 36
sumaR, 131
sumasDe2Cuadrados, 339
sumasDeColumnas, 393
sumasDeFilas, 393
sumas, 101
suma, 41, 173
sumllAP, 134
sumllA, 134
sumllP, 133
sumllR, 133
superpar, 123
sustitucion, 198
sustituyeImpar, 91
tabla, 328
take', 61
takeArbol, 182
takeWhile', 118
terminoIndep, 238
termino, 404
ternasPitagoricas, 49
tituloR, 107
titulo, 106
todos', 204
todosIguales, 394
todosPares, 43
todos, 203
total, 271
transitiva, 268
traspuesta, 101, 250, 392
tresDiferentes, 28
tresIguales, 28
triangulares, 162
triangular, 29
triangulo, 41
ullman, 338
ultimaCifra, 24
ultimoDigito, 77
une, 113
unionG, 282
union, 281
unitario, 281
universo, 266
until', 378
valor, 187, 192, 197
variables, 187
variacionesN, 369
variacionesRN, 371
variacionesR, 371
variaciones, 369
vectorLista, 247
vivas, 53
volumenEsfera, 23
xor1, 24, 30
xor2, 25, 30
xor3, 25, 30
xor4, 25, 30