Relación de Ejercicios 2

Para realizar estos ejercicios, crea un nuevo fichero (con extensión hs) con identificador formado por tus iniciales de apellidos y nombre, seguido de Rel2 y seguido del ejercicio o ejercicios que contiene; añade al principio de tu fichero la siguiente cabecera, reemplazando los datos necesarios:

```
-------------------------------------------------------------------------------
-- Estructuras de Datos. 2º Curso. ETSI Informática. UMA
--
-- (completa y sustituye los siguientes datos)
-- Titulación: Grado en Ingeniería ………………………………… [Informática | del Software | de Computadores].
-- Alumno: APELLIDOS, NOMBRE
-- Fecha de entrega:  DIA | MES | AÑO
--
-- Relación de Ejercicios 2. Ejercicios resueltos: ..........
--
-------------------------------------------------------------------------------

import Test.QuickCheck
```

1. Consideremos la declaración

```
data Direction = North | South | East | West
                             deriving (Eq,Ord,Enum,Show)
```

a) Utilizando la función `fromEnum` que devuelve un nº natural distinto para valor del tipo, define directamente una versión alternativa de la relación de orden `(<)`, y llámala `(<<)`

```
(<<) :: Direction -> Direction -> Bool
```

Prueba que las dos relaciones de orden son iguales a través de QuickCheck, y la propiedad:

```
p_menor x  y =  (x < y) ==  (x << y)
instance Arbitrary Direction where
       arbitrary = do
              n <- choose (0,3)
              return $ toEnum n
```

b) Elimina la clase `Ord` en la cláusula `deriving` de la declaración `data Direction`, y trata de definir una instancia de Ord “manualmente” definiendo el operador (<=) a través de la relación anterior (<<).

2. Define una función `máximoYresto :: Ord a => [a] -> (a,[a])` que devuelva en forma de par el máximo de una lista y los restantes elementos. Considera dos casos:

a) El orden en que aparecen los restantes puede ser arbitrario:

```
máximoYresto [1,2,30,4,5,6,7] → (30,[1,2,7,4,5,6])
```

b) Los restantes deben aparecer en el orden original

```
máximoYresto' [1,2,30,4,5,6,7] → (30,[1,2,4,5,6,7])
```

3. Define una función `reparte :: [a] -> ([a],[a])` para repartir los elementos de una lista en dos sublistas, asignando los elementos en forma alternada a cada una de las listas y en el mismo orden que el original

```
reparte [1,2,3,4,5,6,7] → ([1,3,5,7],[2,4,6])
```

4. Define una función `distintos :: Eq a => [a] -> Bool`, que compruebe si todos los elementos de la lista argumento son distintos. Por ejemplo:

```
distintos [1,7,3] → True                    distintos [1,7,3,1] → False
```

5. La función predefinida `replicate :: Int -> a -> [a]` toma un número natural *n* y un valor *x* y devuelve una lista con el valor *x* repetido *n* veces.

a) Dado que no es posible volver a definir una función predefinida, define usando *listas por comprensión* una función `replicate'` que se comporte como la función predefinida:

```
replicate' 3 0 → [0,0,0]                replicate' 4 'a' → "aaaa"
```

b) Lee y entiende la siguiente propiedad referente a la función `replicate'`:

```
p_replicate' n x =  n >= 0 && n <= 1000 ==>
                              length (filter (==x) xs) == n
                              && length (filter (/=x) xs) == 0
                                where xs = replicate' n x
```

c) Comprueba esta propiedad usando *QuickCheck* (recuerda importar `Test.QuickCheck` al principio de tu programa).

6. Usando una lista por comprensión y una función `divideA` que compruebe si un entero divide a otro, define una función `divisores` que devuelva la lista de divisores naturales de un número natural. Por ejemplo:

```
divisores 10 → [1,2,5,10]
```

Haz las modificaciones necesarias para obtener una nueva función `divisores'` que devuelva los divisores (positivos y negativos) de un número entero:

```
divisores' (-10) → [-10,-5,-2,-1,1,2,5,10]
```

7. El máximo común divisor de dos números *x* e *y*, denotado con *mcd(x,y)*, es el máximo del conjunto formado por los divisores comunes de *x* e *y*. Tal máximo existirá si los números *x* e *y* no son simultáneamente nulos (de lo contrario, todo número natural es divisor común, y no existe el máximo del conjunto de divisores comunes). Según esta definición, para el cálculo del *mcd(x,y)* basta considerar divisores positivos con *x* e *y* naturales.

a) Define, usando una lista por comprensión y la función `divisores` del ejercicio anterior, una función `mcd` que calcule el máximo común divisor de dos números. Por ejemplo:

```
mcd 30 75 → 15          mcd (-30) 75 → 15
```

Para ello, tomando de entre los divisores de *x*, los que también son divisores de *y* tendrás los divisores comunes de *x* e *y*, por lo que basta que selecciones el elemento máximo de esta lista a través, por ejemplo, de la función predefinida que devuelve el mayor elemento de una lista:

```
maximum :: (Ord a) => [a] -> a
```

b) Define y comprueba, usando *QuickCheck*, la siguiente propiedad:

para *x,y,z* enteros, con $x \neq 0$, $y \neq 0$, $z \neq 0$, se verifica $mcd(z*x,\ z*y) = |z|\ mcd(x,y)$

c) A partir de la siguiente propiedad que relaciona el *mcd* con el *mcm* (mínimo común múltiplo) de dos números

$$mcd\ x\ y \cdot mcm\ x\ y = x \cdot y$$

escribe una función que calcule el mcm de dos números. Por ejemplo:

```
mcm 9 15 → 45                mcm 30 75 → 150
```

Nota: el algoritmo de Euclides para el cálculo del *mcd* es más eficiente que el que has desarrollado en este ejercicio.

8. Un número natural *p* es primo si tiene exactamente dos divisores positivos distintos: *1* y *p*; por tanto, *1* no es un número primo.

a) Define una función `esPrimo` para comprobar si un número es primo. Por ejemplo:

```
esPrimo 7 → True                    esPrimo 10 → False
```

b) Usando una lista por comprensión, define una función `primosHasta` que devuelva una lista con todos los números primos menores o iguales a un valor dado. Por ejemplo:

`primosHasta 50 → [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]`

c) Da otra definición (con nombre `primosHasta'`) que use la función predefinida `filter` en vez de la lista por comprensión.

d) Comprueba que las dos funciones que has definido se comportan del mismo modo, comprobando con *QuickCheck* la siguiente propiedad:

`p1_primos x = primosHasta x == primosHasta' x`

Nota: existen métodos más eficientes para calcular listas de primos, como la *Criba de Eratóstenes*.

9. La conjetura de Goldbach fue formulada por Christian Goldbach en 1742 en una célebre carta a Euler. Es uno de los problemas abiertos (no resuelto) más antiguo de las Matemáticas y dice:

Todo número par mayor que 2 puede escribirse como la suma de dos primos.

Los dos primos no tienen que ser distintos. La conjetura ha sido comprobada con programas de ordenador para todos los pares menores a 10^{18}, pero no ha podido ser demostrada. El objetivo de este ejercicio es comprobarla con Haskell.

a) Usando una lista por comprensión y las funciones del ejercicio anterior, define una función `pares` que tome como parámetro un entero n y devuelva una lista de tuplas con todos los pares de primos que suman n. Por ejemplo:

`pares 3 → [ ]` `pares 10 → [(3,7),(5,5)]`

`pares 40 → [(3,37),(11,29),(17,23)]`

(observa que los pares no aparecen duplicados: por ejemplo, el par (29,11) no aparece en la lista anterior).

b) Define una función `goldbach` que tome como parámetro un entero n y devuelva `True` si n es un entero par mayor que 2 y además existe al menos una pareja de primos que sume n. Por ejemplo:

`goldbach 10 → True` `goldbach 3 → False`

Para resolver este apartado puedes utilizar la función predefinida `null :: [a] -> Bool`, que toma una lista y devuelve `True` si es vacía:

`null "ab" → False` `null [] → True` `null [(1,2),(3,4)] → False`

c) Para comprobar la conjetura, define una función `goldbachHasta` que tome como parámetro un entero n y que devuelva `True` si para todos los números pares mayores que 2 y menores o iguales a n se cumple la conjetura. Por ejemplo: `goldbachHasta 100 → True.`

Ayuda: usa listas por comprensión y la función predefinida `and :: [Bool] -> Bool` que toma una lista de booleanos y devuelve `True` si todos son ciertos.

d) Existe una forma más débil de la conjetura, llamada *Conjetura débil de Goldbach* (CDG): todo impar mayor que 5 es suma de tres números primos. Si *n* es impar *>5*, entonces *n-3* es par *>2*, y según la CG, es suma de dos primos, de donde por ser *3* un primo, *n* será suma de tres primos; de aquí que CG asegure CDG; sin embargo, CDG tampoco ha sido demostrada hasta la fecha. Define una nueva función `goldbachDébilHasta` para comprobar esta forma de la conjetura.

10. Los factores propios de un número *x* son los divisores naturales de *x* estrictamente menores a *x*. Un número natural es *perfecto* si coincide con la suma de sus factores propios.

a) Escribe una función `esPerfecto` para comprobar si un número es perfecto. Por ejemplo:

`esPerfecto 6 → True` `esPerfecto 10 → False`

b) Escribe otra función que devuelva una lista con los números perfectos menores o iguales a un valor dado. Por ejemplo:

perfectosMenoresQue 500 → `[6,28,496]`

11. La función predefinida `take` toma un número natural `n` y una lista `xs`, y devuelve una lista con los `n` primeros elementos de `xs`.

a) Dado que no es posible volver a definir una función predefinida, completa la siguiente definición

```
take' :: Int -> [a] -> [a]
take' n xs = [ ???  | (p,x) <- zip [0.. ??? ] xs ]
```

para que la función `take'` se comporte como la función predefinida `take`:

```
take' 3 [0,1,2,3,4,5] → [0,1,2]
take' 0 [0,1,2,3,4,5] → []
take' 5 [0,1,2]       → [0,1,2]
```

b) La función predefinida `drop` toma un número natural `n` y una lista `xs`, y devuelve la lista que se obtiene al eliminar los primeros n elementos de `xs`. Completa la siguiente definición

```
drop' :: Int -> [a] -> [a]
drop' n xs = [ ??? | (p,x) <- zip [ ??? ] xs,  ??? ]
```

para que la función `drop'` se comporte como la función predefinida `drop`:

```
drop' 3 [0,1,2,3,4,5] → [3,4,5]
drop' 0 [0,1,2,3,4,5] → [0,1,2,3,4,5]
drop' 5 [0,1,2]       → []
```

c) Escribe y comprueba con *QuickCheck* la siguiente propiedad: para cualquier n≥0 y cualquier lista `xs`, si concatenamos la lista `take' n xs` con la lista `drop' n xs`, obtenemos la lista original `xs`.

12. La función predefinida

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

toma una lista de listas y devuelve la lista que se obtiene al concatenar todos sus elementos:

```
concat [ [1,2,3], [5,6], [8,0,1,2] ] → [1,2,3,5,6,8,0,1,2]
```

a) Dado que no es posible volver a definir una función predefinida, define usando `foldr` una función `concat'` que se comporte como la predefinida.
Ayuda: observa que el resultado se puede calcular con la siguiente expresión:

```
[1,2,3] ++ ( [5,6] ++ ( [8,0,1,2] ++ [] ) )
```

donde el operador predefinido (++) concatena dos listas.

b) Da ahora una definición alternativa `concat''` usando listas por comprensión. Usa para ello dos generadores; el primero extraerá cada lista de la lista de listas argumento; el segundo extraerá los elementos de las listas extraídas por el primer generador.

13. Las funciones predefinidas `and`, `tail` y `zip` han sido explicadas en el tema 2. Analiza y entiende qué hace la siguiente función:

```
desconocida :: (Ord a) => [a] -> Bool
desconocida xs = and [ x<=y | (x,y) <- zip xs (tail xs) ]
```

14. La función predefinida

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

devuelve el prefijo más largo con los elementos de una lista (2º argumento) que cumplen una condición (1er argumento). Por ejemplo:

```
takeWhile even [2,4,6,8,11,13,16,20] → [2,4,6,8]
```

ya que el 11 es el primer elemento que no es par. Otro ejemplo de uso es:

```
takeWhile (<5) [2,4,6,1] → [2,4]
```

ya que 6 es el primer elemento de la lista mayor o igual a 5. Para los mismos argumentos, la función `dropWhile` suprime el prefijo que `takeWhile` devuelve. Por ejemplo:

```
dropWhile even [2,4,6,8,11,13,16,20] → [11,13,16,20]
```

```
dropWhile (<5) [2,4,6,1] → [6,1]
```

a) Usando estas funciones, define una función `inserta` que tome un elemento *x* y una lista *xs* que ya está ordenada ascendentemente (asume que esta precondición se cumple), y que devuelva la lista ordenada que se obtiene al insertar *x* en su posición adecuada dentro de *xs*. Por ejemplo:

```
inserta 5 [1,2,4,7,8,11] → [1,2,4,5,7,8,11]
inserta 2 [1,2,4,7,8,11] → [1,2,2,4,7,8,11]
inserta 0 [1,2,4,7,8,11] → [0,1,2,4,7,8,11]
inserta 20 [1,2,4,7,8,11] → [1,2,4,7,8,11,20]
```

b) Sin usar ninguna función auxiliar, define directamente y en forma recursiva la función `inserta`.

c) Lee, entiende y comprueba con *QuickCheck* la siguiente propiedad referente a la función `inserta`:

```
p1_inserta x xs = desconocida xs ==> desconocida (inserta x xs)
```

d) Podemos utilizar la función `inserta` que hemos definido para ordenar ascendentemente una lista desordenada. Por ejemplo, si quisiéramos ordenar la lista `[9,3,7]`, podríamos hacerlo evaluando la expresión:

```
9 `inserta` (3 `inserta` (7 `inserta` []))
```

Razona por qué funciona este algoritmo para ordenar una lista.

e) Usando la función `foldr` y la función `inserta`, define una función `ordena` que tome una lista de valores y la devuelva ordenada. Por ejemplo:

```
ordena [9,3,7] → [3,7,9]      ordena "abracadabra"→ "aaaaabbcdrr"
```

f) Define y comprueba con *QuickCheck* la siguiente propiedad: para cualquier lista `xs`, `ordena xs` es una lista ordenada.

15. La función de orden superior predefinida

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

toma como argumentos una función `f` y un valor `x`, y devuelve la lista infinita siguiente:

```
iterate f x → [x, f x, f (f x), f (f (f x))), f (f (f (f x)))), ... ]
```

Es decir, el primer término es `x` y los demás se obtienen aplicando la función `f` al término que le precede. Usando `iterate`, es posible definir secuencias aritméticas si la función `f` suma cierta cantidad fija. Por ejemplo:

```
iterate (+1) 0 → [ 0, 1, 2, 3, 4 ...
iterate (+2) 1 → [ 1, 3, 5, 7, 9 ...
```

a) Una secuencia geométrica está constituida por una secuencia de elementos en la que cada uno de ellos se obtiene multiplicando el anterior por una constante denominada razón o factor de la secuencia. Define usando `iterate` una función `geométrica` que devuelva una lista con una secuencia geométrica, dados el valor inicial y la razón. Por ejemplo:

```
geométrica 1 2  → [ 1, 2, 4, 8, 16 ...
geométrica 10 3 → [ 10, 30, 90, 270 ...
```

b) ¿Qué comprueba la siguiente propiedad?

```
p1_geométrica x r = x>0 && r>0 ==>
                     and [ div z y == r | (y,z) <- zip xs (tail xs) ]
                       where xs = take 100 (geométrica x r)
```

c) Usando `iterate`, define una función `múltiplosDe` que devuelva una lista con los múltiplos de su argumento. Por ejemplo:

```
múltiplosDe 2 → [ 0, 2, 4, 6, 8, 10 ...
múltiplosDe 3  → [ 0, 3, 6, 9, 12, 15 ...
```

d) Usando `iterate`, define una función `potenciasDe` que devuelva una lista con las potencias de su argumento. Por ejemplo:

```
potenciasDe 2 → [ 1, 2, 4, 8, 16, 32 ...
```

```
potenciasDe 3 → [ 1, 3, 9, 27, 81 ...
```

16. Aunque la función predefinida `lcm` (*least common multiple*) ya lo hace, el objetivo de este ejercicio es escribir una función `mcm` para calcular el mínimo común múltiplo de dos naturales.

a) Define una función `múltiplosDe`, que tome como parámetro un número mayor que cero y devuelva una lista infinita con sus múltiplos positivos. Por ejemplo:

```
múltiplosDe 3 → [ 0, 3, 6, 9, 12, 15 ...
```

Ayuda: usa la función predefinida `iterate`, descrita en un ejercicio anterior.

b) Define la función sobrecargada para tipos con orden

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

que tome dos listas ordenadas ascendentemente (asume esta precondición) y devuelva el menor elemento común a ambas. Por ejemplo:

```
primeroComún [1,2,5,7,9] [3,3,4,5,7] → 5
```

Para ello, compara las cabezas de las dos listas y avanza sobre aquella que tenga el menor elemento, hasta que las cabezas sean iguales.

c) El mínimo común múltiplo de dos naturales *x* e *y* es el menor natural positivo múltiplo de ambos. Utilizando las funciones definidas en los apartados previos, escribe una función `mcm` que calcule el mínimo común múltiplo de dos naturales usando esta definición.

d) Comprueba usando QuickCheck tu definición mediante esta propiedad:

```
p_mcm x y = x>=0 && y>=0 ==> mcm x y == lcm x y
```

17. Define la función sobrecargada para tipos con orden

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

que tome **tres** listas ordenadas ascendentemente y devuelva el menor elemento común a ambas. Por ejemplo:

```
primeroComúnDeTres [1,2,5,7,9] [3,3,4,5,7] [1,3,7,11] → 7
```

Ayuda: completa un esquema como el siguiente :

```
primeroComúnDeTres (x:xs) (y:ys) (z:zs)
 | x > y  = primeroComúnDeTres (x:xs) ys  (z:zs)
 . . .
```

Prueba la corrección de la solución.

18. La secuencia de factores primos de un natural *n* es la secuencia ordenada (lista) de números primos tal que su producto es exactamente el número *n*. Por ejemplo, la secuencia de factores primos de 220 es la lista `[2,2,5,11]`. Sea ahora la siguiente función que devuelve una lista con los factores primos de un número natural:

```
factPrimos :: Integer -> [Integer]
factPrimos x = fp x 2
  where
     fp x d
      | x' < d      = [x]
      | r == 0      = d : fp x' d
      | otherwise   = fp x (d+1)
      where (x',r) = divMod x d  -- cociente y resto
```

Así:

```
factPrimos 220 → [2,2,5,11]
```

a) Calcula manualmente y paso a paso la forma normal de `factPrimos 220`.

b) Razona por qué todos los factores que se obtienen con la función `factPrimos` son números primos. Razona también por qué cuando `x'< d` la factorización ha concluido.

c) Una fuente de ineficiencia para el algoritmo anterior es que además de considerar el factor primo 2, se consideran los demás números pares, y éstos nunca serán factores primos. Escribe una función `factPrimos'` que corrija este problema.

d) Define una propiedad `p_factores` para comprobar con QuickCheck que el producto de los factores primos de un número natural coincide con el propio número. Puedes usar la función predefinida `product` para multiplicar los elementos de una lista.

19. Un método para calcular el mínimo común múltiplo (mcm) de dos números naturales consiste en obtener sus descomposiciones en factores primos $p_1^{k_1} p_2^{k_2} \dots p_s^{k_s}$ y multiplicar los factores de la forma p^k comunes y no comunes, ambos con mayor exponente. Por ejemplo: mcm(72,50) = mcm($2^3 3^2$, $2^1 5^2$) = $2^3 3^2 5^2$. Otra forma de proceder consiste en tomar primos repetidos de las listas de factores primos:

```
factPrimos 72 → [2,2,2,3,3]          -- 2³3²
factPrimos 50 → [2,5,5]              -- 2¹5²
```

de donde el mcm de 72 y 50 es 2*2*2*3*3*5*5, o lo que es lo mismo $2^3 3^2 5^2$.

a) Escribe una función recursiva que tome como parámetros dos listas de factores primos (ordenadas ascendentemente) y devuelva la lista ordenada formada por las sublistas de factores primos comunes más larga, además de la sublista de los factores primos no comunes. Por ejemplo:

```
mezcla [2,2,2,3,3] [2,5,5] → [2,2,2,3,3,5,5]
```

Ayuda: observa los colores en los argumentos y el resultado.

b) Usando la función `mezcla`, define una función `mcm'` que calcule el mcm de dos números naturales.

c) Recuerda que `lcm` es la función predefinida para calcular el mcm. Comprueba tu función con QuickCheck mediante la siguiente propiedad:

```
p_mcm' x y = x>=0 && y>=0 ==> mcm' x y == lcm x y
```

20. Un método para calcular el máximo común divisor (mcd) de dos números naturales (no nulos simultáneamente) consiste en obtener las descomposiciones en factores primos de ambos y multiplicar los factores comunes de la forma p^k con menor exponente. Por ejemplo:

```
factPrimos 48 → [2,2,2,2,3]          factPrimos 60 → [2,2,3,5]
```

por lo que el mcd de 48 y 60 es 2*2*3.

a) Escribe una función recursiva `mezc'` que tome las listas de factores primos ordenadas ascendentemente y devuelva una lista ordenada con los factores comunes repetidos según la menor potencia. Por ejemplo:

```
mezc' [2,2,2,2,3] [2,2,3,5] → [2,2,3]
```

Ayuda: observa los colores en los argumentos y el resultado.

b) Usando la función `mezc'`, define una función `mcd''` que calcule el mcd de dos naturales no nulos simultáneamente.

c) Recuerda que `gcd` es la función predefinida para calcular el mcd. Comprueba tu función con QuickCheck mediante la siguiente propiedad:

```
p_mcd'' x y = x>0 && y>0 ==> mcd'' x y == gcd x y
```

Modifica la precondición para comprobar la corrección con naturales no nulos simultáneamente.

21. La función nub de la biblioteca Data.`List` elimina los elementos repetidos de una lista respetando la posición de la primera aparición. Por ejemplo:

```
nub [1,3,1,2,7,2,9] → [1,3,2,7,9]
```

a) Define tu propia versión de esta función (llámala `nub'` y sobrecárgala para tipos con igualdad).

b) Compruébala con QuickCheck usando la siguiente propiedad:

```
p_nub' xs = nub xs == nub' xs
```

Recuerda importar la función nub de la biblioteca Data.List.

c) Sea la siguiente propiedad que pretende comprobar la corrección de nub' (la función distintos se definió en el ejercicio 4):

```
p_sinRepes xs = distintos (nub' xs)
```

Razona por qué es incompleta, es decir, por qué aunque es una condición necesaria, esta propiedad no es suficiente para garantizar la corrección de nub'.

d) La función predefinida all comprueba si todos los elementos de una lista verifican una condición:

```
all (>10)   [100,50,20]  → True
all (=='0') "0000"       → True
all even    [1,2,3,4]    → False
```

La función predefinida elem comprueba si un valor pertenece a una lista:

```
5   `elem` [1,2,5,9] → True
'1' `elem` "0000"    → False
```

Sea la siguiente función (no predefinida):

```
todosEn :: (Eq a) => [a] -> [a] -> Bool
ys `todosEn` xs = all (`elem` xs) ys
```

que comprueba si todos los elementos de la lista ys pertenecen a la lista xs. Por ejemplo:

```
"011001"  `todosEn` "01" → True
"01A1001" `todosEn` "01" → False
```

Define usando todosEn una propiedad p_sinRepes' que garantice la corrección de la función nub'.

22. Las cadenas binarias son aquellas cadenas de caracteres que solo incluyen los caracteres '0' y '1'.

a) Define la función binarios, que tome como parámetro un número natural *n*, y devuelva todas las cadenas binarias de longitud n. Por ejemplo:

```
binarios 0 → [""]
binarios 1 → ["0","1"]
binarios 2 → ["00","01","10","11"]
binarios 3 → ["000","001","010","011","100","101","110","111"]
```

Ayuda: observa que hay 2^n cadenas binarias de longitud n, por lo que hay una (la lista vacía) de longitud cero, es decir, el caso base es:

```
binarios 0 = [ [] ]
```

Para obtener las cadenas de longitud *n* basta con tomar todas las de longitud *n*-1, añadirles un cero por delante y volver a tomarlas añadiéndoles ahora un uno. Recuerda que una cadena de caracteres es una lista de tipo [Char], por lo que puedes usar cualquier función sobre listas que necesites.

b) Sea la siguiente propiedad para comprobar la corrección de la función binarios:

```
p_binarios n = n>=0 && n<=10 ==>
                    long xss == 2^n
                    && distintos xss
                    && all (`todosEn` "01") xss
        where xss = binarios n
```

siendo

```
long :: [a] -> Integer
long xs = fromIntegral (length xs)
```

la función (no predefinida) que devuelve la longitud de una lista con tipo Integer. Observa que las pruebas se han limitado a cadenas binarias con longitud inferior a 11 para que el tiempo de las comprobaciones sea práctico (recuerda que el nº de cadenas crece exponencialmente).

Lee, entiende y comprueba con QuickCheck la propiedad p_binarios.

23. Sea xs una lista con n elementos distintos. Las variaciones con repetición de los n elementos de la lista xs tomados de m en m son todas las listas de longitud m que se pueden obtener combinando los elementos de la lista xs, pudiendo éstos estar repetidos.

a) Define una función recursiva `varRep` que tome como parámetros un natural m y una lista xs y devuelva las variaciones con repetición de xs tomadas de m en m:

```
varRep 0 "abc" → [""]
varRep 1 "abc" → ["a","b","c"]
varRep 2 "abc" → ["aa","ab","ac","ba","bb","bc","ca","cb","cc"]
```

Ayuda: este problema es como el ejercicio anterior (función `binarios`) pero con n elementos en vez de sólo el `'0'` y el `'1'`. Para seleccionar cada uno de los n elementos a la hora de añadirlos puedes usar un generador en una lista por comprensión.

b) Usando la definición anterior, o por otro método, prueba que existen n^m variaciones con repetición de n elementos tomados de m en m. Lee, entiende y comprueba con QuickCheck para listas de caracteres y enteros la siguiente propiedad:

```
p_varRep m xs = m>=0 && m<=5 && n<=5 && distintos xs ==>
                    long vss == n^m
                    && distintos vss
                    && all (`todosEn` xs) vss
        where vss = varRep m xs
              n   = long xs
```

c) Las variaciones con repetición permiten resolver problemas combinatorios como el siguiente:

Con un punto y una raya (símbolos del alfabeto Morse) ¿cuántas señales distintas de 5 caracteres pueden emitirse?

Calcula la solución a este problema usando la función `varRep`.

24. Sea xs una lista con n elementos distintos. Las variaciones ordinarias (sin repetición) de los n elementos de la lista xs tomados de m en m (con m≤n) son todas las listas de longitud m que se pueden obtener combinando los elementos de la lista xs, sin que éstos estén repetidos dentro de una misma variación.

a) Define una función recursiva `var` que tome como parámetros un natural m y una lista xs y devuelva las variaciones de xs tomadas de m en m:

```
var 0 "abc" → [""]
var 1 "abc" → ["a","b","c"]
var 2 "abc" → ["ab","ac","ba","bc","ca","cb"]
var 3 "abc" → ["abc","acb","bac","bca","cab","cba"]
```

Ayuda: este problema es como el ejercicio anterior (función `varRep`) pero solo se puede añadir un nuevo elemento a una variación de tamaño inferior si el elemento no pertenece aún a ésta. Puedes comprobar que un elemento no pertenece a una lista con la función predefinida `notElem`:

```
0 `notElem` [1..10] → True
5 `notElem` [1..10] → False
```

b) Usando la definición anterior, o por otro método, prueba que existen n! / (n-m)! variaciones de n elementos tomados de m en m. Lee, entiende y comprueba con QuickCheck para listas de caracteres y enteros la siguiente propiedad:

```
p_var m xs = n<=5 && distintos xs && m>=0 && m<=n ==>
                 long vss == fact n `div` fact (n-m)
                 && distintos vss
                 && all distintos vss
                 && all (`todosEn` xs) vss
  where
```

```
    vss = var m xs
    n = long xs
    fact :: Integer -> Integer
    fact x = product [1..x]
```

c) Las variaciones permiten resolver problemas combinatorios como el siguiente:

¿Cuáles son los números de tres cifras distintas que se pueden escribir usando los dígitos del 1 al 9?

Calcula la solución a este problema usando la función `var`.

25. Sea xs una lista con n elementos distintos. Las permutaciones de esta lista son todas las maneras posibles de ordenar sus elementos. Existen n! permutaciones.

a) Define una función `intercala`, que dado un valor y una lista con n elementos devuelva las n+1 listas que se pueden obtener al insertar el nuevo elemento en cada una de las posibles posiciones (al principio, en segunda posición, ..., al final) de la lista original. Por ejemplo:

```
intercala 0 [1,2,3] → [ [0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0] ]
```

b) Define una función `perm`, que devuelva todas las permutaciones posibles de la lista que tome como argumento. Por ejemplo:

```
perm []        → [ [] ]
perm [3]       → [ [3] ]
perm [2,3]     → [ [2,3], [3,2] ]
perm [1,2,3]   → [ [1,2,3], [2,1,3], [2,3,1], [1,3,2], [3,1,2], [3,2,1] ]
```

Ayuda: observa que para obtener las permutaciones de la lista [1,2,3], basta obtener primero las de la lista [2,3] e insertar el 1 de todas las formas posibles en cada una estas permutaciones, con lo que ya tienes un algoritmo recursivo. Entender el siguiente ejemplo también puede ayudarte:

```
[ zs | ys <- [ [2,3], [3,2] ], zs <- intercala 1 ys ] →
              [ [1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1] ]
```

c) Lee, entiende y comprueba con QuickCheck para listas de caracteres y enteros la siguiente propiedad:

```
p_perm xs =  n<8 && distintos xs ==>
                long pss == fact n
                && distintos pss
                && all (`esPermutaciónDe` xs) pss
  where
    n = long xs
    pss = perm xs
```

donde

```
esPermutaciónDe :: (Eq a) => [a] -> [a] -> Bool
xs `esPermutaciónDe` ys  = null (xs \\ ys) && null (ys \\ xs)
```

permite comprobar si una lista es una permutación de otra y el operador (`\\`), definido en la biblioteca `Data.List`, calcula la diferencia de dos listas.

d) Las permutaciones permiten resolver problemas combinatorios como el siguiente: *¿Cuáles son los números de 5 cifras distintas que se pueden escribir usando los dígitos del 1 al 5?*.

Calcula la solución a este problema usando la función `perm`.

26. Sea xs una lista con n elementos distintos. Las combinaciones de los n elementos de la lista xs tomados de m en m (con m≤n) son todos los posibles conjuntos de m elementos (no repetidos) que pueden formarse a partir de los n elementos originales. Hablamos de conjuntos porque, a diferencia de lo que ocurre con las variaciones, el orden de los elementos no tiene importancia (es decir, "abc" se considera la misma combinación que "bca"). Por ejemplo:

```
comb 0 "abcd" → [""]
comb 3 "bcd"  → ["bcd"]
comb 2 "bcd"  → ["cd","bd","bc"]
comb 3 "abcd" → ["bcd","acd","abd","abc"]
```

a) Define una función comb que tome un natural m y una lista xs con n elementos distintos y devuelva las combinaciones de los n elementos de xs tomados de m en m. Ayuda: observa cómo aparecen las combinaciones en los ejemplos anteriores para determinar una definición recursiva.

b) Hay $\binom{n}{m} = \frac{n!}{m!(n-m)!}$ combinaciones de n elementos tomados de m en m. Lee, entiende y comprueba con QuickCheck para listas de caracteres y enteros la siguiente propiedad:

```
p_comb m xs = m>=0 && n<13 && n>=m && distintos xs ==>
               long css == fact n `div` (fact m * fact (n-m))
               && and [ long cs == m | cs <- css ]
               && diferentes css
               && all (`todosEn` xs) css
               && all distintos css
  where
    n = long xs
    css = comb m xs
```

donde

```
diferentes :: (Eq a) => [[a]] -> Bool
diferentes []       = True
diferentes (xs:xss) = all (=/= xs) xss  &&  diferentes xss
  where xs =/= ys = not (xs `esPermutaciónDe` ys)
```

es una función que dada una lista de listas comprueba que ninguna es permutación de otra:

```
diferentes ["bcd","acd","abd"] → True
diferentes ["bcd","cdb"]       → False
```

c) Las combinaciones permiten resolver problemas combinatorios como el siguiente:

Siete amigos hacen cola para el cine. Al llegar a la taquilla sólo quedan 4 entradas. ¿De cuántas formas podrían repartirse estas entradas para ver la película?

Calcula la solución a este problema usando la función comb.

27. En Haskell una cadena de caracteres (String) es una lista de tipo [Char]. En este ejercicio vamos a estudiar cómo el procesamiento de cadenas de caracteres resulta muy útil en Biología.

Un problema común en la Biología moderna es entender la estructura de las moléculas de ADN y el papel de las estructuras específicas para determinar la función de una molécula. Una secuencia de ADN es comúnmente representada como una secuencia de los cuatro nucleótidos - adenina (A), citosina (C), guanina (G) y timina (T) - por lo que una molécula puede ser representada con una cadena de caracteres con los correspondientes nucleótidos, como "AAACAACTTCGTAAGTATA".

Una forma de entender la función de una cadena de ADN es ver si contiene subcadenas que coincidan con una colección de secuencias de ADN ya conocidas - es decir, secuencias cuya función y estructura ya se conoce - con la idea de que estructuras similares tienden a implicar funciones similares. Organismos simples como las bacterias pueden tener millones de nucleótidos en sus secuencias del ADN y los cromosomas humanos se cree que constan del orden de 246 millones de bases, por lo que es necesario desarrollar programas de ordenador muy eficientes para procesar estas cadenas.

a) Define una función recursiva `esPrefijoDe` que tome dos listas como argumentos, y compruebe si la primera es un prefijo de la segunda, es decir, si la segunda comienza exactamente por la primera. Por ejemplo:

```
"ATG" `esPrefijoDe` "ATGACATGCACAAGTATGCAT" → True
"ATC" `esPrefijoDe` "ATGACATGCACAAGTATGCAT" → False
```

Nota: la función `isPrefixOf` de la biblioteca `List` realiza esto mismo.

b) Define una función recursiva `búsquedas :: String -> String -> [Int]` que tome como parámetros dos listas. La primera será la cadena a buscar y la segunda la cadena donde buscar. La función debe devolver una lista de enteros con todas las posiciones donde aparezca la cadena buscada. Por convenio, consideraremos que las posiciones de las letras en una cadena empiezan a numerarse por cero. Por ejemplo:

```
búsquedas "ATG" "ATGACATGCACAAGTATGCAT" → [0,5,15]
```

ya que la cadena "ATG" aparece justo al principio (posición 0), aparece a continuación 5 posiciones después y por último aparece en la posición 15 (las apariciones de la cadena buscada se indican en subrayado).

Ayuda: usa la función `esPrefijoDe` para definir la función `búsquedas`.

c) Define una función `distancia` que dadas dos cadenas compare los caracteres en las mismas posiciones de las dos cadenas y devuelva cuantos son diferentes. Por ejemplo:

```
distancia "ATGAG" "ACGAA" → 2
```

ya que la caracteres en las posiciones 1 (T y C) y 4 (G y A) difieren en ambas cadenas. Si una cadena es más corta que otra, considera que todos los caracteres extra de la más larga son diferencias. Por ejemplo:

```
distancia "ATG" "ACGAA" → 3
```

ya que ambas cadenas difieren en los caracteres en las posiciones 1, 3 y 4.

d) Con objeto de poder buscar subcadenas parecidas a una dada, define una función `parecidas` que tome tres parámetros: un natural y dos cadenas. El natural indicará la distancia máxima permitida, la primera cadena será la cadena a buscar y la segunda la cadena donde buscar. La función debe devolver una lista con todas las posiciones donde aparece una cadena similar (con distancia menor o igual a la permitida) a la buscada. Por ejemplo:

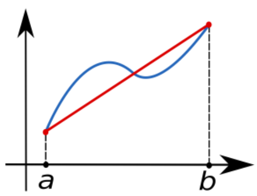
```
parecidas 1 "ATG" "ATGACATGCACAAGTATGCAT" → [0,5,11,15,19]
```

ya que en las posiciones `0,5,11,15` y `19` (indicadas en subrayado) hay cadenas cuya distancia a la cadena `"ATG"` es menor o igual a 1. Otro ejemplo más es:

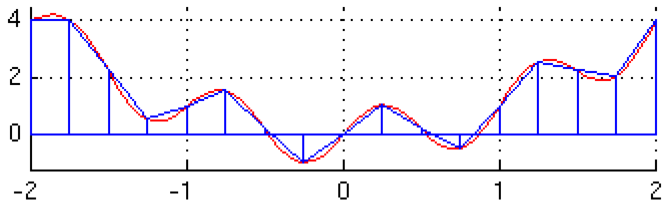
```
parecidas 2 "ATG" "ATGACATGCACAAGTATGCAT" → [0,3,5,9,11,12,13,15,19]
```

28. En este ejercicio estudiamos como calcular la integral definida de una función `f :: Double -> Double`, en un intervalo [a,b] usando la regla de los trapecios, según la cual el área bajo la curva de `f` se aproxima mediante el área de un trapecio:

$$\int_a^b f(x)dx \cong (b-a)\frac{f(a)+f(b)}{2}$$



Para calcular el área de un modo más preciso, se suele usar la versión compuesta de la regla, en la que se divide el intervalo [a,b] a integrar en varios subintervalos, cuyas áreas se aproximan cada una con la regla anterior. La aproximación de la integral definida en el intervalo [a,b] será la suma de las áreas de los distintos trapecios correspondientes a cada subintervalo. Por ejemplo, la siguiente figura muestra como se ha aproximado la integral definida de la función en rojo en el intervalo [-2,2] mediante la suma de las áreas de 16 trapecios (en azul):



a) Define una función recursiva tal que la llamada `inteDef f a b epsilon` aproxime la integral definida de la función `f` en el intervalo `[a,b]` mediante la regla compuesta. Usa para ello un algoritmo del tipo divide y vencerás. El parámetro `epsilon` indica el tamaño máximo de los subintervalos en los que se dividirá el intervalo [a,b]. Así, si la amplitud del intervalo [a,b] es menor o igual a `epsilon`, la función devolverá el área del trapecio correspondiente. En otro caso, dividirá el intervalo `[a,b]` en dos subintervalos del mismo ancho (la mitad del intervalo original), aproximará **recursivamente** la integral de ambos intervalos y devolverá su suma. Por ejemplo:

```
inteDef cos          0 (pi/2) 0.001   → 0.9999999509771441
inteDef (\x -> x)    0 5      0.001   → 12.5
inteDef (\x -> x^2) 0 5      0.001   → 41.666666977107525
```

b) El desarrollo de la regla compuesta da lugar a la siguiente expresión:

$$\int_a^b f(x)dx \cong \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2}+\sum_{k=1}^{n-1} f\left(a+k\frac{b-a}{n}\right)\right)$$

que permite aproximar la integral definida dividiendo el intervalo [a,b] en `n` subintervalos del mismo ancho. Haz uso de una lista por comprensión y la función predefinida `sum` para definir una función `inteDef' f a b n` que aproxime la integral definida de `f` con la fórmula anterior. Por ejemplo:

```
inteDef' cos          0 (pi/2) 1000 → 0.9999997943832334
inteDef' (\x -> x)    0 5      1000 → 12.499999999999998
inteDef' (\x -> x^2) 0 5      1000 → 41.666687500000016
```

29. La mediana de un conjunto de valores es aquél valor tal que el 50% de los valores del conjunto se encuentra por debajo de él y el otro 50% se encuentra por encima. Si el número de valores es impar, la mediana coincide con alguno de los valores, por ejemplo:

```
mediana [3,20,1,10,50] →10
```

ya que el 1 y el 3 son menores a 10, y además el 20 y el 50 son mayores a 10. Si el número de valores es par, la mediana se define como la media aritmética de los dos valores centrales del conjunto. Por ejemplo:

```
mediana [3,20,1,10,50,15] →12.5
```

ya que los elementos centrales son 10 y 15, y la media de éstos es 12.5.

a) Escribe una función que, dada una lista de valores de tipo `Double`, devuelva la mediana.
Ayuda: prueba a ordenar los elementos de la lista y usa el operador predefinido `(!!)`, que permite acceder al elemento que ocupa cierta posición en una lista. Por ejemplo:

```
[3,20,1,10,50,15] !! 0 → 3
[3,20,1,10,50,15] !! 1 → 20
[3,20,1,10,50,15] !! 2 → 1
```

b) Escribe una propiedad `p_mediana` para comprobar con QuickCheck que para cualquier lista `xs` de tipo Double, si ordenamos la lista y tomamos los elementos en la primera mitad de la lista ordenada, todos ellos son menores o iguales a la mediana de `xs`, y que los restantes elementos son mayores o iguales a la mediana. Para comprobar que todos los elementos de una lista cumplen una propiedad puedes usar la función predefinida `all :: (a->Bool) -> [a] -> Bool`:

```
all even [2,4,10]   → True
all (>5) [2,4,10]   → False
```

`all (<20) [2,4,10]` → `True`

30. En este ejercicio estudiamos la secuencia de Fibonacci que comienza con p,q, que se define en la forma : $x_0 = p$, $x_1 = q$, y para $n \geq 2$, $x_n = x_{n-2} + x_{n-1}$.

a) Usando parámetros acumuladores, define una función recursiva `fibs :: [Integer]` que devuelva la secuencia infinita de todos los términos de la secuencia de Fibonacci que comienza con 0, 1, 1, 2, 3,... Usa una función auxiliar `fibsAux :: Integer -> Integer -> [Integer]` con dos argumentos de modo que cada llamada `fibsAux p q` calcule la lista de los números de Fibonacci `[p, q, p+q, … ]` Por ejemplo:

```
take 4 (fibsAux 3 4) → [3,4,7,11]
take 10 fibs → [0,1,1,2,3,5,8,13,21,34]
```

b) Define una función `cocientePorMillón` que tome dos enteros y devuelva el cociente entero que se obtiene al dividir el primer número multiplicado por un millón por el segundo, con lo cual puedes obtener un cociente con seis cifras decimales exactas. Por ejemplo:

```
cocientePorMilllón 123 7 → 17571428
123 / 7                  → 17.571428571428573
```

c) Define una función `relaciones` que tome una lista de enteros y devuelva una lista con los cocientes que se obtienen al dividir (según se ha definido en el apartado anterior) cada término de la lista por su anterior (el segundo por el primero, el tercero por el segundo, etc.). Para ver cuál es la relación entre términos consecutivos de la secuencia de Fibonacci, evalúa la expresión

```
take 20 . tail $ relaciones fibs
```

¿Hacia qué valor convergen estos cocientes?

Ayuda: la relación entre los términos de la sucesión de Fibonacci permite escribir $\frac{x_{n+1}}{x_n} = \frac{x_n + x_{n-1}}{x_n} = 1 + \frac{1}{\frac{x_n}{x_{n-1}}}$, de donde el límite $\lim \frac{x_{n+1}}{x_n} = \lim \frac{x_n}{x_{n-1}} = \Phi$ satisface $\Phi = 1 + \frac{1}{\Phi}$, y de aquí, $\Phi = \frac{\sqrt{5}+1}{2} = 1.618033988749\ldots$, que es la proporción Áurea, o número de oro.

31. Usando parámetros acumuladores, define una función recursiva `facts :: [Integer]` que devuelva la lista infinita ordenada de los factoriales de los números naturales [0!, 1!, 2!, …]. Para ello partimos de la siguiente observación: a partir de los valores n! y (n+1), puedes obtener los valores (n+1)! y (n+2) realizando solamente una suma y un producto, ya que (n+1)! = (n+1) · n!. Define pues una función auxiliar tal que la llamada `factsAux n p` compute la lista `[p, p*n, p*n*(n+1), p*n*(n+1)*(n+2),…]`. Por ejemplo:

```
take 6 (factsAux 1 3) → [3,3,6,18,72,360]
take 10 facts → [1,1,2,6,24,120,720,5040,40320,362880]
```

32. Supongamos que representamos el siguiente polinomio de grado n

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$

mediante una lista de tipo `Double` de longitud n+1 con los coeficientes de todos los términos del polinomio, de menor a mayor grado: $[a_0, a_1, a_2, a_3, \cdots, a_n]$. Si alguno de los términos no existe en el polinomio (es nulo), incluiremos un cero en su correspondiente posición en la lista:

```
type Base = Double; type Valor = Base;
type Polinomio = [Base]; type Punto = Base
evalPoli :: Polinomio -> Punto -> Valor
evalPoli xs p = sum $ zipWith (\x n -> x*p^n)  xs [0..]
```

La función `evalPoli` puede ser utilizada para evaluar un polinomio (representado con la lista xs) en el punto p. Por ejemplo, `evalPoli [1,2,3] 5 → 86`, ya que `[1,2,3]` es la representación del polinomio $p(x) = 1 + 2x + 3x^2$, y $p(5)$ es 86.

a) Lee y entiende la definición de la función `evalPoli`.

b) Define, usando parámetros acumuladores, una función recursiva `evalPoli1` que tome como parámetros una lista con los coeficientes de un polinomio y un punto (de tipo `Double`), y devuelva el valor del polinomio en dicho punto. Comprueba la corrección de tu definición con la siguiente propiedad usando QuickCheck:

```
p_evalPoli1 xs p = length xs < 6 ==> evalPoli1 xs p ~= evalPoli xs p
```

donde

```
infix 4 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < epsilon
  where epsilon = 1/1000
```

c) Para un polinomio de grado n, ¿cuántos productos realiza tu función al evaluarlo? ¿cuántas sumas se realizan en la evaluación? Si usas el operador potencia, cuenta cada potencia con exponente x como x productos.

d) Observa que el polinomio $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n$ puede escribirse de forma alternativa del siguiente modo: $p(x) = a_0 + x(a_1 + a_2x^1 + a_3x^2 + \cdots + a_nx^{n-1})$, donde se ha sacado factor común *x* para los términos con grado mayor a cero. Si repetimos este proceso para el término entre paréntesis, obtenemos: $p(x) = a_0 + x(a_1 + x(a_2 + a_3x^1 + \cdots + a_nx^{n-2}))$. Repitiendo este proceso un número suficiente de veces, llegamos al siguiente desarrollo $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + xa_n)))$, que se conoce como la fórmula de Horner para evaluar un polinomio. Define una función recursiva `evalPoli2` que evalúe un polinomio siguiendo la fórmula de Horner. Por ejemplo: `evalPoli2 [1,2,3] 5` → `86`

e) Para un polinomio de grado n, ¿cuántos productos realiza `evalPoli2` al evaluarlo? ¿cuántas sumas se realizan en la evaluación? De los métodos para evaluar polinomios estudiados en este problema, ¿cuál es el más eficiente?

f) Usando `foldr`, define una función `evalPoli3` que evalúe un polinomio con la fórmula de Horner, es decir, completa la siguiente definición:

```
evalPoli3 :: Polinomio-> Punto -> Valor
evalPoli3 xs p = foldr ?????? ??? xs
```

33. El desarrollo en serie de Taylor alrededor del punto *a* de una función *f* de reales en reales infinitamente derivable se define como:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

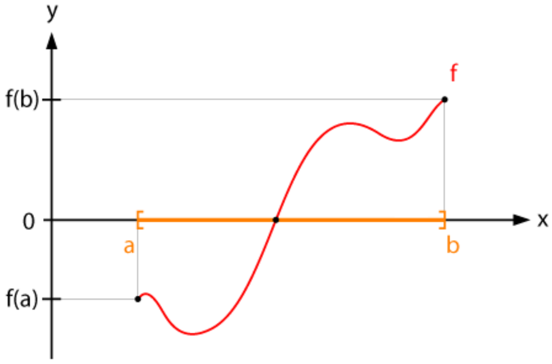
donde $f^{(n)}$ es la enésima derivada de *f*. Si *a*=0, entonces la serie se llama de MacLaurin. Estos desarrollos son muy útiles, ya que tomando pocos términos de la serie es posible calcular una buena aproximación a la función original. Por ejemplo, la serie de MacLaurin para la función seno es:

$$seno(x) = \sum_{n=0}^{\infty} \frac{(-1)^{(n)}}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

a) Define una función `macLaurinSeno` que tome como parámetro *x* y devuelva una lista infinita con los valores de los términos de la serie de MacLaurin para el seno de *x*. Define primero una variable local `nums` que represente la lista infinita [x, $-x^3$, x^5, $-x^7$, …], otra variable local `denoms` para la lista [1!, 3!, 5!, 7!, …] y empareja ambas listas término a término con la división (puedes usar `zipWith (/)` para calcular la lista final).

b) Define una función `aproxSeno` que tome como parámetros *n* y *x*, y devuelva la aproximación al seno de *x* que se obtiene sumando los *n* primeros términos de la serie de Maclaurin para el seno.

34. Sea f una función de reales en reales continua en el intervalo cerrado [a,b], que además toma valores con signos opuestos en los extremos de dicho intervalo (es decir, los signos de f(a) y f(b) son distintos). Bajo dichas suposiciones es seguro que f tiene una raíz en dicho intervalo. Este resultado se conoce como el teorema de Bolzano:



El método de bipartición usa la técnica divide y vencerás para encontrar una aproximación a dicha raíz. Los parámetros del método son f, a, b y epsilon. Para ello:

- Sea c el punto medio del intervalo determinado por a y b.
- Si la amplitud del intervalo [a,b] es menor o igual que epsilon, se devuelve el punto c como aproximación de la raíz.
- Si f(c)≅ 0, entonces se devuelve el punto c como aproximación de la raíz.
- Si hay un cambio de signo en los extremos del intervalo [a,c], se repite todo el proceso con dicho intervalo.
- En otro caso, el cambio de signo estará en el intervalo [c,b], por lo que se repite el proceso con este intervalo.

Define una función `bipartición` que tome como parámetros f, a, b y epsilon y devuelva una aproximación a una raíz de f en el intervalo [a,b] calculada con el método de bipartición. Por ejemplo:

```
Main> bipartición cos 0 pi 0.001
1.5707963267948966
Main> pi/2
1.5707963267948966
Main> cos (bipartición cos 0 pi 0.001)
6.123031769111886e-17
Main> bipartición (\x -> x^2-10) (-4) 0 0.001
-3.16259765625
Main> sqrt 10
3.1622776601683795
Main> bipartición (\x -> x^2-10) 4 5 0.001
*** Exception: No hay cambio de signo en el intervalo
```

35. Sea el siguiente tipo recursivo para representar el conjunto de los números naturales:

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

de modo que `Cero` representa el natural 0 y si n es un natural, `Suc n` representa su sucesor. Por ejemplo:

```
uno, dos, tres :: Nat
uno  = Suc Cero
dos  = Suc (Suc Cero)        -- o también dos = Suc uno
tres = Suc (Suc (Suc Cero)) -- o también tres = Suc dos
```

Copia además la siguiente instancia para que sea posible comprobar propiedades para el tipo Nat con QuickCheck:

```
instance Arbitrary Nat where
  arbitrary = do
    n <- choose (0,25) -- genera naturales pequeños
    return $ aNat n
```

donde:

```
aNat :: Integer -> Nat
aNat 0       = Cero
aNat n | n>0 = Suc . aNat $ n-1
```

a) Completa la siguiente función recursiva para que devuelva True sii su parámetro corresponde a un número natural par:

```
esPar :: Nat -> Bool
esPar Cero    = ?????
esPar (Suc n) = ?????
```

Por ejemplo:

```
Main> esPar (Suc (Suc (Suc (Suc Cero))))
True
Main> esPar tres
False
```

b) Formula y comprueba con QuickCheck la siguiente propiedad: para cualquier natural n, la paridad del sucesor del sucesor de n es la misma que la paridad de n.

c) Completa en la siguiente instancia las ecuaciones para la suma de naturales:

```
instance Num Nat where
         Cero   + y    = ???
         Suc x  + y    = ???
         abs x         = x
         signum Cero   = Cero
         signum x      = uno
         fromInteger x= aNat x
```

Por ejemplo:

```
Main> dos +  tres
Suc (Suc (Suc (Suc (Suc Cero))))
Main> 2  +  tres
Suc (Suc (Suc (Suc (Suc Cero))))
```

Ayuda: observa que $(x+1)+y = (x+y)+1$, y que Suc x denota $(x+1)$.

d) Define y comprueba con QuickCheck las siguientes propiedades de la función (+) sobre Nat:

- es conmutativa
- es asociativa
- admite a Cero como elemento neutro

e) Usando la (+), incluye en la instancia una definición recursiva para el producto (*) de dos naturales. Por ejemplo:

```
Main> 2 * 3 :: Nat
Suc (Suc (Suc (Suc (Suc (Suc Cero)))))
```

Ayuda: observa que $(x+1)\cdot y = x\cdot y + y$, y que Suc x denota $(x+1)$.

f) Define y comprueba con QuickCheck las siguientes propiedades del producto:

- (*) es conmutativa
- (*) es asociativa

- uno es el elemento neutro de `(*)`

g) Incluye en la instancia de la clase Num, definiciones recursivas para el operador diferencia `(-)` de dos naturales. Por ejemplo:

```
Main> tres -  2
Suc Cero

Main> 2 -  tres
*** Exception: Resultado negativo
```

h) Para demostrar una propiedad para cualquier valor finito de tipo `Nat`, basta:

Caso base: Demostrar que la propiedad se cumple para `Cero`

Paso inductivo: Demostrar que si la propiedad se cumple para `n`, entonces también se cumple para `Suc n`

Demuestra que la función `(+)` cumple la propiedad asociativa:

```
(x +  y)  +  z = x +  (y +  z)
```

Hazlo por inducción estructural sobre x, es decir, usando la definición de la función `sumaN` demuestra:

Caso base: `(Cero + y) +  z = Cero +  (y  +  z)`

Paso inductivo: Si `(x  +  y) +  z = x +  (y +  z)`,
entonces `((Suc x)  +  y) +  z = (Suc x)  +  ( y +  z)`

36. Sea el siguiente tipo para representar vectores de reales de dimensión 3:

```
data Vector = V Double Double Double

instance Show Vector where
       show (V x y z) = concat (zipWith (++) vs ["i","j","k"])
         where
          vs = show x : map conSigno [y,z]
          conSigno x = (if x>=0 then " + " else " - ") ++ show (abs x)
```

y los siguientes ejemplos:

```
v1, v2 :: Vector
v1 = V 1 (-3) 2
v2 = V (-2) 5 9
```

a) Lee y **entiende** la función `show` para mostrar vectores. Observa que:

```
Main> v1
1.0i - 3.0j + 2.0k

Main> v2
-2.0i + 5.0j + 9.0k
```

b) Completa la siguiente instancia de la clase `Eq` para vectores, de modo que dos vectores sean iguales si cada una de sus componentes son aproximadamente iguales:

```
instance Eq Vector where
  (V ux uy uz) == (V vx vy vz) = ?????
```

Por ejemplo:

```
Main> V (1/3) (2/3) (3/3) == V 0.333 0.666 1
True
```

c) Sea la siguiente función:

```
zipWithVector f (V ux uy uz) (V vx vy vz) = V (f ux vx) (f uy vy) (f uz vz)
```

¿Cuál es su tipo?

d) Recuerda que la suma, diferencia y el producto vectorial se definen del siguiente modo:

$$\begin{bmatrix}u_x\\u_y\\u_z\end{bmatrix}+\begin{bmatrix}v_x\\v_y\\v_z\end{bmatrix}=\begin{bmatrix}u_x+v_x\\u_y+v_y\\u_z+v_z\end{bmatrix}\qquad\begin{bmatrix}u_x\\u_y\\u_z\end{bmatrix}-\begin{bmatrix}v_x\\v_y\\v_z\end{bmatrix}=\begin{bmatrix}u_x-v_x\\u_y-v_y\\u_z-v_z\end{bmatrix}\qquad\begin{bmatrix}u_x\\u_y\\u_z\end{bmatrix}\times\begin{bmatrix}v_x\\v_y\\v_z\end{bmatrix}=\begin{bmatrix}u_yv_z-u_zv_y\\u_zv_x-u_xv_z\\u_xv_y-u_yv_x\end{bmatrix}$$

Usando la función `zipWithVector`, define las funciones

```
(+), (-) (*) :: Vector -> Vector -> Vector
```

que devuelvan la suma, diferencia, y producto vectorial de dos vectores, completando la siguiente instancia:

```
instance Num Vector where
       (+)             = ?????
       (-)             = ?????
       v1 * v2       = ?????
       abs (V x y z) = error "abs no tiene sentido para vectores"
       signum        = error "signum no tiene sentido para vectores"
       fromInteger x = V (fromInteger x) (fromInteger x) (fromInteger x)
```

.

```
Main> 0 - v1
-1.0i + 3.0j - 2.0k
Main> v1 - v2
3.0i - 8.0j - 7.0k
Main> v1 + v2
-1.0i + 2.0j + 11.0k
Main> v1 *  v2
-37.0i - 13.0j - 1.0k
```

e) Copia la siguiente instancia para que sea posible comprobar propiedades para el tipo `Vector` con QuickCheck:

```
instance Arbitrary Vector where
  arbitrary = do
    x <- arbitrary
    y <- arbitrary
    z <- arbitrary
    return (V x y z)
```

Define y comprueba con QuickCheck las siguientes propiedades para vectores:

- Distributiva de suma y producto: $(v_1+v_2)*v_3 = v_1*v_3 + v_2*v_3$
- Anticonmutativa del producto: $v_1*v_2 = -(v_2*v_1)$
- Identidad de Jacobi: $v_1*(v_2*v_3)+ v_3*(v_1*v_2)+ v_2*(v_3*v_1) = 0$

Cuando definas las funciones correspondientes a las propiedades, da además los tipos para que QuickCheck las compruebe sólo con vectores.

37. Sea el siguiente tipo para representar matrices de números reales:

```
type Fila = [Double]
data Matriz = M Int Int [Fila] deriving Eq
```

de modo que el primer entero indica el número de filas de la matriz, el segundo indica el número de columnas y por último están las componentes, como una lista de filas, donde cada fila es una lista de reales. Por ejemplo:

```
m1 :: Matriz
m1 = M 2 3 [ [ 1/3, -1/2, 1/5 ]
           , [ 1/8,  3/7, 1/4 ]
           ]
```

representa la siguiente matriz 2x3:

$$\begin{bmatrix} \frac{1}{3} & -\frac{1}{2} & \frac{1}{5} \\ \frac{1}{8} & \frac{3}{7} & \frac{1}{4} \end{bmatrix}$$

a) Copia la siguiente definición de la función `show` para matrices:

```
instance Show Matriz where
  show (M f c fs) =
    unlines . map (("| "++) . (++" |") . unwords . map rellena) $ fs
```

```
        where
          ancho     = 8  -- ancho a ocupar para cada columna
          decimales = 2  -- nº de decimales
          rellena n
            | l >= ancho = replicate ancho '*'
            | otherwise  = replicate (ancho-l) ' ' ++ xs
            where
              xs = showFFloat (Just decimales) n ""
              l  = length xs
```

y pruébala:

```
Main> m1
|     0.33    -0.50     0.20 |
|     0.13     0.43     0.25 |
```

Intenta entenderla (**difícil**).

b) Define una función `esMatriz :: Matriz -> Bool` que devuelva `True` si la lista de filas tiene tantas filas como indica el primer entero y cada una de las filas tiene tantos elementos como indica el segundo. Por ejemplo:

```
Main> esMatriz m1
True

Main> esMatriz (M 2 3 [[1,2,3],[4,5]])
False
```

Ayuda: usa las funciones predefinidas `length` y `all`.

c) Define una función `sumaF :: Fila -> Fila -> Fila` que dadas dos filas (con el mismo número de elementos) devuelva la fila que se obtiene al sumar uno a uno los componentes de cada una. Por ejemplo:

```
Main> sumaF [1,2,3] [4,5,6]
[5.0,7.0,9.0]
```

d) Usando `sumaF`, define una función `sumaM` para sumar dos matrices:

```
Main> sumaM m1 m1
|     0.67    -1.00     0.40 |
|     0.25     0.86     0.50 |

Main> sumaM m1 (M 2 2 [[1,2],[3,4]])
*** Exception: matrices no sumables
```

e) Define una función `restaM` que calcule la diferencia de dos matrices. Por ejemplo:

```
Main> restaM m1 m1
|     0.00     0.00     0.00 |
|     0.00     0.00     0.00 |
```

f) (**Difícil**) Completa la siguiente función para calcular la traspuesta de una matriz, que se obtiene cambiando filas por columnas:

```
traspuesta :: Matriz -> Matriz
traspuesta (M f c fs) = (M c f fs')
  where
    fs' = trasp fs

    trasp :: [Fila] -> [Fila]
    trasp ([]:_) = []
    trasp fs     = ?????
```

La recursión debe realizarse en la función `trasp`, que a partir de una lista de filas devuelve otra lista de filas, donde las filas devueltas se corresponden con las columnas originales. Observa que la primera fila de la traspuesta, por ejemplo, es una lista formada con los primeros elementos (cabezas) de cada una de las filas originales. Tras construir la primera fila de la traspuesta, las cabezas de cada una de las filas originales pueden ser eliminadas y el proceso puede ser repetido. Al final, quedará una lista de listas vacías, y ese es el caso base recogido en la definición de `trasp`. Por ejemplo:

```
Main> traspuesta m1
|     0.33     0.13 |
|    -0.50     0.43 |
|     0.20     0.25 |
```

g) Copia y **entiende** la siguiente función que calcula el producto de dos matrices:

```
porM :: Matriz -> Matriz -> Matriz
porM (M f1 c1 fs1) (M f2 c2 fs2)
  | c1 /= f2  = error "matrices no multiplicables"
  | otherwise = M f1 c2 [ [ prodEsc f c | c <- tfs2 ] | f <- fs1 ]
  where
       prodEsc f c = sum (zipWith (*) f c)
       M _ _ tfs2  = traspuesta (M f2 c2 fs2)
```

Recuerda que el producto de dos matrices se realiza multiplicando (con el producto escalar) cada una de las filas de la primera matriz por cada una de las columnas de la segunda. Por ejemplo:

```
Main> porM m1 (M 3 3 [[1,2,3],[4,5,6],[7,8,9]])
|    -0.27    -0.23    -0.20 |
|     3.59     4.39     5.20 |
```

h) Completa la siguiente instancia de la clase Num para el tipo Matriz:

```
instance Num Matriz where
  m1 + m2        = ?????
  m1 - m2        = ?????
  m1 * m2        = ?????
  abs (M f c fs) = ?????
  fromInteger    = error "fromInteger no tiene sentido para matrices"
  signum         = error "signum no tiene sentido para matrices"
```

donde abs debe devolver una matriz cuyos elementos sean los valores absolutos de la matriz parámetro. Evalúa varios ejemplos para comprobar que los operadores aritméticos de la clase Num (+,- y *) pueden usarse ahora con valores de tipo Matriz.

Ejercicios complementarios

38. Recordemos el operador predefinido para concatenación de listas:

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

a) Define una función `p_neutroDer` para comprobar con QuickCheck que la lista vacía es el elemento neutro por la derecha del operador `(++)`.

b) A partir de definición recursiva dada para `(++)` demuestra por inducción sobre listas que la lista vacía es el elemento neutro por la derecha: `xs ++ [] = xs`. Para ello, demuestra el siguiente caso base y paso inductivo:

(Caso base) `[] ++ [] = []`

(Paso inductivo) Si `xs ++ [] = xs` entonces `(x:xs) ++ [] = (x:xs).`

39. Consideremos la definición anterior del operador `(++).`

a) Define una función `p_asociativa` para comprobar con QuickCheck que el operador de concatenación de listas `(++)` cumple la propiedad asociativa.

b) Demuestra también por inducción sobre listas que la concatenación de listas cumple la propiedad asociativa:

`(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`

Hazlo por inducción sobre `xs`, es decir, demuestra los siguientes caso base y paso inductivo:

(Caso base) `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

(Paso inductivo) Si `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`

entonces `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

c) Aunque hemos demostrado en el apartado anterior que el resultado `(xs ++ ys) ++ zs` coincide con el de `xs ++ (ys ++ zs)`, uno de los dos modos de realizar el cálculo es más eficiente. Sea `lx`, `ly` y `lz` las longitudes de las listas `xs`, `ys` y `zs`. Calcula cuántos pasos son necesarios para calcular `(xs ++ ys) ++ zs` y cuántos para `xs ++ (ys ++ zs)`.

d) ¿Entiendes por qué el operador `(++)` está predefinido como asociativo a la derecha?

40. Demuestra por inducción sobre listas finitas que `map` es *funtor*, es decir, que verifica las siguientes propiedades:

```
map id xs = xs
map (f . g) xs = (map f . map g) xs
```

donde

```
id :: a -> a
id x = x
```

es la función predefinida identidad y el operador `(.)` es la composición de funciones.