EJERCICIOS DE MANEJO LISTAS

1) Definir la función **esVacia** que devuelve True o False si una lista es vacía o no.

```
esVacia :: [a] -> Bool
esVacia [] = True
esVacia (x:y) = False
```

2) Definir las funciones **cabeza** y **cola**, que devuelven el primer elemento y del segundo al último elemento de una lista, respectivamente.

```
cabeza :: [a] -> a
cabeza (x:y) = x

cola :: [a] -> [a]
cola (x:y) = y
```

3) Definir las siguientes funciones para manejo de listas:

i) **long**: devuelve la cantidad de elementos que hay en la lista.

```
long :: [a] -> Int
long [] = 0
long (x:y) = 1 + long y
```

ii) **suma**: devuelve la suma de todos los elementos de una lista de enteros.

```
suma :: [Int] -> Int
suma [] = 0
suma (x:y) = x + suma y
```

iii) **member**: devuelve si un número pertenece a una lista de números.

```
member :: (Int, [Int]) -> Bool
member (n, []) = False
member (n, (x:y)) = if n == x then True else member (n, y)
```

iv) **append**: devuelve la lista producto de la concatenación de las dos listas parámetros.

```
append :: ([a], [a]) -> [a]
append ([], z) = z
append ((x:y), z) = x : append (y, z)
```

v) **tomar**: dados un número n y una lista, devuelve otra lista correspondiente a los n primeros elementos de la lista dada.

```
tomar :: (Int, [a]) -> [a]
tomar (0, y) = []
tomar (x, []) = []
tomar (n, (x:y)) = x : (tomar (n-1, y))
```

vi) **term**: dados una lista y un número n, devuelve el término n-ésimo de la lista.

```
term :: ([a], Int) -> a
term (x:y, 0) = x
term (x:y, n) = term(y, n-1)
```

vii) **rev**: invierte los elementos de una lista.

```
rev :: [a] -> [a]
rev [] = []
rev (x:y) = append (rev y, x:[])
```

4) Definir una función generadora de listas **desdeHasta** que, dados dos números, produzca una lista de números consecutivos, comenzando con el primer parámetro y terminando con el segundo parámetro.

```
desdeHasta :: (Int, Int) -> [Int]
desdeHasta (a, b) = if a < b then a : (desdeHasta(a+1, b)) else (if a > b then desdeHasta(b, a) else a : [])
```

5) Utilizar la función **desdeHasta** anterior para definir la función **factorial** de una manera distinta a la dada en clase.

```
fact :: Int -> Int
fact n = multiplicacion(desdeHasta(1, n))

multiplicacion :: [Int] -> Int
multiplicacion [] = 1
multiplicacion (x:y) = x * multiplicacion y
```

6) Definir las siguientes funciones usando recursividad explícita (funciones que se llaman a sí mismas) por un lado y recursividad implícita (funciones que no se llaman a sí mismas, pero que llaman a otras que son explícita o implícitamente recursivas) por otro.

i) **ultimo**: devuelve el último elemento de una lista dada.

```
ultimo :: [a] -> a
ultimo (x:[]) = x
ultimo (x:y) = ultimo y

ultimo2 :: [a] -> a
ultimo2 x = cabeza (rev(x))

ultimo3 :: [a] -> a
ultimo3 x = term (x, (long x) - 1)
```

ii) **todosMenosUltimo**: devuelve una lista dada sin el último elemento.

```
todosMenosUltimo :: [a] -> [a]
todosMenosUltimo [] = []
todosMenosUltimo x = tomar ((long x) - 1, x)

todosMenosUltimoR1 :: [a] -> [a]
todosMenosUltimoR1 [] = []
todosMenosUltimoR1 (x:y) = if esVacia y then todosMenosUltimoR1 y else x : (todosMenosUltimoR1 y)

todosMenosUltimoR2 :: [a] -> [a]
todosMenosUltimoR2 [] = []
todosMenosUltimoR2 (x:[]) = []
todosMenosUltimoR2 (x:y) = x : (todosMenosUltimoR2 y)
```

7) Definir usando ejercicios anteriores la función **capicua**, que dada una lista de Char, determina si es o no capicúa.

```
capicua :: [Char] -> Bool
capicua x = x == rev x
```

8) Definir una función **xorl** que calcule el o exclusivo bit a bit habitual entre dos números en binario. Utilizar listas cuyos elementos representarán los dígitos de los números (ya que deben admitirse números con longitud variable).
Ejemplos:
xorl ([1,1,1], [1,1,0,1]) -> [1,0,1,0]
xorl ([0,1,0], [1,0,1,1,1,0]) -> [1,0,1,1,0,0]
xorl ([1,1,1,0,1,0,1], [1,1,0,1]) -> [1,1,1,1,0,0,0]

```
xorl :: ([Int], [Int]) -> [Int]
xorl ([], z) = z
xorl (z, []) = z
xorl (x:xs, y:ys) = if long xs > long ys then xorl (y:ys, x:xs)
                                         else (if long xs < long ys then y : xorl ((x:xs), ys) else (xor (x, y)) : xorl (xs, ys))

xor :: (Int, Int) -> Int
xor (x, y) = if x /= y then 1 else 0
```

Otra versión:

```
xorlDos :: ([Int], [Int]) -> [Int]
xorlDos a = rev (xorlRev a)

xorlRev :: ([Int], [Int]) -> [Int]
xorlRev ([], z) = z
xorlRev (z, []) = z
xorlRev (a, b) = ( xor (ultimo a, ultimo b) ) : xorlRev (todosMenosUltimoR2 a, todosMenosUltimoR2 b)
```

9) a) Representar una persona (nombre, edad y sexo) adecuadamente utilizando tuplas y/u otros tipos adecuados de Haskell.

```
type Nombre = String
type Edad = Int
type Sexo = Char
type Persona = (Nombre, Edad, Sexo)
```

9) b) Mediante una función, devolver el promedio de edades de una lista de personas dada.

```
promedioEdades :: [Persona] -> Int
promedioEdades p = (sumaEdades p) `div` (long p)

sumaEdades :: [Persona] -> Int
sumaEdades [] = 0
sumaEdades (x:y) = edad x + sumaEdades y

edad :: Persona -> Edad
edad (nombre, edad, sexo) = edad
```

Otra versión de suma edades:

```
sumaEdades2 :: [Persona] -> Int
sumaEdades2 [] = 0
sumaEdades2 ((n, e, s):y) = e + sumaEdades2 y
```

13) Definir la función **flat** que, dada una lista de listas, devuelva una lista con la concatenación de todas sus sublistas.
Ejemplo: flat [[1,8,5], [3,4], []] -> [1,8,5,3,4]

```
flat :: [[a]] -> [a]
flat [] = []
flat (x:y) = append (x, flat y)
```

14) Definir (usando flat) la función **long_ll** que, dada una lista de listas de algo, devuelva un número que sea la cantidad total de elementos acumulados de todas las sublistas.

```
long_ll :: [[a]] -> Int
long_ll x = long (flat x)
```

15) Definir la función **intercalar**, que hace una intercalación de dos listas genéricas, devolviendo una nueva lista.

```
intercalar :: ([a], [a]) -> [a]
intercalar (z, []) = z
intercalar ([], z) = z
intercalar (x:xs, y:ys) = x : (y : (intercalar(xs,ys)))
```

16) Definir la función **merge**, que devuelva un apareo de dos listas de enteros (asumiendo que ambas se encuentran ordenadas de menor a mayor), devolviendo una nueva lista ordenada.

```
merge :: ([Int], [Int]) -> [Int]
merge (z, []) = z
merge ([], z) = z
merge (x:xs, y:ys) = if x < y then x : (merge (xs, y:ys)) else y : (merge (x:xs, ys))
```

17) Formular adecuadamente y programar una función que realice el **buscar y reemplazar** para enteros en una lista (una versión con el reemplazo de todas las apariciones correspondientes, y otra versión con el reemplazo de la primera aparición del entero indicado en caso en que aparezca).

```
buscoyReemplazo :: (Int, Int, [Int]) -> [Int]
buscoyReemplazo (a, b, []) = []
buscoyReemplazo (a, b, (x:y)) = if a == x then b : buscoyReemplazo (a, b, y) else x : buscoyReemplazo (a, b, y)

buscoyReemplazoPrimerElemento :: (Int, Int, [Int]) -> [Int]
buscoyReemplazoPrimerElemento (a, b, []) = []
buscoyReemplazoPrimerElemento (a, b, (x:y)) = if a == x then b:y else x : buscoyReemplazoPrimerElemento (a, b, y)
```

18) Definir una función **dec_a_hex**, que dado un número devuelva una lista de caracteres correspondiente al parámetro representado en hexadecimal.

Ejemplos:
dec_a_hex 27514 -> 6B7A
dec_a_hex 234829 -> 3954D

```
dec_a_hex :: Int -> [Char]
dec_a_hex x = rev (dec_a_hexRev x)

dec_a_hexRev :: Int -> [Char]
dec_a_hexRev 0 = []
dec_a_hexRev n = (convertHex(n `mod` 16)) : (dec_a_hexRev (n `div` 16))

convertHex :: Int -> Char
convertHex 0 = '0'
convertHex 1 = '1'
convertHex 2 = '2'
convertHex 3 = '3'
convertHex 4 = '4'
convertHex 5 = '5'
convertHex 6 = '6'
convertHex 7 = '7'
convertHex 8 = '8'
convertHex 9 = '9'
convertHex 10 = 'A'
convertHex 11 = 'B'
convertHex 12 = 'C'
convertHex 13 = 'D'
convertHex 14 = 'E'
convertHex 15 = 'F'
```

19) Definir una función **mayoría** que dada una lista de enteros indique cuál es el que más veces aparece. En caso de empate/s, se considera el primero que aparezca un máximo de veces.

```
mayoria :: [Int] -> Int
mayoria (x:[]) = x
mayoria (x:(y:z)) = if cuantas(x, (x:(y:z))) < cuantas(y, (x:(y:z))) then mayoria(y:z) else mayoria(x:z)

cuantas :: (Int, [Int]) -> Int
cuantas (n, []) = 0
cuantas (n, (x:y)) = if n == x then 1 + cuantas (n, y) else cuantas (n, y)
```

Otra versión:

```
mayoria2 :: [Int] -> Int
mayoria2 (x:[]) = x
mayoria2 (x:y) = if cuantas(x, x:y) >= cuantas(mayoria2 y, x:y) then x else mayoria2 y
```

20) Definir una función **eeeo** ("están en este orden") eeeo que tome dos strings y determine si es cierto o no que los chars del primero aparecen en el mismo orden en el segundo, sin importar si hay otros chars intercalados.

```
estanEnEsteOrden :: ([Char],[Char]) -> Bool
estanEnEsteOrden ([],y) = True
estanEnEsteOrden (x,[]) = False
estanEnEsteOrden ((x:xs),(y:ys)) = if x == y then estanEnEsteOrden(xs, ys) else estanEnEsteOrden((x:xs),ys)
```

21) Definir la función **es_prefija**, que verifica si una lista es prefija de otra lista. Como casos particulares asumir que cualquier lista es prefija de sí misma y la lista vacía es prefija de cualquier lista.

```
es_prefija :: ([Int],[Int]) -> Bool
es_prefija ([], y) = True
es_prefija (x, []) = False
es_prefija ((x:xs),(y:ys)) = if x == y then es_prefija (xs, ys) else False
```

22) Definir en Haskell una función que, dadas dos listas, determine si la primera es una sublista de la segunda; es decir si los elementos de la primera aparecen consecutivos en alguna parte de la segunda.

```
esSubLista :: ([Int],[Int]) -> Bool
esSubLista ([],[]) = True
esSubLista (x,[]) = False
esSubLista (x,y) = if es_prefija (x,y) then True else esSubLista (x, cola y)
```

23) Definir una función **posicion**, que devuelve la posición de una sublista dentro de otra lista.

```
posicion :: ([Int],[Int]) -> Int
posicion ([], y) = -1
posicion (x, y) = if esSubLista(x,y) then damePosicion(x,y) else -1

damePosicion :: ([Int],[Int]) -> Int
damePosicion (x, []) = 0
damePosicion (x, y) = if es_prefija (x,y) then 1 else 1 + damePosicion (x, cola y)
```

24) Definir genéricamente la función **asoc**, que devuelva el valor asociado a otro valor (string), según una lista de pares "valor, valor asociado". Si no lo encuentra, que se genere un error.

```
asoc :: ([([Char], a)], [Char]) -> a
asoc (x:y, valorABuscar) = if encontreValor(dameValor x, valorABuscar) then dameValorAsociado x else asoc(y, valorABuscar)

dameValor :: ([Char], a) -> [Char]
dameValor (valor, valorAsociado) = valor

encontreValor :: ([Char], [Char]) -> Bool
encontreValor (x, y) = x == y
```

```
dameValorAsociado :: ([Char], a) -> a
dameValorAsociado (valor, valorAsociado) = valorAsociado
```

Otra versión:

```
asoc :: ([([Char], a)], [Char]) -> a
asoc ((x,y):z, v) = if x == v then y else asoc(z, v)
```

25) Para una lista cualquiera de personas (representadas usando 3-uplas, con nombre, edad y sexo para cada persona), se desea saber cuáles tienen edad igual a 1 año más que alguna otra de la lista.

```
edad1AnioMasQueOtro :: [Persona] -> [Nombre]
edad1AnioMasQueOtro [] = []
edad1AnioMasQueOtro x = dameNombres(damePersonas1AnioMas(ordenarPorEdad x))

dameNombres :: [Persona] -> [Nombre]
dameNombres [] = []
dameNombres (x:y) = nombre(x) : dameNombres(y)

nombre :: Persona -> Nombre
nombre (nombre, edad, sexo) = nombre

damePersonas1AnioMas :: [Persona] -> [Persona]
damePersonas1AnioMas [] = []
damePersonas1AnioMas (x:[]) = []
damePersonas1AnioMas (x:(y:z)) = if (edad x) + 1 == edad y then y : damePersonas1AnioMas(y:z)
                                                           else damePersonas1AnioMas(y:z)

ordenarPorEdad :: [Persona] -> [Persona]
ordenarPorEdad [] = []
ordenarPorEdad (x:[]) = [x]
ordenarPorEdad (x:(y:z)) = if edad x < edad y then x : ordenarPorEdad(y:z) else y : ordenarPorEdad(x:z)
```

27) Definir la función **clasificaPar**, que dada una lista de enteros devuelva una lista de los mismos números en donde aparecen primero todos los elementos pares y después todos los elementos impares de la lista original.

```
clasificaPar :: [Int] -> [Int]
clasificaPar x = append ((listaPar x), (listaImpar x))

listaPar :: [Int] -> [Int]
listaPar [] = []
listaPar (x:y) = if esPar x then x : listaPar y else listaPar y

listaImpar :: [Int] -> [Int]
listaImpar [] = []
listaImpar (x:y) = if esPar x then listaImpar y else x : listaImpar y

esPar :: Int -> Bool
esPar x = x `mod` 2 == 0
```

28) Definir lo más genéricamente posible una función **empareja**, que tome dos listas, y un elemento, y devuelva las dos listas, pero en donde a la lista de menor longitud se le agregue dicho elemento a izquierda tantas veces como haga falta para que las longitudes de ambas listas devueltas sean las mismas.

```
empareja :: ([a], [a], a) -> ([a], [a])
empareja ([], [], e) = ([], [])
empareja (x, y, e) = if long x < long y then ((agregarElementos(x, long y - long x, e)), y)
                                        else (x, (agregarElementos(y, long x - long y, e)))

agregarElementos :: ([a], Int, a) -> [a]
agregarElementos (lista, 0, e) = lista
agregarElementos (lista, n, e) = e : agregarElementos(lista, n-1, e)
```

29) Definir en Haskell una función que dadas dos listas devuelva los elementos que aparecen en la primera más veces que en la segunda, evitando en cualquier caso devolver elementos repetidos.

```
masEnPrimeraL :: ([Int], [Int]) -> [Int]
masEnPrimeraL ([], y) = []
masEnPrimeraL (x:xs, y) = if cuantas(x, x:xs) > cuantas(x, y)
                          then x : masEnPrimeraL(borrarDeLista(x, x:xs), y)
                          else masEnPrimeraL(borrarDeLista(x, x:xs), y)

borrarDeLista :: (Int, [Int]) -> [Int]
borrarDeLista (e, []) = []
borrarDeLista (e, x:y) = if x /= e then x : borrarDeLista(e, y) else borrarDeLista(e, y)
```

30) Escribir una función **precedenA0** que, dada una lista de enteros, devuelva la suma de aquellos que precedan inmediatamente a un 0 en la lista.

```
precedenA0 :: [Int] -> Int
precedenA0 [] = 0
precedenA0 (x:[]) = 0
precedenA0 (x:y) = if cabeza y == 0 then x + precedenA0 y else precedenA0 y
```

31) Definir la función **diagSec**, la que, dada una matriz (cuadrada) de cualquier tamaño y con cualquier contenido, devuelva una lista conteniendo la diagonal secundaria, "de abajo hacia arriba".

```
diagSec :: [[a]] -> [a]
diagSec [] = []
diagSec x = diagPrincipal(rev x)
```

<u>Otra opción:</u>

```
diagSec2 :: [[a]] -> [a]
diagSec2 [] = []
diagSec2 x = rev(diagSecInv x)

diagSecInv :: [[a]] -> [a]
diagSecInv [] = []
diagSecInv (x:y) = (ultimo x) : diagSecInv (todosMenosUltimoL y)

todosMenosUltimoL :: [[a]] -> [[a]]
todosMenosUltimoL [] = []
todosMenosUltimoL (x:y) = todosMenosUltimo x : todosMenosUltimoL y
```

31') Calcular diagonal principal.

```
diagPrincipal :: [[a]] -> [a]
diagPrincipal [] = []
diagPrincipal (x:y) = cabeza x : diagPrincipal (colaL y)

colaL :: [[a]] -> [[a]]
colaL [] = []
colaL (x:y) = cola x : colaL y
```

32) Definir una función que, dada una matriz genérica (de m x n elementos) devuelva la misma matriz rotada 180°.
Ejemplo: rotarMat180 [[1,2,3],[4,5,6],[7,8,9]] -> [[9,8,7],[6,5,4],[3,2,1]]

```
rotarMat180 :: [[a]] -> [[a]]
rotarMat180 [] = []
rotarMat180 x = revL (rev x)

revL :: [[a]] -> [[a]]
revL [] = []
revL (x:y) = rev x : revL y
```

32') Rotar matriz 90 grados.
Ejemplo: rotarMat90 [[1,2,3],[4,5,6],[7,8,9]] -> [[7,4,1],[8,5,2],[9,6,3]]

```
rotarMat90 :: [[a]] -> [[a]]
rotarMat90 [] = []
rotarMat90 x = revL (transponerMat x)
```

33) Definir una función que, dada una matriz genérica (de m x n elementos) devuelva la matriz transpuesta (es decir, las viejas columnas se transforman en las nuevas filas, y viceversa).
Ejemplo: transponerMat [[2,3,0],[1,2,0],[3,5,6]] -> [[2,1,3],[3,2,5],[0,0,6]]

```
transponerMat :: [[a]] -> [[a]]
transponerMat [] = []
transponerMat ([]:y) = []
transponerMat z = cabezaL z : transponerMat (colaL z)

cabezaL :: [[a]] -> [a]
cabezaL [] = []
cabezaL (x:y) = cabeza x : (cabezaL y)
```

34) Definir una función que tome una matriz de enteros cuadrada (de n × n para cualquier n >= 1) y determine si los elementos de la diagonal principal son todos iguales.

```
elementosDiagIguales :: [[Int]] -> Bool
elementosDiagIguales x = todosIguales (diagPrincipal x)

todosIguales :: [Int] -> Bool
todosIguales (x:[]) = True
todosIguales (x:y) = if x == cabeza y then todosIguales y else False
```

35) Definir una función **columnas**, que dada una matriz (rectangular, de cualquier tamaño y contenido), y dos enteros n y m, devuelva la matriz formada por las columnas que están entre la columna n-ésima hasta la m-ésima inclusive.

```
columnas :: ([[a]], Int, Int) -> [[a]]
columnas ([], n, m) = []
columnas ((x:y), n, m) = (dameListaDesdeHasta(x, n, m)) : (columnas (y, n, m))

dameListaDesdeHasta :: ([a], Int, Int) -> [a]
dameListaDesdeHasta (lista, n, m) = dameListaHasta(dameListaDesde(lista, n), m)

dameListaDesde :: ([a], Int) -> [a]
dameListaDesde (x, 1) = x
dameListaDesde ([], y) = []
dameListaDesde ((x:y), n) = dameListaDesde (y, n-1)

dameListaHasta :: ([a], Int) -> [a]
dameListaHasta (x, 1) = []
dameListaHasta ([], y) = []
dameListaHasta ((x:y), n) = x : (dameListaHasta (y, n-1))
```