Trabajo Práctico 3 **Prog. Avanzada**

Ejercicio 1

Define una función que, dadas dos listas **ys** y **xs** de naturales ordenadas, retorne el *merge* de estas listas, es decir, la lista ordenada compuesta por los elementos de **ys** y **xs**.

Respuesta

```
merge :: [Int] -> [Int] -> [Int]
merge [] [] = []
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
                  |x <= y = x : merge xs (y:ys)
                  |otherwise =  y: merge (x:xs) ys
```

Ejercicio 2

Define una función que, dada una lista de naturales, la ordene.

Respuesta

```
ordNum :: [Int] -> [Int]
ordNum [] = []  -- caso base del vacio
ordNum (x:xs) = ordNum left ++ [x] ++ ordNum right
                where
                    left = [a | a <- xs, a <= x]
                    right = [b | b <- xs, b > x]
```

Ejercicio 3

Define una función que, recursivamente y sólo utilizando adición y multiplicación, calcule , dado un natural *n*, el número 2^n.

Respuesta

```
dosElev :: Int -> Int
dosElev 0 = 1
dosElev n = 2 * (dosElev (n-1))
```

Ejercicio 4

Define una función que, dado un número natural *n*, retorne su representación binaria como secuencia de bits.

Respuesta

```
numBits :: Int -> [Int]
numBits 0 = []
numBits n = numBits (n `div` 2) ++ [n `mod` 2]
```

Ejercicio 5

Define una función que, dado un número natural *n* en su representación binaria, decida si *n* es par o no.

Respuesta

```haskell
binaryPar :: [Int] -> Bool  --dado un numero binario (en lista) retorna si es par o no
binaryPar [] = False
binaryPar xs = last xs `mod` 2 == 0
```

Ejercicio 6

Define la función que retorne la distancia de Hamming: dadas dos listas es el número de posiciones en que los correspondientes elementos son distintos. Por ejemplo: distanciaH “*roma*” “*camino*” -> 3

Respuesta

```haskell
distanciaH :: Eq a => [a] -> [a] -> Int
distanciaH [] [] = 0    --evalua el primer argumento
distanciaH xs [] = 0
distanciaH [] ys= 0
distanciaH (x:xs) (y:ys)
            |x /= y = 1 + distanciaH xs ys
            |otherwise = distanciaH xs ys
```

Ejercicio 7

Define la función que, dado un número natural, decida si el mismo es un cuadrado perfecto o no.

Respuesta

```haskell
cuadPerfect :: Int -> Bool
cuadPerfect n = (length [x | x<-[1..n], (x^2 == n)]) > 0

--Otro metodo
cuadPerfectDos :: Int -> Bool
cuadPerfectDos n = [x | x<-[0..n], x*x == n] /= []
```

Ejercicio 8

Define la función repetidos de forma tal que dado un elemento **z** y un entero **n**; aparece **n** veces.

Respuesta

```haskell
repetiDos :: Ord a => a -> Int -> [a]
repetiDos a 0 = []
repetiDos a 1 = [a]
repetiDos a n = repetiDos a (n-1) ++ [a]
```

Ejercicio 9

Define la función **nElem** tal que **nElem xs n** es elemento en-ésimo de **xs,** empezando a numerar desde el 0. Por ejemplo:

nElem [1,3,2,4,9,7] 3 -> 4

Respuesta

```
nElem :: [Int] -> Int -> Int
nElem [] n = undefined
nElem (x:xs) 0 = x
nElem (x:xs) n = nElem xs (n-1)
```

Ejercicio 10

Define la función **posicionesC** tal que **posicionesC xs c** es la lista de la posiciones del carácter **c** en la cadena xs. Por ejemplo:

posicionesC “Catamarca” ‘a’ == [1,3,5,8]

Respuesta

```
posicionesC :: [Char] -> Char -> [Int]
posicionesC [] c = []
posicionesC xs c = [i | (x,i) <- zip xs [0..], c == x ]
```

Ejercicio 11

Define la función **compact**, dada una lista retorna la lista sin los elementos repetidos consecutivos. Por ejemplo: **compact [1,3,3,5,8,3] = [1,3,5,8,3]**

Respuesta

```
functcompac :: [Int] -> [Int]
functcompac [] = []
functcompac [x] = [x]
functcompac (x:xs)
                  | x /= head xs = x : functcompac xs
                  | otherwise = functcompac xs
```