Resolução

Lista PLC

1. Defina uma função que, dados dois números x e y, retorne o m.d.c de x e y.

   R:

```
mdc :: Integer -> Integer -> Integer
mdc n m
    | mod (max n m) (low n m) == 0 = low n m
    | otherwise = mdc (low m n) (mod (max n m) (low n m))

low :: Integer -> Integer -> Integer
low n m
    | m <= n = m
    | otherwise = n
```

2. Defina uma função que, dado um número inteiro positivo x, verifique se x é primo ou não. Lembre-se de utilizar o crivo de Eratóstenes por razões de otimização.

   R:

```
isPrime :: Integer -> Bool
```

```
isPrime 1 = True

isPrime 2 = True

isPrime n = checkPrimes 2 (ceiling $ sqrt (fromIntegral n)) n


checkPrimes :: Integer -> Integer -> Integer -> Bool

checkPrimes chao teto n

                    | mod n chao == 0 = False

                    | chao == teto = True

                    | otherwise = checkPrimes (chao+1) teto n
```

3. Dados dois pontos num espaço tridimensional, defina uma função distância e um tipo Ponto de tal forma que a função calcule a distância entre dois pontos passados como parâmetros. A função tem tipo Ponto -> Ponto -> Double.

   R:

4. Usando compreensão de lista, define uma expressão que calcule a soma 12+22+...+1002 dos quadrados dos cem primeiros inteiros positivos.

   R:

   let a = [1..100]

```
powerOfAList :: [Integer] -> [Integer]

powerOfAList n = [floor ((fromIntegral x)**2) | x <- n]
```

5. Suponha que uma grade de coordenadas de tamanho m x n is dada por uma lista de todos os pares (x,y) tal que 0≤x≤m e 0≤y≤n. Usando compreensão de lista, defina a função grid:: Int -> Int -> [(Int, Int)] que retorna uma grade de um dado tamanho.

R:

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

grid n m = [(x,y) | x <- [0..n], y <- [0..m]]
```

6. Usando compreensão de lista e a função grid definida acima, defina a função square :: Int -> [(Int, Int)] que retorna as coordenadas do quadrado de tamanho n, excluindo a diagonal de (0,0) a (n,n).

7. Defina a função recursiva merge :: Ord a => [a] -> [a] -> [a] que mescla duas listas ordenadas em uma única lista ordenada.

R:

```
import Data.List
```

```
merge :: Ord a => [a] -> [a] -> [a]
merge n m
    | sort n == n && sort m == m = sort (n ++ m)
    | otherwise = []
```

8. Usando a função merge, defina a função msort :: Ord a => [a] -> [a] que implementa merge sort, no qual uma lista vazia e uma lista com um único elemento estão ordenadas; qualquer outra lista é ordenada por mesclar as duas listas que resultam de ordenar as duas metades da lista separadamente.

R:

```
import Data.List

msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort n = merge (msort (left)) (msort (right))
                        where (left, right) = split n

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

```
merge n m
    | sort n == n && sort m == m = sort (n ++ m)
    | otherwise = []


split :: [a] -> ([a],[a])
split n = (take half n, drop half n)
                    where half = div (length n) 2
```

9. Defina e implemente a função aplicaFuncoes :: [Int->Int] -> [Int] -> [[Int]] que recebe uma lista de funções unárias e uma lista de valores e retorna uma lista de listas na qual cada lista contém a lista de valores recebidos na entrada aplicada a uma das funções.

R:

```
aplicaFuncoes :: [Int->Int] -> [Int] -> [[Int]]
aplicaFuncoes [] _ = []
aplicaFuncoes fx@(f:fs) list@(x:xs) = aplicaFunc f list : aplicaFuncoes fs list
```

```
aplicaFunc :: (Int -> Int) -> [Int] -> [Int]

aplicaFunc _ [] = []

aplicaFunc fx list@(x:xs) = fx x : aplicaFunc fx xs
```

10. Implemente um tipo algébrico DiasSemana, começando do Domingo e indo até Sábado, instancie ele para Enum, Show, Ord e Eq e crie 3 funções:

ordenaUteis :: [DiasSemana] -> [DiasSemana]
--Dada uma lista de dias da semana retorna os dias úteis ordenados;

datasIguais :: [(DiasSemana,Int)] -> DiasSemana -> [Int]
--Dada uma lista de tuplas (Dia da semana, data) e um dia da semana, retorna uma lista com somente as datas que sejam do mesmo dia da semana;

imprimeMes :: DiasSemana -> [(Int, DiasSemana)]
--Recebe um dia da semana (referente ao primerio dia do mes) e retorna uma lista de tuplas com todos os dias do mês. Considere o mês como tendo 30 dias.

R :

```
import Data.List

import Data.Function
```

```
data DiasSemana = Domingo | Segunda | Terça | Quarta | Quinta | Sexta | Sabado
deriving (Enum, Show, Ord, Eq)


ordenaUteis :: [DiasSemana] -> [DiasSemana]

ordenaUteis [] = []

ordenaUteis n@(x:xs) = sort $ filter isUtil n


datasIguais :: [(DiasSemana, Int)] -> DiasSemana -> [Int]

datasIguais [] _ = []

datasIguais n@(x:xs) dia

                | fst x == dia = snd x : datasIguais xs dia

                | otherwise = datasIguais xs dia


proximo :: DiasSemana -> DiasSemana

proximo n
```

```
        | n == Domingo = Segunda
        | n == Segunda = Terça
        | n == Terça = Quarta
        | n == Quarta = Quinta
        | n == Quinta = Sexta
        | n == Sexta = Sabado
        | otherwise = Domingo


imprimeMes :: DiasSemana -> [(Int,DiasSemana)]
imprimeMes n = util 1 n


util :: Int -> DiasSemana -> [(Int, DiasSemana)]
util 31 _ = []
util dia semana = (dia, semana) : util (dia+1) (proximo semana)
```

```
isUtil :: DiasSemana -> Bool

isUtil n

    | n == Domingo || n == Sabado = False

    | otherwise = True
```