1) Dada uma lista de Int, retorne uma dupla com os elementos da lista de entrada que são primos em ordem não decrescente sem repetições e uma lista de Bool informando, para cada Int da lista de entrada(na ordem de entrada), se ele é primo ou não.

```
isPrime :: Int -> Bool
isPrime k = length [x | x <- [1..k], k `mod` x == 0] == 2

primeList :: [Int] -> [Bool]
primeList [] = []
primeList (x:xs) = [isPrime x] ++ primeList(xs)

qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs) = qSort [y | y <- xs,  y < x] ++ [x] ++ qSort [y| y <- xs, y > x]

primes :: [Int] -> ([Int],[Bool])
primes x = (qSort([y | y <- x, isPrime y]), primeList x)

Outra solução:
primos :: [Int] -> ([Int], [Bool])
primos list = (x, y)
  where
    x = [a | a <-list, isPrime a]
    y = map isPrime list
```

Outra solução maior porém mais fácil:

```
countDiv :: Int -> Int -> Int
countDiv n x
    | x == 1 = 1
    | (n `mod` x == 0) = 1 + countDiv (n) (x - 1)
    | otherwise = countDiv (n) (x - 1)

isPrime :: Int -> Bool
isPrime n = n > 1 && countDiv n n <= 2

listPrimes :: [Int] -> [Int]
listPrimes list
    | list == [] = []
    | isPrime (head list) = (head list) : listPrimes (tail list)
    | otherwise = listPrimes (tail list)

boolPrimes :: [Int] -> [Bool]
boolPrimes list
    | list == [] = []
    | isPrime (head list) = True : boolPrimes (tail list)
    | otherwise = False : boolPrimes (tail list)

q1 :: [Int] -> ([Int], [Bool])
q1 list = (listPrimes list, boolPrimes list)
```

Solução mais simples:

```
primos lista = (filter primo lista, map primo lista)

primo x = (length (divisores x)) < 2

divisores x = [a | a <- [2..x], (mod x a) == 0]
```

2) Dados dois números inteiros A e B(A<=B), retorne a quantidade de números primos existentes no intervalo fechado deles.

```
isPrime :: Int -> Bool
isPrime k = length [x | x <- [1..k], k `mod` x == 0] == 2

primesBetween :: Int -> Int -> Int
primesBetween a b = length([x | x <- [a..b], isPrime x])
```

Outra solução maior porém mais fácil:

```
countDiv :: Int -> Int -> Int
countDiv n x
   | x == 1 = 1
   | (n `mod` x == 0) = 1 + countDiv (n) (x - 1)
   | otherwise = countDiv (n) (x - 1)

isPrime :: Int -> Bool
isPrime n = n /= 1 && countDiv n n <= 2

q2 :: Int -> Int -> Int
q2 a b
   | b == a - 1 = 0
   | isPrime b = 1 + q2 (a) (b - 1)
   | otherwise = q2 (a) (b - 1)
```

Solução mais simples:

```
primoEntre a b = length (filter primo [a..b])

primo x = (length (divisores x)) < 2

divisores x = [a | a <- [2..x], (mod x a) == 0]
```

3) Dada uma lista L de números naturais e um inteiro K retorne o Késimo elemento de L ou -1 caso o elemento K não exista.

```
kFind :: [Int] -> Int -> Int
kFind x k | (k-1) < length x = x !! (k-1)
          | otherwise = -1
```

4) Dado um número inteiro > 2, escreva uma função que retorna uma dupla contendo os dois números primos que somados resultam em um dado inteiro par > 2. Não importa a ordem na dupla.

```
isPrime :: Int -> Bool
isPrime k = length [x | x <- [1..k], k `mod` x == 0] == 2

findSum :: Int -> (Int, Int)
findSum k = [(x,y) | x <- [2..k], y <- [2..k], isPrime x, isPrime y, (x+y)==k]!!0
```

Outra solução:

```
findPrimes a = if (mod a 2 == 0) then escolhePar a else (0,0)
escolhePar a = head [(x,y) | x <- [2.. a], y <- [2..a] , y + x == a, isPrime x, isPrime y]
```

Outra solução:

```
findPrimes x = head [(a, b) | a <- filter primo [2..x], b <- filter primo [1..x], (a + b) == x]

primo x = (length (divisores x)) < 2

divisores x = [a | a <- [2..x], (mod x a) == 0]
```

5) Crie a função mergesort :: [Int] > [Int] , a qual ordena uma lista de números inteiros.

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

```
mergesort :: [Int] -> [Int]
mergesort as = juntar (dividir as)

-- Junta as listas
juntar :: [[Int]] -> [Int]
juntar (a:as) = conquistar a (juntar as)
juntar _ = []

-- Transforma um array de Int em um array de arrays com 1 Int cada
dividir :: [Int] -> [[Int]]
dividir (a:as) = [[a]] ++ dividir as
dividir _ = []
```

```
-- Junta ordenadamente duas listas ordenadas
conquistar (a:as) (b:bs)
 | a < b = a : conquistar as (b:bs)
 | otherwise = b : conquistar (a:as) bs
conquistar (a:as) _ = (a:as)
conquistar _ (b:bs) = (b:bs)
```

Outra solução:

```
mergesort [] = []
mergesort [x] = [x]
mergesort list = intercala (mergesort (take (length list `div` 2) list))  (mergesort (drop (length list `div` 2) list))

intercala [] [] = []
intercala a [] = a
intercala [] b = b
intercala (a:as) (b:bs) = if (a > b) then b: intercala (a:as) bs
          else a: intercala as (b:bs)
```

6) Crie a função somaDig :: Int > Int , a qual retorna a soma dos dígitos de um número inteiro.

```
sumDig :: Int -> Int
sumDig 0 = 0
sumDig k = k`mod`10 + sumDig(k`div`10)

Outra Solução:
sumDig :: Int -> Int
sumDig x = foldr (+) 0 [read [a] :: Int | a <- (show x)]
```

7) Dado uma lista de lista de inteiros, retorne uma lista contendo a soma de todos os elementos de cada lista da lista de entrada. O uso de map e foldr é proibido.

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList(xs)

sumMultList :: [[Int]] -> [Int]
sumMultList [] = []
sumMultList (x:xs) = [sumList(x)] ++ sumMultList(xs)
```

Outra solução:

```
innerSum :: [[Int]] -> [Int]
innerSum x = [(sum a) | a <- x]

Outra Solução:
sumListsValue :: [[Int]] -> [Int]
sumListsValue [] = []
sumListsValue list = [ summa a | a <- list]
  where
    summa [] = 0
    summa (x:xs) = x + summa xs
```

8) Dado um número não negativo N, calcule a soma dos números Tribonacci(N) com Fibonacci(N) com Lucas(N).

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib k = fib(k-1) + fib(k-2)

trib :: Int -> Int
trib 0 = 1
trib 1 = 1
trib 2 = 2
trib k = trib(k-1) + trib(k-2) + trib(k-3)

luc :: Int -> Int
luc 0 = 2
luc 1 = 1
luc k = luc(k-1) + luc(k-2)

fsum :: Int -> Int
fsum k = fib(k) + trib(k) + luc(k)
```

Links para estudo:

http://www.imada.sdu.dk/~rolf/Edu/DM22/F06/haskell-operatorer.pdf
https://gist.github.com/morae/8494016