19CSE313

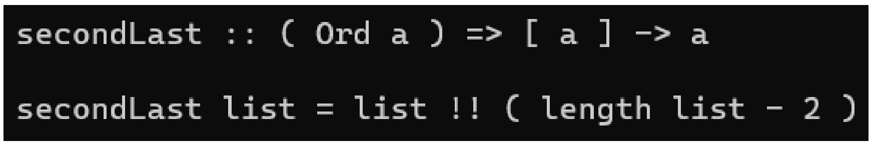
Principles of Programming Languages

Lab 5

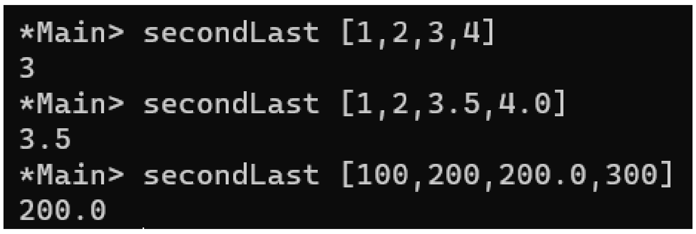
S Abhishek

AM.EN.U4CSE19147

1 - Write a script to find the last but one element of a list.

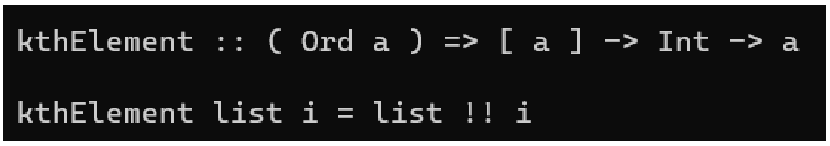


```
secondLast :: ( Ord a ) => [ a ] -> a

secondLast list = list !! ( length list - 2 )
```

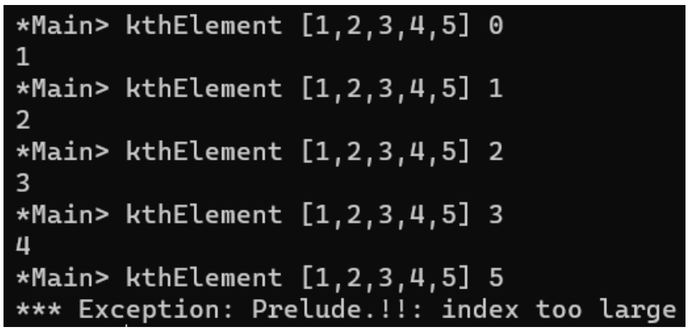


```
*Main> secondLast [1,2,3,4]
3
*Main> secondLast [1,2,3.5,4.0]
3.5
*Main> secondLast [100,200,200.0,300]
200.0
```

2 - Write a script to find the kth element of list, where k is the index.

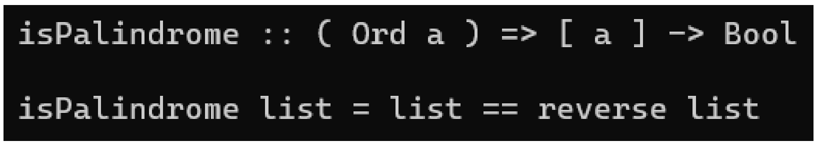


```
kthElement :: ( Ord a ) => [ a ] -> Int -> a

kthElement list i = list !! i
```

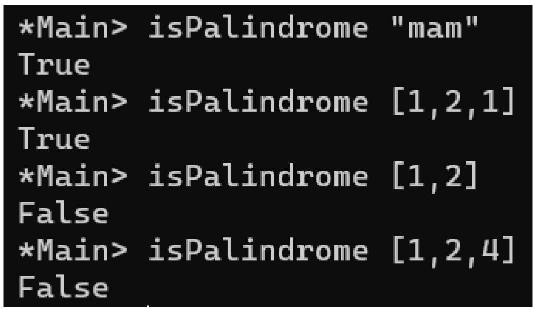


```
*Main> kthElement [1,2,3,4,5] 0
1
*Main> kthElement [1,2,3,4,5] 1
2
*Main> kthElement [1,2,3,4,5] 2
3
*Main> kthElement [1,2,3,4,5] 3
4
*Main> kthElement [1,2,3,4,5] 5
*** Exception: Prelude.!!: index too large
```

3 - Write a script to find out whether a list is a palindrome.

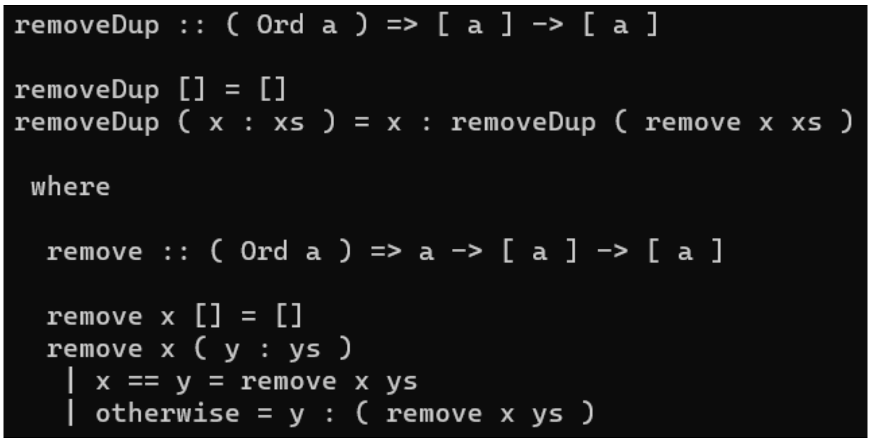


```haskell
isPalindrome :: ( Ord a ) => [ a ] -> Bool

isPalindrome list = list == reverse list
```

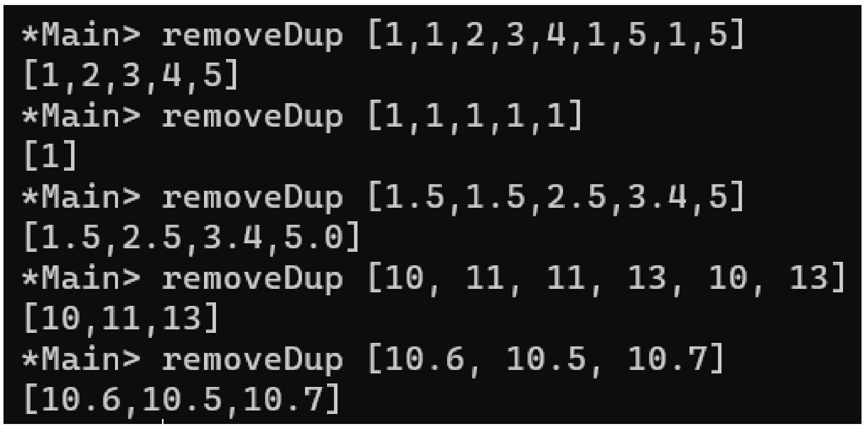


```
*Main> isPalindrome "mam"
True
*Main> isPalindrome [1,2,1]
True
*Main> isPalindrome [1,2]
False
*Main> isPalindrome [1,2,4]
False
```

4 - Write a script to remove duplicates from a given list.

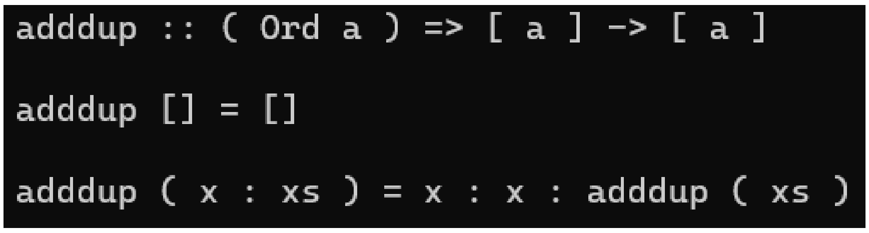


```haskell
removeDup :: ( Ord a ) => [ a ] -> [ a ]

removeDup [] = []
removeDup ( x : xs ) = x : removeDup ( remove x xs )

 where

  remove :: ( Ord a ) => a -> [ a ] -> [ a ]

  remove x [] = []
  remove x ( y : ys )
   | x == y = remove x ys
   | otherwise = y : ( remove x ys )
```

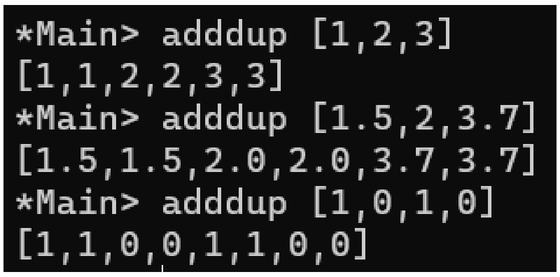


```
*Main> removeDup [1,1,2,3,4,1,5,1,5]
[1,2,3,4,5]
*Main> removeDup [1,1,1,1,1]
[1]
*Main> removeDup [1.5,1.5,2.5,3.4,5]
[1.5,2.5,3.4,5.0]
*Main> removeDup [10, 11, 11, 13, 10, 13]
[10,11,13]
*Main> removeDup [10.6, 10.5, 10.7]
[10.6,10.5,10.7]
```

5 - Define a function duplicate which will duplicate each element of the list and produce a new list.

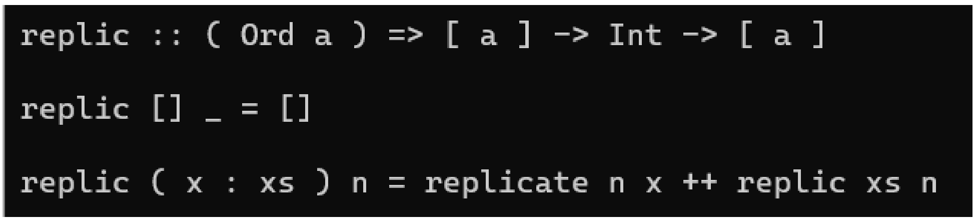


```
adddup :: ( Ord a ) => [ a ] -> [ a ]

adddup [] = []

adddup ( x : xs ) = x : x : adddup ( xs )
```

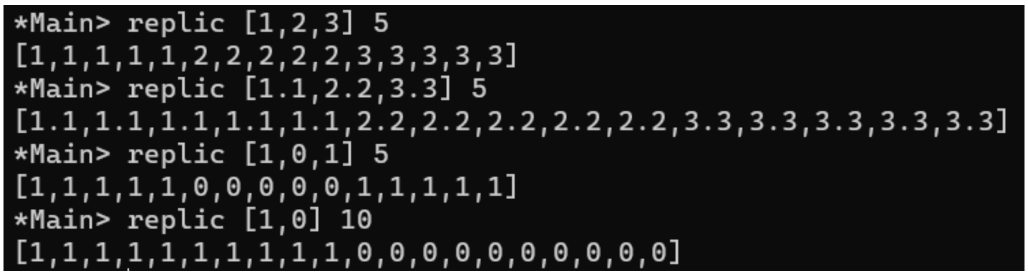


```
*Main> adddup [1,2,3]
[1,1,2,2,3,3]
*Main> adddup [1.5,2,3.7]
[1.5,1.5,2.0,2.0,3.7,3.7]
*Main> adddup [1,0,1,0]
[1,1,0,0,1,1,0,0]
```

6 - Define a function to replicate the elements of a list n times.



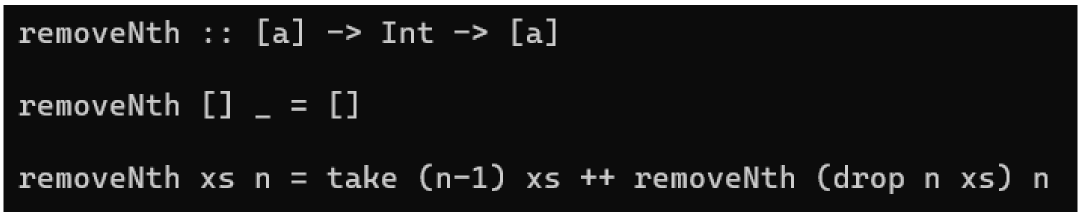
```
replic :: ( Ord a ) => [ a ] -> Int -> [ a ]

replic [] _ = []

replic ( x : xs ) n = replicate n x ++ replic xs n
```



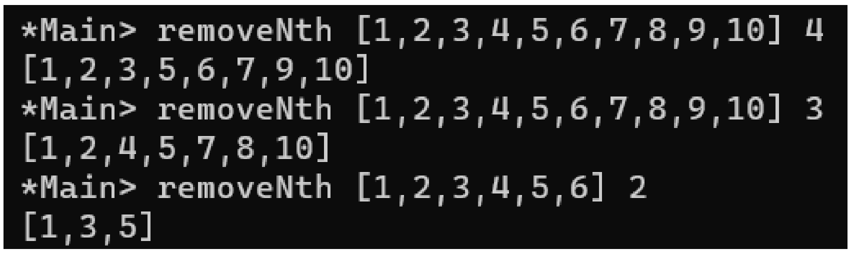
```
*Main> replic [1,2,3] 5
[1,1,1,1,1,2,2,2,2,2,3,3,3,3,3]
*Main> replic [1.1,2.2,3.3] 5
[1.1,1.1,1.1,1.1,1.1,2.2,2.2,2.2,2.2,2.2,3.3,3.3,3.3,3.3,3.3]
*Main> replic [1,0,1] 5
[1,1,1,1,1,0,0,0,0,0,1,1,1,1,1]
*Main> replic [1,0] 10
[1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0]
```

7 - Write a function to remove every nth element from the list.

- Let [1,2,3,4,5,6,7,8,9,10] be the list and value of n be 3, then the resultant list will be [1,2,4,5,7,8,10](after every nth element of the input list is [1,2,3,4,5,6,7,8,9,10] removed).



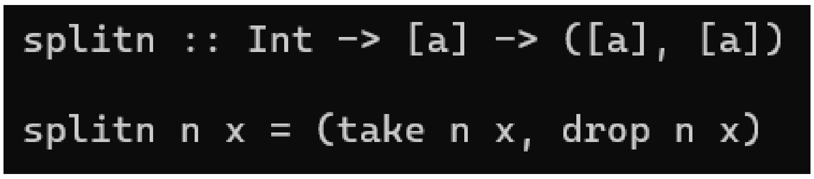
```haskell
removeNth :: [a] -> Int -> [a]

removeNth [] _ = []

removeNth xs n = take (n-1) xs ++ removeNth (drop n xs) n
```



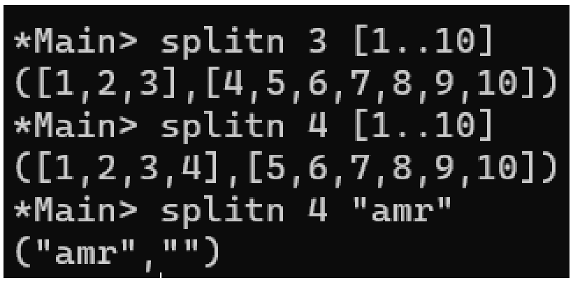
```
*Main> removeNth [1,2,3,4,5,6,7,8,9,10] 4
[1,2,3,5,6,7,9,10]
*Main> removeNth [1,2,3,4,5,6,7,8,9,10] 3
[1,2,4,5,7,8,10]
*Main> removeNth [1,2,3,4,5,6] 2
[1,3,5]
```

8 - Split a list by defining a function which takes an input list and an integer n and divides the list into two the first n elements as the first list and the rest as the second list and form a list of lists.

- Let [1..10] be a list and value of n be 4 then the new list formed is 1,2,3,4],[5,6,7,8,9,10]
- Another example I/P splits "amr" 4 & O/P- ["amr",""]



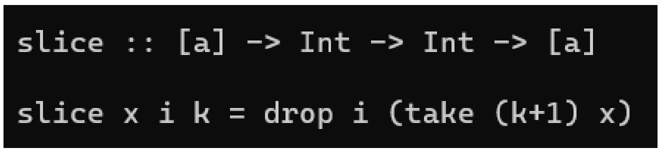
```haskell
splitn :: Int -> [a] -> ([a], [a])

splitn n x = (take n x, drop n x)
```



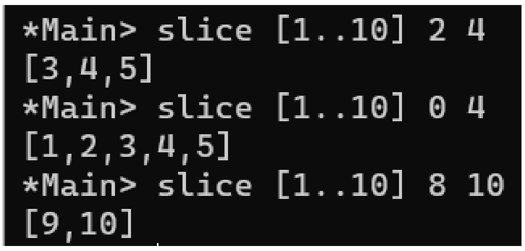
```
*Main> splitn 3 [1..10]
([1,2,3],[4,5,6,7,8,9,10])
*Main> splitn 4 [1..10]
([1,2,3,4],[5,6,7,8,9,10])
*Main> splitn 4 "amr"
("amr","")
```

9 - Define a function that will slice a list based on the input indices i and k.

- Consider a list [1..10] and let i = 2 and k = 4 respectively then the resultant list will be [3,4,5].



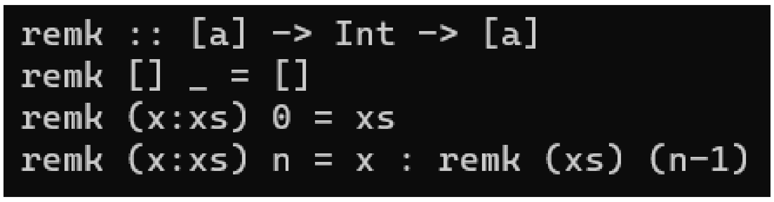
```
slice :: [a] -> Int -> Int -> [a]

slice x i k = drop i (take (k+1) x)
```



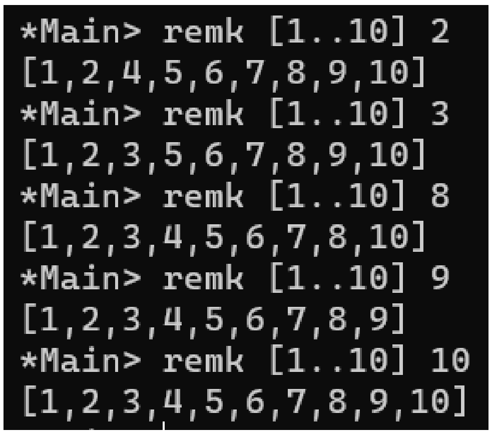
```
*Main> slice [1..10] 2 4
[3,4,5]
*Main> slice [1..10] 0 4
[1,2,3,4,5]
*Main> slice [1..10] 8 10
[9,10]
```

10 - Define a function to remove the kth indexed element from a list.

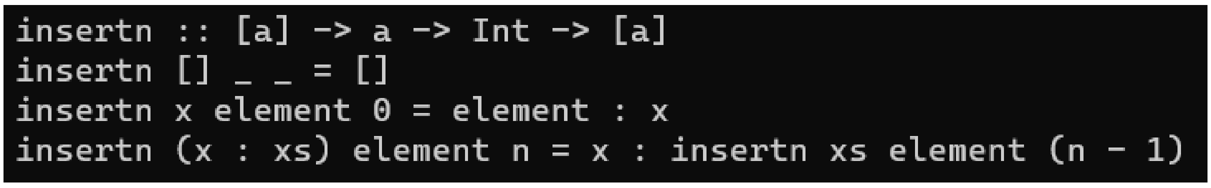
- Consider a list [1..10], and value of n be 2, then resultant list will be [1,2,4,5,6,7,8,9,10].

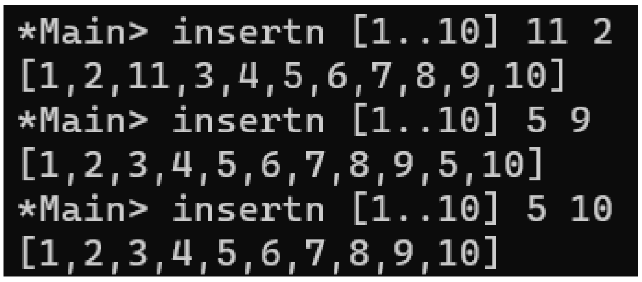


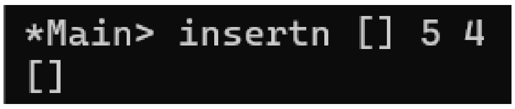
```
remk :: [a] -> Int -> [a]
remk [] _ = []
remk (x:xs) 0 = xs
remk (x:xs) n = x : remk (xs) (n-1)
```


```
*Main> remk [1..10] 2
[1,2,4,5,6,7,8,9,10]
*Main> remk [1..10] 3
[1,2,3,5,6,7,8,9,10]
*Main> remk [1..10] 8
[1,2,3,4,5,6,7,8,10]
*Main> remk [1..10] 9
[1,2,3,4,5,6,7,8,9]
*Main> remk [1..10] 10
[1,2,3,4,5,6,7,8,9,10]
```

11 - Define a function that will insert an element n at a particular index i of a list xs. Let xs=[1..10], i=2, n=11, then the output will be [1,2,11,3,4,5,6,7,8,9,10].


```
insertn :: [a] -> a -> Int -> [a]
insertn [] _ _ = []
insertn x element 0 = element : x
insertn (x : xs) element n = x : insertn xs element (n - 1)
```

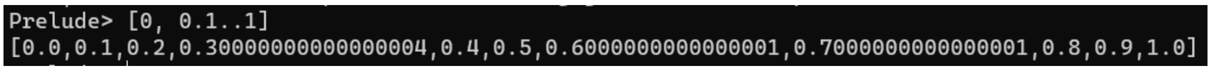

```
*Main> insertn [1..10] 11 2
[1,2,11,3,4,5,6,7,8,9,10]
*Main> insertn [1..10] 5 9
[1,2,3,4,5,6,7,8,9,5,10]
*Main> insertn [1..10] 5 10
[1,2,3,4,5,6,7,8,9,10]
```


```
*Main> insertn [] 5 4
[]
```

List Operations

1 - What value has the expression [0 , 0.1 . . 1] ?

- Check your answer in Hugs and explain any discrepancy there might be between the two.



```
Prelude> [0, 0.1..1]
[0.0,0.1,0.2,0.30000000000000004,0.4,0.5,0.6000000000000001,0.7000000000000001,0.8,0.9,1.0]
```

Expected

- [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

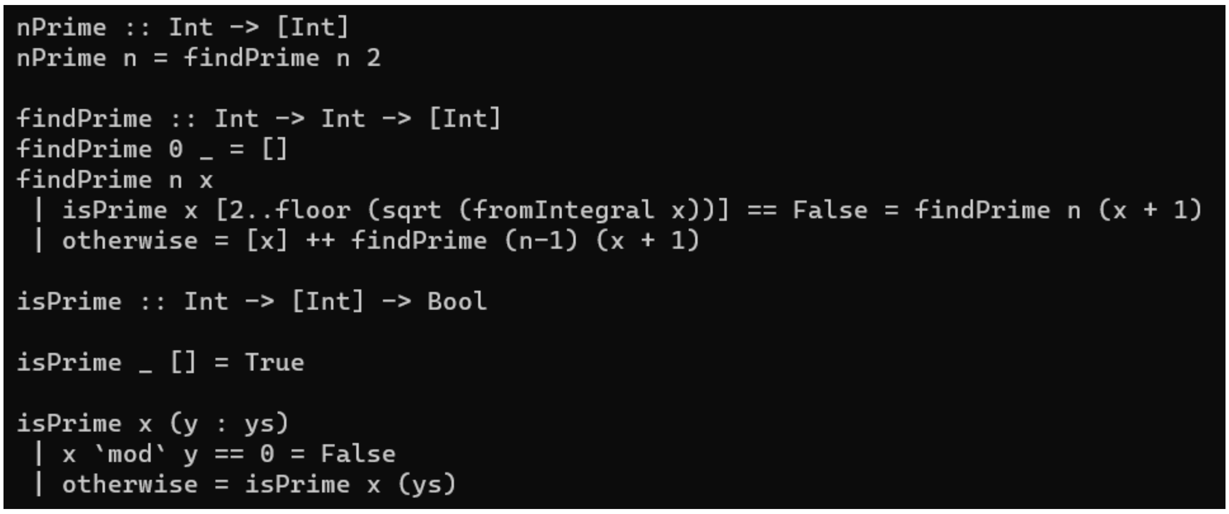
Got

- [0.0,0.1,0.2,0.30000000000000004,0.4000000000000001,0.5000000000000001,0.6000000000000001,0.7000000000000001,0.8,0.9,1.0]

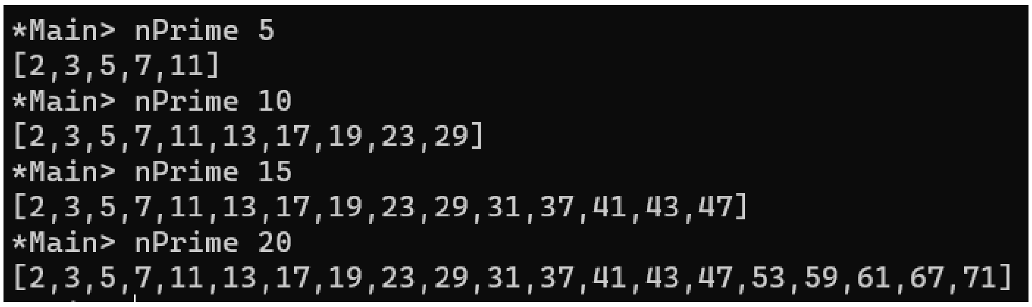
Reason

- Float values are not accurate.
- Floating-point numbers cannot precisely represent all real numbers, and those floating-point operations cannot precisely represent true arithmetic operations.

2 - Define a function that takes an integer number n and returns the list of the first n prime numbers.

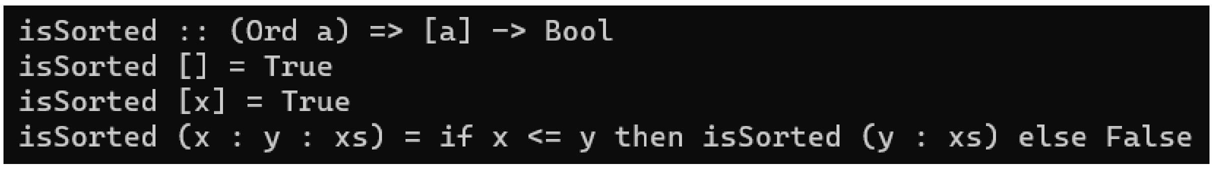


```
nPrime :: Int -> [Int]
nPrime n = findPrime n 2

findPrime :: Int -> Int -> [Int]
findPrime 0 _ = []
findPrime n x
 | isPrime x [2..floor (sqrt (fromIntegral x))] == False = findPrime n (x + 1)
 | otherwise = [x] ++ findPrime (n-1) (x + 1)

isPrime :: Int -> [Int] -> Bool

isPrime _ [] = True

isPrime x (y : ys)
 | x `mod` y == 0 = False
 | otherwise = isPrime x (ys)
```

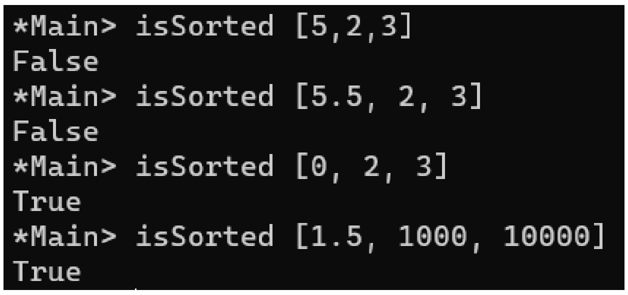


```
*Main> nPrime 5
[2,3,5,7,11]
*Main> nPrime 10
[2,3,5,7,11,13,17,19,23,29]
*Main> nPrime 15
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
*Main> nPrime 20
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

3 - Define a predicate to verifies whether a list is sorted in ascending order.

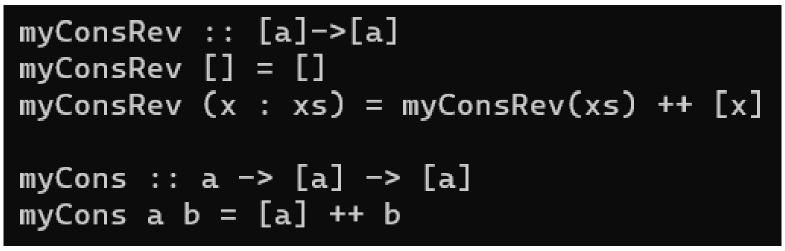


```
isSorted :: (Ord a) => [a] -> Bool
isSorted [] = True
isSorted [x] = True
isSorted (x : y : xs) = if x <= y then isSorted (y : xs) else False
```

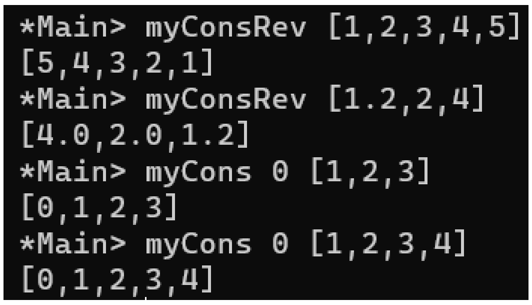


```
*Main> isSorted [5,2,3]
False
*Main> isSorted [5.5, 2, 3]
False
*Main> isSorted [0, 2, 3]
True
*Main> isSorted [1.5, 1000, 10000]
True
```

4 - Define a function myCons that behaves as : but is defined in terms of ++.

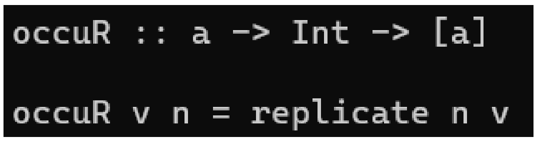


```
myConsRev :: [a]->[a]
myConsRev [] = []
myConsRev (x : xs) = myConsRev(xs) ++ [x]

myCons :: a -> [a] -> [a]
myCons a b = [a] ++ b
```

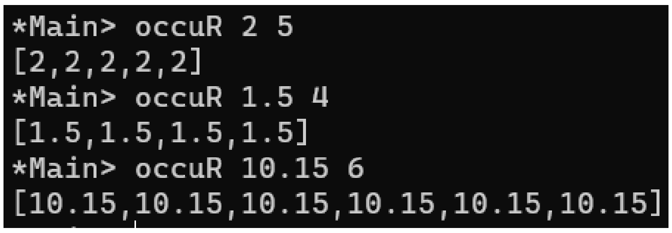


```
*Main> myConsRev [1,2,3,4,5]
[5,4,3,2,1]
*Main> myConsRev [1.2,2,4]
[4.0,2.0,1.2]
*Main> myCons 0 [1,2,3]
[0,1,2,3]
*Main> myCons 0 [1,2,3,4]
[0,1,2,3,4]
```

5 - Define a function that takes an integer number n and a value v and creates a list containing n occurrences of v.

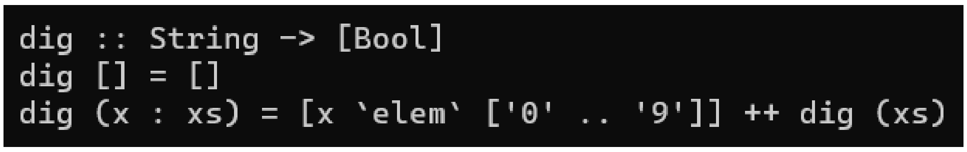


```
occuR :: a -> Int -> [a]

occuR v n = replicate n v
```

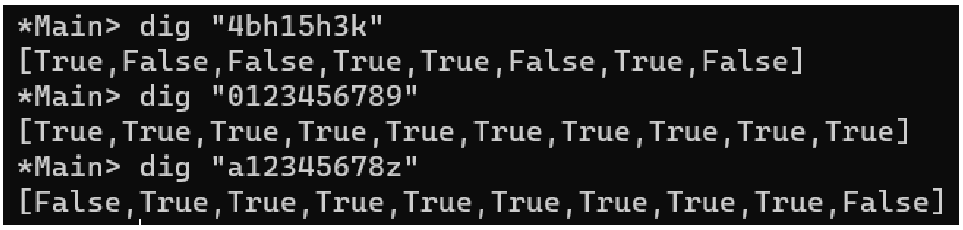


```
*Main> occuR 2 5
[2,2,2,2,2]
*Main> occuR 1.5 4
[1.5,1.5,1.5,1.5]
*Main> occuR 10.15 6
[10.15,10.15,10.15,10.15,10.15,10.15]
```

6 - To find all the digits in a string where the prelude function is True on those characters which are digits: ' 0 ' , ' 1 ' up to ' 9 '

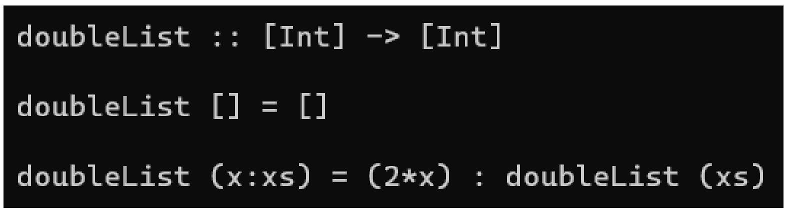


```haskell
dig :: String -> [Bool]
dig [] = []
dig (x : xs) = [x `elem` ['0' .. '9']] ++ dig (xs)
```

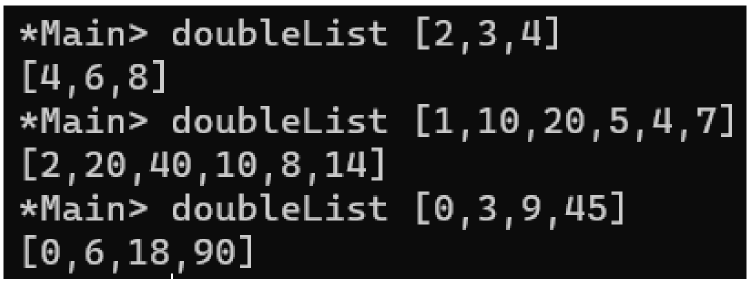


```
*Main> dig "4bh15h3k"
[True,False,False,True,True,False,True,False]
*Main> dig "0123456789"
[True,True,True,True,True,True,True,True,True,True]
*Main> dig "a12345678z"
[False,True,True,True,True,True,True,True,True,False]
```

7 - Give a definition of a function which doubles all the elements of a list of integers.

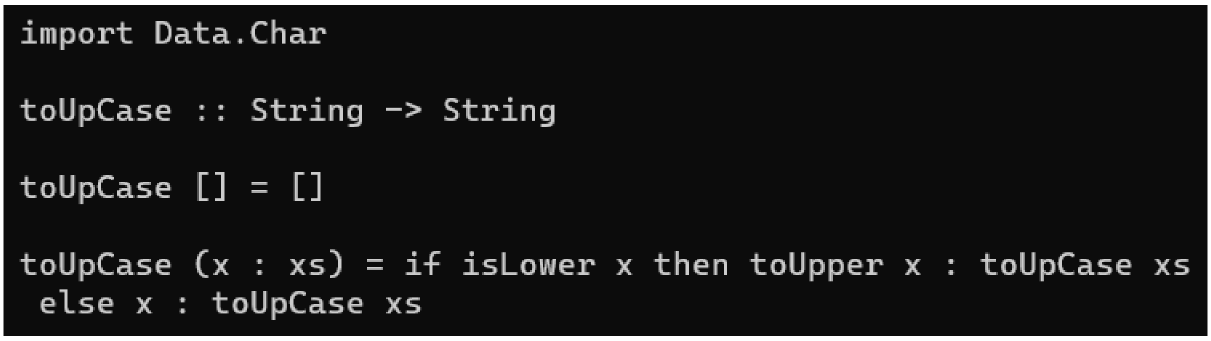


```haskell
doubleList :: [Int] -> [Int]

doubleList [] = []

doubleList (x:xs) = (2*x) : doubleList (xs)
```

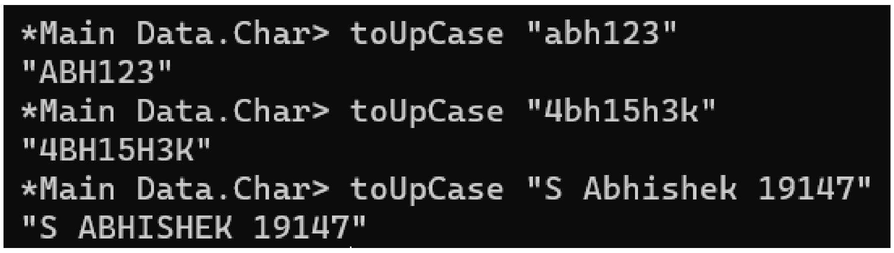


```
*Main> doubleList [2,3,4]
[4,6,8]
*Main> doubleList [1,10,20,5,4,7]
[2,20,40,10,8,14]
*Main> doubleList [0,3,9,45]
[0,6,18,90]
```

8 - Give a definition of a function which converts all small letters in a String into capitals, leaving the other characters unchanged.

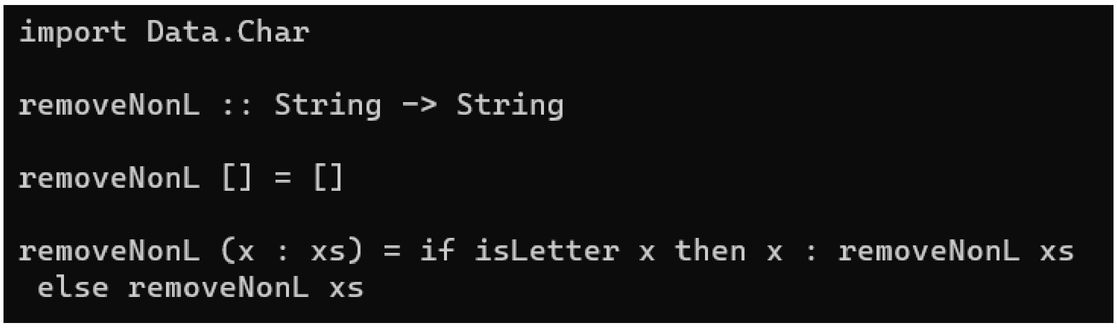


```
import Data.Char

toUpCase :: String -> String

toUpCase [] = []

toUpCase (x : xs) = if isLower x then toUpper x : toUpCase xs
 else x : toUpCase xs
```

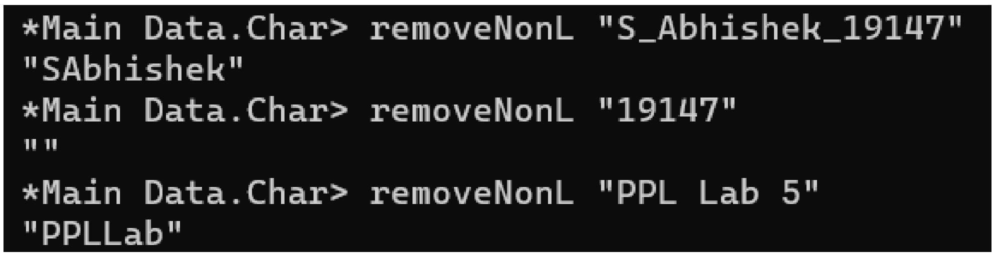


```
*Main Data.Char> toUpCase "abh123"
"ABH123"
*Main Data.Char> toUpCase "4bh15h3k"
"4BH15H3K"
*Main Data.Char> toUpCase "S Abhishek 19147"
"S ABHISHEK 19147"
```

9 - Remove all non-letters from the list

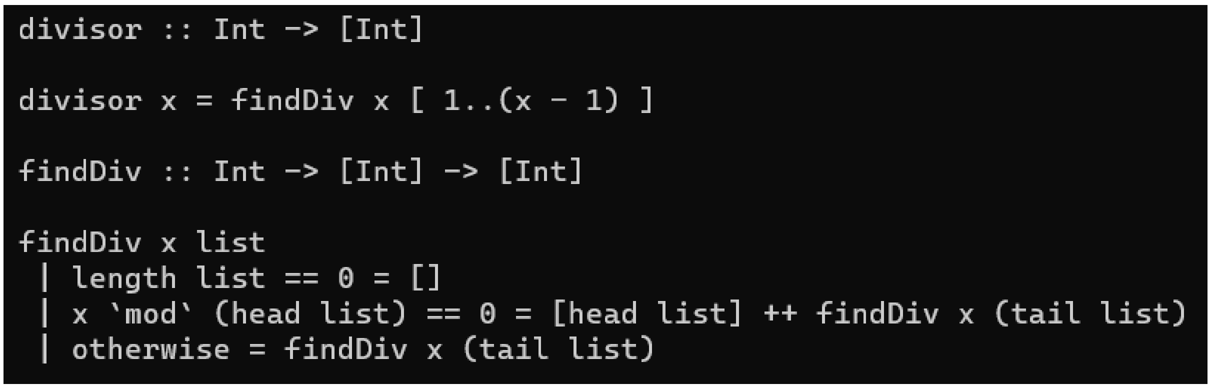


```
import Data.Char

removeNonL :: String -> String

removeNonL [] = []

removeNonL (x : xs) = if isLetter x then x : removeNonL xs
 else removeNonL xs
```

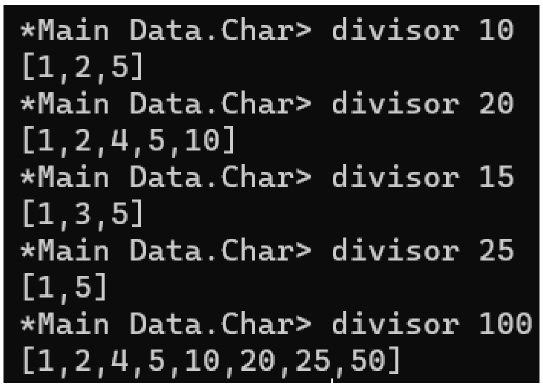


```
*Main Data.Char> removeNonL "S_Abhishek_19147"
"SAbhishek"
*Main Data.Char> removeNonL "19147"
""
*Main Data.Char> removeNonL "PPL Lab 5"
"PPLLab"
```

10 - Define the function which returns the list of divisors of a positive integer (and the empty list for other inputs).

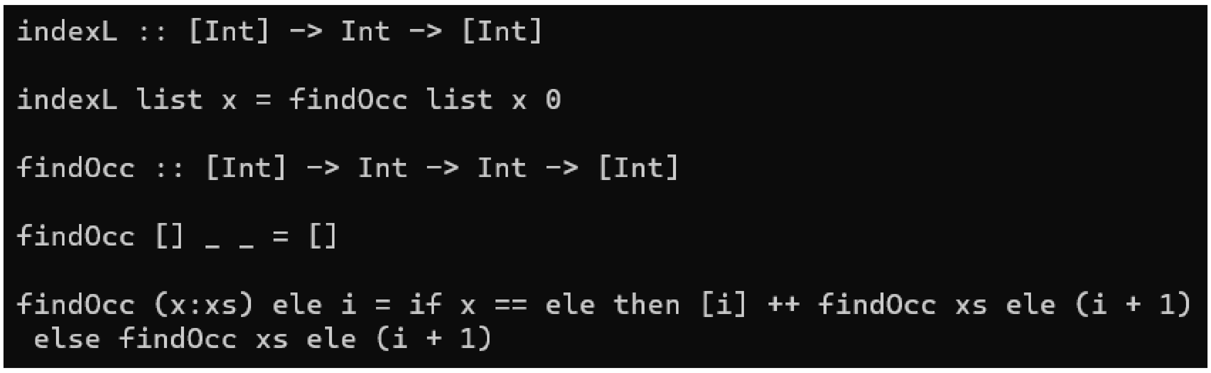


```
divisor :: Int -> [Int]

divisor x = findDiv x [ 1..(x - 1) ]

findDiv :: Int -> [Int] -> [Int]

findDiv x list
 | length list == 0 = []
 | x `mod` (head list) == 0 = [head list] ++ findDiv x (tail list)
 | otherwise = findDiv x (tail list)
```

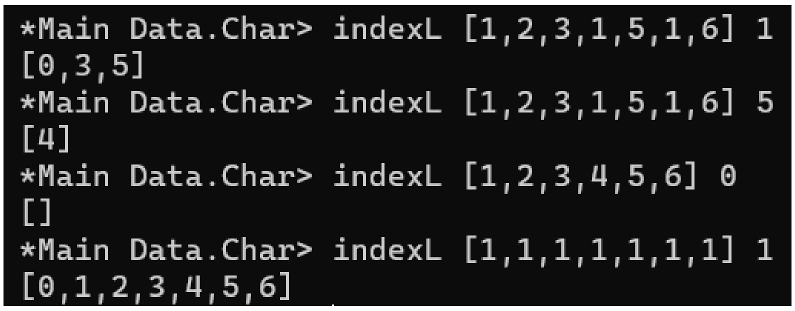


```
*Main Data.Char> divisor 10
[1,2,5]
*Main Data.Char> divisor 20
[1,2,4,5,10]
*Main Data.Char> divisor 15
[1,3,5]
*Main Data.Char> divisor 25
[1,5]
*Main Data.Char> divisor 100
[1,2,4,5,10,20,25,50]
```

11 - Define the function which picks out all occurrences of an integer n in a list.



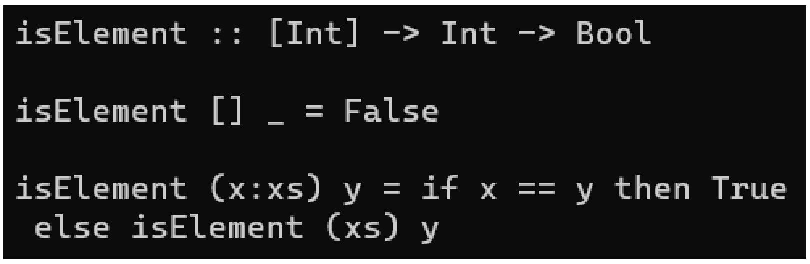
```
indexL :: [Int] -> Int -> [Int]

indexL list x = findOcc list x 0

findOcc :: [Int] -> Int -> Int -> [Int]

findOcc [] _ _ = []

findOcc (x:xs) ele i = if x == ele then [i] ++ findOcc xs ele (i + 1)
 else findOcc xs ele (i + 1)
```



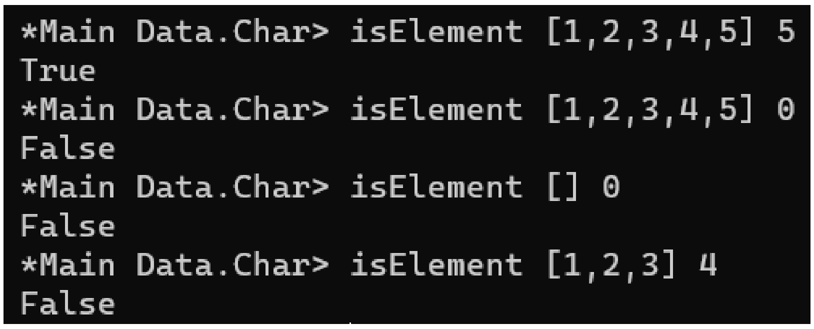
```
*Main Data.Char> indexL [1,2,3,1,5,1,6] 1
[0,3,5]
*Main Data.Char> indexL [1,2,3,1,5,1,6] 5
[4]
*Main Data.Char> indexL [1,2,3,4,5,6] 0
[]
*Main Data.Char> indexL [1,1,1,1,1,1,1] 1
[0,1,2,3,4,5,6]
```

12 - Using matches or otherwise, define a function which is True

if the Int is an element of the list, and False otherwise.

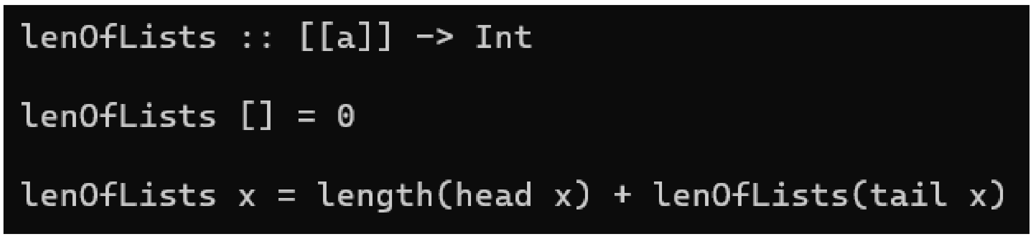


```
isElement :: [Int] -> Int -> Bool

isElement [] _ = False

isElement (x:xs) y = if x == y then True
 else isElement (xs) y
```

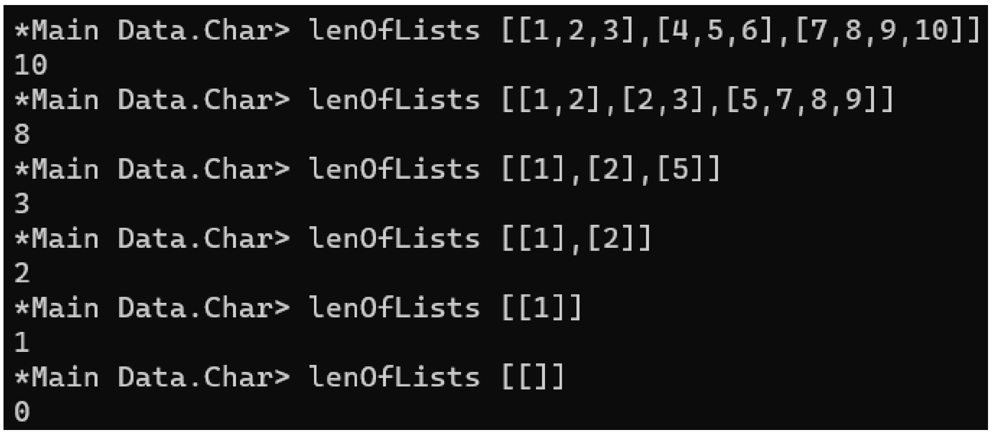


```
*Main Data.Char> isElement [1,2,3,4,5] 5
True
*Main Data.Char> isElement [1,2,3,4,5] 0
False
*Main Data.Char> isElement [] 0
False
*Main Data.Char> isElement [1,2,3] 4
False
```

13 - Given a list of lists, sum the lengths of inner lists



```
lenOfLists :: [[a]] -> Int

lenOfLists [] = 0

lenOfLists x = length(head x) + lenOfLists(tail x)
```



```
*Main Data.Char> lenOfLists [[1,2,3],[4,5,6],[7,8,9,10]]
10
*Main Data.Char> lenOfLists [[1,2],[2,3],[5,7,8,9]]
8
*Main Data.Char> lenOfLists [[1],[2],[5]]
3
*Main Data.Char> lenOfLists [[1],[2]]
2
*Main Data.Char> lenOfLists [[1]]
1
*Main Data.Char> lenOfLists [[]]
0
```

Thankyou!!