```haskell
{-

1- Write a program that  takes as input two numbers and return the       greater

-}

-- first solution
theGreaterBetween1 x y
    | x < y = y
    | otherwise = x

-- second solution
theGreaterBetween2 x y =
   if x < y
   then y
   else x

-- third solution
theGreaterBetween3 x y = max x y
```

```haskell
{-

2- Write a program that takes an integer number and return the string "EVEN" if the number is
even or "ODD" if the number is odd

-}

-- first solution
evenOrOdd1 n =
   if (mod n  2) == 0
   then "EVEN"
   else "ODD"

-- second solution
evenOrOdd2 n
    | (mod n 2) == 0 = "EVEN"
    | otherwise = "ODD"
```

```
{-

3- Write à program that returns the team of a student given its matricula number.
There are three possible teams: RED, GREEN and BLUE. The assignment takes place
with the following criterion: the student with matricula 1 goes to RED team,
the one with matricula 2 in the GREEN, the one with matricula number 3 in the BLUE,
the one with matricula number 4 in the RED, that one with 5 in GREEN etc.
 (Note: you can use case construction)

-}

teamDetermination mat =
  case (mod mat 3) of
      0 -> "BLUE"
      1 -> "RED"
      2 -> "GREEN"
```

```
{-

4- Write a program that takes as input an integer corresponding to the score of a student
and returns "Insufficient" if it is less than 18, "Just enough" (if the score is 18),
"Low" (if the score is between 19-20), "Medium" (if the score is between 21-23),
"Good" (if the score is between 24-26), "High" (if the score is between 27-29),
"Maximum" (if the score is 30) "Impossible" (in others cases) (Note: you can use guard)

-}

scoreApreciation score
      | score < 18 = "Insufficient"
      | score == 18 = "Just enough"
      | score >= 19 && score <= 20 = "Low"
      | score >= 21 && score <= 23 = "Medium"
      | score >= 24 && score <= 26 = "Good"
```

```
    | score >= 27 && score <= 29 =  "High"
    | score == 30 = "Maximum"
    | otherwise = "Impossible"
```

```
{-

5- Write a program that takes three coefficients a, b and c of a second degree equation,
and returns the solutions if these are real; if they are not, it must simply return
"Non-real values".

-}



data SolutionData =
    RealSolution Double Double
  | NonRealValues
  deriving (Eq, Show)

solveEquationSecondDegree a b c =
    let delta = b * b - 4*a*c
    in if delta >= 0
       then RealSolution  ((-b - sqrt delta)/(2*a)) ((-b + sqrt delta)/(2*a))
       else NonRealValues
```

```
{-

6- A secular year (divisible by 100) is a leap year if it is divisible by 400, a non-secular
year is a leap year if it is divisible by 4. For example, the year 1900 was not a leap, 1996 was
a leap, 2000 was, 2002 was not a leap. Write a program that takes a year as input and indicates
whether it is a leap year or not.

-}

isLeapYear year
    | (mod year 100) == 0 =
```

```haskell
        if (mod year 400) == 0
        then True
        else False
    | otherwise =
        if (mod year 4) == 0
        then True
        else False
```

```haskell
{-

7- Write a program that takes as input a list of integers and returns the average of the number
in the list.

-}

average xs = realToFrac (sum xs)/realToFrac (length xs)
```

```haskell
{-

8- Write a program that takes a list of integers as input,
and returns a list consisting first of all the even values
in the order in which they are in the input list and then
all the odd values in the reverse order.
Example: given the values: 8 1 3 2 8 6 5, the
program will return: 8 2 8 6 5 3 1

-}
evenOrdOddReverse :: Integral a => [a] -> [a]
evenOrdOddReverse xs =
    let pair = filter (\x -> (mod x 2) == 0) xs
        impair = filter (\x -> (mod x 2) == 1) xs
    in pair ++ reverse impair
```

```haskell
{-

9- Write a program that takes a list of doubles as input,
and returns a list of 3-value moving averages of these numbers.
The program must check that the number of values in the list
is at least equal to 3. The moving average is an arithmetic average
over only a part of the values (in this case 3),
for example if the sequence of values is given:
2.1, 4.2, 1.3, 6.7, 3.1, 5.5, 2.1, 4.9, 3.0, 5.4, 3.9
the program has to calculate the average of 2.1, 4.2 and 1.3 and record it,
then the average of 4.2, 1.3 and 6.7 and record it,
then 1.3, 6.7 and 3.1 and record it,
etc. up to 3.0, 5.4 and 3.1

-}



-- first solution
mobileThreeAvarage1 :: Fractional a => [a] -> [a]
mobileThreeAvarage1 xs
    | length xs < 3 = error "List too small"
    | otherwise = fst $ foldl f ([], 0) xs
                        where f (yx, ind) x =
                                  if (length xs - ind) >= 3
                                  then let mobileAvg = (x + xs!!(ind + 1) + xs!!(ind + 2))/3
                                       in (yx ++ [mobileAvg], ind + 1)
                                  else (yx, ind + 1)

--second solution
mobileThreeAvarage2 :: Fractional a => [a] -> [a]
mobileThreeAvarage2 [] =  error "List too small"
mobileThreeAvarage2 (_:[]) =  error "List too small"
mobileThreeAvarage2 (_:_:[]) =  error "List too small"
mobileThreeAvarage2 (x:y:z:[]) =  [(x + y + z)/3]
mobileThreeAvarage2 (x:y:z:xs) = ((x + y + z)/3): mobileThreeAvarage2 (y:z:xs)
```

```haskell
import Data.List

{-

10- Write a program that takes a list of integer values as input and identifies the longest sequence
of consecutive equal numbers. If several sequences of the same length are identified, consider only
```

```haskell
the first one identified. The program must indicate the repeated value and the number of repetitions
of that value.
Example:
   Inputs: [19, 3, 15, 15, 7, 9, 9, 9, 9, 12, 3, 3, 3]
   Output: number: 9, occurrences: 4

-}


longuestConsSequence :: (Eq a, Show a) => [a] -> String
longuestConsSequence [] = []
longuestConsSequence xs =
   let gprs = group xs
       res = foldr step [] gprs
               where
                   step :: [a] -> [a] -> [a]
                   step xs ys =
                       if length xs >= length ys
                       then xs
                       else ys
   in case res of
       []-> "Empty"
       (x:_) -> "Output: number: " ++ show x ++ ", occurrences: " ++ show (length res)
```

```haskell
{-

11- Write a program that takes as input a matrix of integers and returns the maximum,
the minimum,the sum and the average.

-}


maxMatrix :: [[Integer]] -> (Integer, Integer, Integer, Double)
maxMatrix [] = error  "empty list"
maxMatrix xs =
   let max = maximum lineraMatrx
       min = foldr step matrx00 xs
               where
                   matrx00 = head $ head xs
                   step :: [Integer] -> Integer -> Integer
                   step ys y =
                       if y <= l1
                       then y
                       else l1
```

```haskell
                            where
                                l1 = minimum ys
        summ = foldr  step 0 xs
                where
                    step :: [Integer] -> Integer -> Integer
                    step line acc = sum line + acc
        aver = fromIntegral (sum lineraMatrx) / fromIntegral (length lineraMatrx)
    in (max, min, summ, aver)
    where lineraMatrx = concat xs
```

```haskell
{-

12- Write a program that takes a matrix of integer values as input
and returns how many values are even and how many are odd.

-}


helper :: [Integer] -> (Int, Int)
helper xs = foldl f (0, 0) xs
                    where f (evenAcc, oddAcc) x =
                                if (mod x 2) == 0
                                    then (evenAcc + 1, oddAcc)
                                    else (evenAcc, oddAcc + 1)


-- first solution
countEvenAndOdd1 :: [[Integer]] -> (Int, Int)
countEvenAndOdd1 mx = helper $ concat mx


-- second solution
countEvenAndOdd2 :: [[Integer]] -> (Int, Int)
countEvenAndOdd2 mx = foldl f (0, 0) (map helper mx)
                            where f (evenAcc, oddAcc) (e, o) = (evenAcc + e, oddAcc + o)
```

```haskell
{-
```

```
15- Write a program that takes as input two matrices and returns the product of the two.
The program must verify that the matrices are valides and if they can be multiplied.
Two matrices can be multiplied if the number of columns of the first matrix is equal to
the number of lines of the second.

-}

import Data.List

isMatrix :: [[a]] -> Bool
isMatrix [[]] = True
isMatrix matrix = foldl f True matrix
                            where --f :: Bool -> [a] -> Bool
                                  f acc line = acc && (length line == length (matrix !! 0))

canBeMultiply :: (Num a) => [[a]] -> [[a]] -> Bool
canBeMultiply a b
             | isMatrix a == False || isMatrix b == False = False
             | length (a !! 0) /= length b = False
             | otherwise = True


matrixProduct :: (Num a) => [[a]] -> [[a]] -> [[a]]
matrixProduct a b
             | canBeMultiply a b == False = error "The matrixs can not be multiplied"
             | otherwise = foldl f [] a
                                 where f ac alineOfa  = ac ++ [row]
                                          where row = foldl ff [] (transpose b)
                                                 where ff acc alineOfbTranspose  =  acc ++ [res]

where res = sum $ zipWith (*) alineOfa alineOfbTranspose
```

```
{-

16- Write a function that divides two integral numbers using recursive subtraction.
The type should be (Integral a) => a -> a -> a.
Redo this exercise using  the type (Integral a) => a -> a ->(a, a)
where (a, a) represents the quotient and the rest of the division.

-}

-- first solution
```

```haskell
divideWithRecursiveSubtraction1 :: (Integral a) => a->a->a
divideWithRecursiveSubtraction1 0 0 = error "Unknown Result"
divideWithRecursiveSubtraction1 _ 0 = error "Infinity quotient"
divideWithRecursiveSubtraction1 dividende divisor =
    if dividende < divisor
    then 0
    else 1 + divideWithRecursiveSubtraction1 (dividende - divisor) divisor


-- second solution
divideWithRecursiveSubtraction2 :: (Integral a) => a -> a -> (a, a)
divideWithRecursiveSubtraction2 0 0 = error "Unknown Result"
divideWithRecursiveSubtraction2 _ 0 = error "Infinity quotient"
divideWithRecursiveSubtraction2 dividende divisor =
    if dividende < divisor
    then (0, dividende)
    else let res = divideWithRecursiveSubtraction2 (dividende - divisor) divisor
         in (1 + fst res, snd res)
```

```haskell
{-

17- Write a function that recursively sums all numbers from 1 to n,
n being the argument. So that if n was 5, you'd add 1 + 2 + 3 + 4 + 5 to get 15.  The type
should be (Eq a, Num a) => a -> a.

-}

sumInterval :: (Eq a, Num a) => a -> a
sumInterval 0 = 0
sumInterval n = n + sumInterval (n - 1)
```

```haskell
{-

18- Write a function that multiplies two integral numbers using recursive summation.
The type should be (Integral a) => a -> a -> a.

-}
```

```haskell
multiplyRecurAdd :: (Integral a) => a -> a -> a
multiplyRecurAdd 0 _ = 0
multiplyRecurAdd _ 0 = 0
multiplyRecurAdd  x y = x + multiplyRecurAdd x (y - 1)
```

```haskell
{-

19- Write a program that takes two strings as input and returns the longest.
The first if they are of equal length.

-}

longestString :: String -> String -> String
longestString xs ys =
    if (length xs) >= (length ys)
    then xs
    else ys
```

```haskell
{-

20- Write a program that takes two strings as input and returns the greater one.

-}

greatestString :: String -> String -> String
greatestString xs ys = if  xs >= ys then xs else ys
```

```haskell
{-

21- Write a program that takes as input a string
and returns the number of characters it is composed of
```

```haskell
-}

-- first solution
numChar1 :: String -> Int
numChar1 = length

-- second solution
numChar2 :: String -> Int
numChar2 "" = 0
numChar2 (_:xs) = 1 + numChar2 xs
```

```haskell
import Data.Char (toUpper)
{-

22- Write a program that takes a string as input, and returns the same string converted to all
uppercase.

-}

toUpperCase :: [Char] -> [Char]
toUpperCase  = map toUpper

toUpperCaseRec :: [Char] -> [Char]
toUpperCaseRec [] = []
toUpperCaseRec (x:xs) = toUpper x : toUpperCaseRec xs
```

```haskell
{-

23- Write a program that takes a string as input and checks
if it contains at least one 'A' among the first 10 characters.

-}

checkAInFirstTen :: String -> Bool
checkAInFirstTen str =
    if (length str) <= 10
    then 'A' `elem` str
```

```
    else 'A' `elem` take 10 str
```

```
{-

24- Write a program that takes a string as input and counts how many digits it contains.
Example "Hello2022! C6? " must give 5.

-}

import Data.Char

countDigitsRec :: [Char] -> Integer
countDigitsRec [] = 0
countDigitsRec (x:xs) =
      if isDigit x
      then 1 + countDigitsRec xs
      else countDigitsRec xs


countDigitsFold :: [Char] -> Integer
countDigitsFold = foldr (\ x acc -> if isDigit x then 1 + acc else acc) 0
```

```
{-

25- Write a program that takes a string as its input and counts
how many uppercase letters, lowercase letters, digits and other characters it consists of
Example
      "Hello2022! C6? " must give:
      uppercase: 2, lowercase: 4, digits: 5, others: 4.

-}

import Data.Char

countThem :: [Char] -> [Char]
countThem xs =
      let  (uc,lc,dc,oc)  =
            foldr step (0,0,0,0) xs
                  where
```

```haskell
                 step :: Char -> (Int, Int, Int, Int) -> (Int, Int, Int, Int)
                 step c (upp, low, dig, oth)
                       | isUpper c = (upp + 1, low, dig, oth)
                       | isLower c = (upp, low + 1, dig, oth)
                       | isDigit c = (upp, low, dig + 1, oth)
                       | otherwise = (upp, low, dig, oth + 1)
      in "uppercase: " ++ show uc ++
      ", lowercase: " ++ show lc ++
      ", digits: " ++ show dc ++
      ", others: " ++ show oc
```

```haskell
{-

26- Write a program that takes two strings of different lengths as input
and indicates whether the shortest is contained only once in the longest.

-}

import Data.List

removeAllBeforeOccurence :: String -> String -> String
removeAllBeforeOccurence smallone bigone =
    let occ = dropWhile (\x -> (smallone!!0 /= x)) bigone
        sm = take (length smallone) occ
    in if sm == smallone
       then drop (length smallone) occ
       else removeAllBeforeOccurence smallone occ

myContains :: String -> String -> Bool
myContains "" _  =  False
myContains smallone bigone  =
    if not (isInfixOf smallone bigone)
    then False
    else
      let occ = removeAllBeforeOccurence smallone bigone
      in if   isInfixOf smallone occ
         then False
         else True

contains :: String -> String -> Bool
contains first second =
    if (length first) > (length second)
    then myContains second first
```

```haskell
    else myContains first second
```

```haskell
{-

27- Write a function that tells you whether or not a given String (or list) is a palindrome.
Here you'll want to use a function called reverse,
a predefined function that does what it sounds like.
reverse :: [a] -> [a]
reverse "blah" is "halb"
      Example:
      radar, rotor, madam, kayak, anilina, otto, elle

-}

-- first solution
isPalindrome :: Ord a => [a] -> Bool
isPalindrome xs = xs == reverse  xs

-- second solution
isPalindromeRec :: Ord a => [a] -> Bool
isPalindromeRec [] = True
isPalindromeRec (x:xs) =
      x == last xs && isPalindrome (init xs)
```

```haskell
{-

32- Given a number, determine whether or not it is valid per the Luhn formula. The Luhn
algorithm is a simple checksum formula used to validate a variety of identification numbers,
such as credit card numbers and Canadian Social Insurance Numbers.
The task is to check if a given string is valid.
Validating a Number
Strings of length 1 or less are not valid. Spaces are allowed in the input, but they should be
stripped before checking. All other non-digit characters are disallowed.
Example 1: valid credit card number
4539 3195 0343 6467
The first step of the Luhn algorithm is to double every second digit, starting from the right.
We will be doubling
4_3_ 3_9_ 0_4_ 6_6_
If doubling the number results in a number greater than 9 then subtract 9 from the product. The
```

```haskell
results of our doubling:
8569 6195 0383 3437
Then sum all of the digits:
8+5+6+9+6+1+9+5+0+3+8+3+3+4+3+7 = 80
If the sum is evenly divisible by 10, then the number is valid. This number is valid!
Example 2: invalid credit card number
8273 1232 7352 0569
Double the second digits, starting from the right
7253 2262 5312 0539
Sum the digits
7+2+5+3+2+2+6+2+5+3+1+2+0+5+3+9 = 57
57 is not evenly divisible by 10, so this number is not valid.

-}

import Data.Char

isValidateCCNumber :: [Char] -> Bool
isValidateCCNumber [] = error "invalid number"
isValidateCCNumber [x] = error "invalid number"
isValidateCCNumber xs =

   let cleanxs = filter isDigit xs
       dgs = map digitToInt cleanxs
       dxs = doubleSndDigits dgs
       sm = sum dxs
   in sm `mod` 10 == 0
   where
       doubleSndDigits :: [Int] -> [Int]
       doubleSndDigits xs =
           fst $
               foldr step ([], 0) xs
                   where
                       step x (ys, ind) =
                           if even ind
                           then (x:ys, ind + 1)
                           else
                               let xTime2 = 2*x
                               in if xTime2 > 9
                                  then ((xTime2 - 9):ys, ind + 1)
                                  else (xTime2 :ys, ind + 1)
```

```haskell
{-

34- Implement run-length encoding and decoding.
Run-length encoding (RLE) is a simple form of data compression, where runs (consecutive data elements) are replaced by just one data value and count.
   For example we can represent the original 53 characters with only 13.
   "WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW"  ->  "12WB12W3B24WB"
   RLE allows the original data to be perfectly reconstructed from the compressed data, which makes it a lossless data compression.
   "AABCCCDEEEE"  ->  "2AB3CD4E"  ->  "AABCCCDEEEE"
   For simplicity, you can assume that the unencoded string will only contain the letters A through Z (either lower or upper case) and whitespace. This way data to be encoded will never contain any numbers and numbers inside data to be decoded always represent the count for the following character.

-}

import Data.List
import Data.Char


runLenght :: [Char] -> [Char]
runLenght [] = []
runLenght xs =
   let grouped = group xs
       grouped' = map countGroups grouped
   in concat grouped'
   where
       countGroups :: [Char] -> [Char]
       countGroups ys =
           if size == 1
           then ys
           else show size ++ [head ys]
           where size = length ys



unRunLenght :: [Char] -> [Char]
unRunLenght [] = []
unRunLenght xs =
   let grouped = groupBy (\x y -> isDigit x && isDigit y) xs
       original = expand grouped
   in original
   where
       expand :: [[Char]] -> [Char]
       expand [] = []
       expand (xs:xss) =
           if all isTrue $ fmap isDigit xs
```

```haskell
        then  replicate n (head $ head xss) ++ expand (tail xss)
        else  xs ++ expand xss
         where
             n = toDigit xs
             toDigit :: [Char] -> Int
             toDigit x = read x :: Int
             isTrue :: Bool -> Bool
             isTrue = (== True)
```