1.15 Interchange the value of two variables without using a 3rd variable.

```
main :: IO ()
main = do
    putStrLn "Enter the first number:"
    input1 <- getLine
    let num1 = read input1 :: Int
    putStrLn "Enter the second number:"
    input2 <- getLine
    let num2 = read input2 :: Int
    putStrLn $ "Before interchanging: num1 =" ++ show num1 ++ ", num2 =" ++ show num2
    let num1 = num1 + num2
    let num2 = num1 - num2
    let num1 = num1 - num2
    putStrLn $ "After interchanging: num1 =" ++ show num1 ++ ", num2 =" ++ show num2
```

1.16 Calculate the displacement S,initial velocity u, acceleration a, time t, $S=ut+1/2at^2$.

```
main :: IO ()
main = do
    putStrLn "Enter initial velocity (u):"
    inputU <- getLine
    let u = read inputU :: Double
    putStrLn "Enter acceleration (a):"
    inputA <- getLine
    let a = read inputA :: Double
    putStrLn "Enter time (t):"
    inputT <- getLine
    let t = read inputT :: Double
    let s = u * t + 0.5 * a * t^2
    putStrLn $ "Displacement (S) =" ++ show s
```

2 Lab 3:-Selection and lists

2.1 Average Marks

```
classAverageAndMessage :: [Int] -> String
classAverageAndMessage marks =
    let n = length marks
        totalMarks = sum marks
        average = fromIntegral totalMarks / fromIntegral n :: Double
        belowAverageCount = length $ filter (\x -> fromIntegral x < average) marks
    in
        if belowAverageCount > 2
            then "More than 2 students have marks below class average."
            else "No issue with class average."
```

```
main :: IO ()
main = do
    putStrLn "Enter marks of students (space-separated):"
    input <- getLine
    let marks = map read (words input) :: [Int]
        averageMessage = classAverageAndMessage marks
```

```
    average = fromIntegral (sum marks) / fromIntegral (length marks) :: Double
putStrLn $ "Class-Average:-" ++ show average
putStrLn averageMessage
```

2.2 Numbers occurring odd number of times

```
import Data.List (group, sort)

frequency :: (Ord a) => [a] -> [(a, Int)]
frequency xs = map (\x -> (head x, length x)) $ group $ sort xs

main :: IO ()
main = do
    input <- getLine
    let elements = map read (words input) :: [Int]
        freq = frequency elements
        oddFreq = filter (\(_, f) -> odd f) freq
        oddNumbers = map fst oddFreq

    putStrLn $ "Numbers-occurring-odd-number-of-times:-" ++ show oddNumbers
```

2.3 Duplicates

```
import Data.List (group, sort)

frequency :: (Ord a) => [a] -> [(a, Int)]
frequency xs = map (\x -> (head x, length x)) $ group $ sort xs

greater :: Int -> Bool
greater f = f > 1

main :: IO ()
main = do
    input <- getLine
    let elements = map read (words input) :: [Int]
        freq = frequency elements
        dupFreq = filter (\(_, f) -> greater f) freq
        dupNumbers = map fst dupFreq

    putStrLn $ "Duplicates:-" ++ show dupNumbers
```

2.4 Product of all elements

```
main :: IO ()
main = do
    input <- getLine
    let elements = map read (words input) :: [Int]
        products = product elements

    putStrLn $ "Product-of-all-elements:-" ++ show products
```

2.5 Odd elements

```
main :: IO ()
main = do
    input <- getLine
    let elements = map read (words input) :: [Int]
        list = filter (\e -> odd e) elements

    putStrLn $ "Odd-elements:-" ++ show list
```

2.6 Sum of even elements

```
main :: IO ()
main = do
    input <- getLine
    let elements = map read (words input) :: [Int]
        list = filter (\e -> even e) elements
        sumEven = sum list

    putStrLn $ "Sum-of-even-elements:-" ++ show sumEven
```

2.7 Minimum list

```
main :: IO ()
main = do
    input <- getLine
    let list = map read (words input) :: [Int]
        min1 = minimum list

    putStrLn $ "Minimum-element-in-the-list:-" ++ show min1
```

2.8 Enter strings seperates by spaces

```
import Data.List

-- Function to calculate the sum of elements at odd indices in a list
sumAtOddIndices :: Num a => [a] -> a
sumAtOddIndices xs = sum [x | (i, x) <- zip [0..] xs, odd i]

main :: IO ()
main = do
    putStrLn "Enter-the-list-elements-separated-by-spaces:"
    input <- getLine
    let numbers = map read $ words input :: [Int]
    let sumOddIndices = sumAtOddIndices numbers
    putStrLn $ "Sum-of-elements-at-odd-indices:-" ++ show sumOddIndices
```

2.9 Enter List by spaces

```
import Data.List

searchNumber :: (Eq a, Show a) => [a] -> a -> String
searchNumber xs target
    | target elem xs = "Number-" ++ show target ++ "-found-in-the-list."
    | otherwise = "Number-" ++ show target ++ "-not-found-in-the-list."
```

```
main :: IO ()
main = do
    putStrLn "Enter the list elements separated by spaces:"
    input <- getLine
    let numbers = map read $ words input :: [Int]

    putStrLn "Enter the number to search:"
    searchInput <- getLine
    let searchValue = read searchInput :: Int

    putStrLn $ searchNumber numbers searchValue
```

2.10 Next Number

```
solution :: [Int] -> Int -> Int
solution [] _ = error "Empty List"
solution (l:ls) n
    | l == n = case ls of
                 [] -> error "No element after the given number"
                 (x:_) -> x
    | otherwise = solution ls n

main :: IO()
main = do
    putStrLn "Enter the list elements:"
    input <- getLine
    let numbers = map read $ words input :: [Int]
    putStrLn "Enter a number that is present in the list:"
    num <- readLn
    let ans = solution numbers num
    putStrLn $ "The element after " ++ show num ++ " is " ++ show ans
```

2.11 Hourly Work Hour

```
calculatePay :: Float -> Float -> Float
calculatePay hourlyRate hoursWorked
    | hoursWorked <= 40 = hourlyRate * hoursWorked
    | otherwise = (hourlyRate * 40) + ((hoursWorked - 40) 1.5 hourlyRate)

main :: IO()
main = do
    putStrLn "Enter hourly pay rate:"
    hourlyRate <- readLn
    putStrLn "Enter number of hours worked for the week:"
    hoursWorked <- readLn
    let weeklyPay = calculatePay hourlyRate hoursWorked
    putStrLn $ "Weekly pay:" ++ show weeklyPay
```

2.12 Enter temperature

```
import Control.Concurrent.STM (check)
checkTemperture :: Float -> String
checkTemperture temperature
    | temperature >= 80 = "Go play golf"
    | temperature >= 70 && temperature <= 79 = "Put on a jacket"
    | otherwise = " it is way too cold"
```

```
main :: IO()
main = do
    putStrLn "Enter-the-temperature-in-degrees-Fahrenheit:-"
    temperature <- getLine
    let temp = read temperature :: Float
    putStrLn $ checkTemperture temp


deskCheck :: IO()
deskCheck = do
    let testTemperature = [95, 72, 50]
    mapM_ (\temp -> putStrLn.$ "Temperature:-"++show temp++"\n"++checkTemperture temp ++ "\n
        ----------") testTemperature
```

2.13 Find type of triangle

```
triangle :: Int -> Int -> Int -> String
triangle a b c
    | a == b && b == c = "Equilateral-triangle"
    | a == b || b == c || a == c = "Isosceles-triangle"
    | otherwise = "Scalene-triangle"

main :: IO ()
main = do
    putStrLn "Enter-the-three-sides-of-the-triangle:"
    a <- readLn :: IO Int
    b <- readLn :: IO Int
    c <- readLn :: IO Int
    putStrLn (triangle a b c)
```

2.14 Driving age

```
checkDrivingAge :: String -> Int -> String
checkDrivingAge name age
    | age >= 16 = name ++ ",-you-are-old-enough-to-drive"

    | otherwise = name ++ ",-you-need-to-wait-" ++ show(16 - age)++"-years-before-you-can-
        drive-legally"
main :: IO()
main = do
    putStrLn "Enter-your-name:-"
    name <- getLine
    putStrLn "Enter-your-Age:-"
    agestr <- getLine
    let age = read agestr :: Int
    putStrLn $ checkDrivingAge name age
```

3 Lab 4:-Write Haskell Programs using functions to perform the following tasks : Numbers And Strings

3.1 Prime or not.

```
isPrime :: Integer -> Bool
isPrime n
  | n <= 1    = False
  | otherwise = not $ any (\x -> n mod x == 0) [2..sqrtN]
  where
    sqrtN = floor $ sqrt $ fromInteger n

main :: IO ()
main = do
  putStrLn "Enter a number:"
  input <- getLine
  let number = read input :: Integer
  if isPrime number
    then putStrLn $ show number ++ " is prime."
    else putStrLn $ show number ++ " is not prime."
```

3.2 Palindrome or not.

```
isPalindrome :: Integer -> Bool
isPalindrome n = reverseDigits n == n

reverseDigits :: Integer -> Integer
reverseDigits = read . reverse . show

main :: IO ()
main = do
  putStrLn "Enter a number:"
  input <- getLine
  let number = read input :: Integer
  if isPalindrome number
    then putStrLn $ show number ++ " is a palindrome."
    else putStrLn $ show number ++ " is not a palindrome."
```

3.3 Armstrong or not.

```
isArmstrong :: Integer -> Bool
isArmstrong n = n == sumOfCubes n

sumOfCubes :: Integer -> Integer
sumOfCubes = sum . map (^3) . digits

digits :: Integer -> [Integer]
digits 0 = [0]
digits n = reverse $ digitList n

digitList :: Integer -> [Integer]
digitList 0 = []
digitList n = n mod 10 : digitList (n div 10)

main :: IO ()
main = do
  putStrLn "Enter a number:"
  input <- getLine
  let number = read input :: Integer
  if isArmstrong number
    then putStrLn $ show number ++ " is an Armstrong number."
    else putStrLn $ show number ++ " is not an Armstrong number."
```

3.4 Sum of all even numbers up to a limit.

```
sumOfEvens :: Integer -> Integer
sumOfEvens limit = sum [x | x <- [2,4..limit]]

main :: IO ()
main = do
  putStrLn "Enter the limit:"
  input <- getLine
  let limit = read input :: Integer
  let result = sumOfEvens limit
  putStrLn $ "The sum of even numbers up to " ++ show limit ++ " is: " ++ show result
```

3.5 Sum of all odd numbers up to a limit.

```
sumOfOdds :: Integer -> Integer
sumOfOdds limit = sum [x | x <- [1,3..limit]]

main :: IO ()
main = do
  putStrLn "Enter the limit:"
  input <- getLine
  let limit = read input :: Integer
  let result = sumOfOdds limit
  putStrLn $ "The sum of odd numbers up to " ++ show limit ++ " is: " ++ show result
```

3.6 Binary to Decimal.

```
binaryToDecimal :: String -> Integer
binaryToDecimal binaryString = sum $ zipWith (\bit power -> read [bit] * 2^power) (reverse
    binaryString) [0..]

main :: IO ()
main = do
  putStrLn "Enter a binary number:"
  input <- getLine
  let binaryString = filter (elem "01") input
  let result = binaryToDecimal binaryString
  putStrLn $ "Decimal equivalent: " ++ show result
```

3.7 Decimal to Binary.

```
decimalToBinary :: Integer -> String
decimalToBinary 0 = "0"
decimalToBinary n = reverse $ go n
  where
    go 0 = ""
    go x = let (q, r) = x divMod 2 in show r ++ go q

main :: IO ()
main = do
  putStrLn "Enter a decimal number:"
  input <- getLine
  let number = read input :: Integer
  let result = decimalToBinary number
  putStrLn $ "Binary equivalent: " ++ result
```

3.8 Print the sequence of number.

```
digitToWord :: Int -> String
digitToWord n
  | n >= 0 && n <= 9 = ["-Zero", "-One", "-Two", "-Three", "-Four", "-Five", "-Six", "-Seven
      ", "-Eight", "-Nine"] !! n
  | otherwise = concat $ map (digitToWord . digitToInt) (show n)
  where
    digitToInt :: Char -> Int
    digitToInt c = read [c]

main :: IO ()
main = do
  putStrLn "Enter-a-number:"
  num <- readLn
  putStrLn $ "The-number-in-words:-" ++ digitToWord num
```

3.9 Generate prime numbers between intervals.

```
isPrime :: Integer -> Bool
isPrime n
  | n <= 1    = False
  | otherwise = not $ any (\x -> n mod x == 0) [2..sqrtN]
  where
    sqrtN = floor $ sqrt $ fromInteger n

generatePrimes :: Integer -> Integer -> [Integer]
generatePrimes start end = filter isPrime [start..end]

main :: IO ()
main = do
  putStrLn "Enter-the-start-of-the-interval:"
  startInput <- getLine
  let start = read startInput :: Integer

  putStrLn "Enter-the-end-of-the-interval:"
  endInput <- getLine
  let end = read endInput :: Integer

  let primes = generatePrimes start end
  putStrLn $ "Prime-numbers-between-" ++ show start ++ "-and-" ++ show end ++ ":-" ++ show
      primes
```

3.10 Find numbers and their sum between 100 and 200 divisible by 9.

```
divisibleBy9 :: Integer -> Bool
divisibleBy9 x = x mod 9 == 0

main :: IO ()
main = do
  let numbers = [1..200]
  let divisibleNumbers = filter divisibleBy9 numbers
  let numberOfDivisibles = length divisibleNumbers
  let sumOfDivisibles = sum divisibleNumbers

  putStrLn $ "Number-of-integers-divisible-by-9-between-1-and-200:-" ++ show
      numberOfDivisibles
putStrLn $ "Sum-of-integers-divisible-by-9-between-1-and-200:-" ++ show sumOfDivisibles
```

3.11 Palindrome or not.

```
isPalindrome :: String -> Bool
isPalindrome str = str == reverse str

main :: IO ()
main = do
  putStrLn "Enter a string:"
  input <- getLine
  let result = isPalindrome input
  if result
    then putStrLn "The string is a palindrome."
    else putStrLn "The string is not a palindrome."
```

3.12 Count numbers of vowels, consonants, numbers in the string.

```
import Data.Char (isAlpha, isDigit, toLower)

countVowelsConsonantsNumbers :: String -> (Int, Int, Int)
countVowelsConsonantsNumbers input =
  (count isVowel, count isConsonant, count isDigit)
  where
    count predicate = length $ filter predicate input
    isVowel c = toLower c elem "aeiou"
    isConsonant c = isAlpha c && not (isVowel c)

main :: IO ()
main = do
  putStrLn "Enter a string:"
  input <- getLine
  let (vowels, consonants, numbers) = countVowelsConsonantsNumbers input
  putStrLn $ "Number of vowels:" ++ show vowels
  putStrLn $ "Number of consonants:" ++ show consonants
  putStrLn $ "Number of numbers:" ++ show numbers
```

3.13 Menu driven program for finding length of string, join two strings, reverse a string, and compare two strings.

```
main :: IO ()
main = do
  putStrLn "Menu:"
  putStrLn "1. Length of string"
  putStrLn "2. Join two strings"
  putStrLn "3. Reverse two strings"
  putStrLn "4. Compare two strings"
  putStrLn "5. Exit"

  putStrLn "Enter your choice (1-5):"
  choice <- getLine

  if choice == "1" then
      lengthOfString
  else
      if choice == "2" then
          joinStrings
      else
          if choice == "3" then
              reverseStrings
```

```
else
    if choice == "4" then
        compareStrings
    else
        if choice == "5" then
            putStrLn "Exiting..."
        else
            putStrLn "Invalid-choice."

lengthOfString :: IO ()
lengthOfString = do
  putStrLn "Enter-the-string:"
  input <- getLine
  putStrLn $ "Length-of-the-string-is:-" ++ show (length input)

joinStrings :: IO ()
joinStrings = do
  putStrLn "Enter-the-first-string:"
  input1 <- getLine
  putStrLn "Enter-the-second-string:"
  input2 <- getLine
  putStrLn $ "Joined-string:-" ++ input1 ++ input2

reverseStrings :: IO ()
reverseStrings = do
  putStrLn "Enter-the-first-string:"
  input1 <- getLine
  putStrLn "Enter-the-second-string:"
  input2 <- getLine
  putStrLn $ "Reversed-strings:-" ++ reverse input1 ++ "-" ++ reverse input2

compareStrings :: IO ()
compareStrings = do
  putStrLn "Enter-the-first-string:"
  input1 <- getLine
  putStrLn "Enter-the-second-string:"
  input2 <- getLine
  if input1 == input2
    then putStrLn "The-strings-are-equal."
    else putStrLn "The-strings-are-not-equal."
```

4 Lab 5:Given a list of strings, find the string which has occurred more no. of times.

```
import Data.List

countOccurrences :: [String] -> [(String, Int)]
countOccurrences [] = []
countOccurrences (x:xs) = (x, count x xs + 1) : countOccurrences (filter (/= x) xs)
  where
    count :: Eq a => a -> [a] -> Int
    count _ [] = 0
    count y (z:zs) = if y == z then 1 + count y zs else count y zs

main :: IO ()
main = do
  let myList = ["helloworld", "hello", "world", "hello"]
      myOccurrences = countOccurrences myList
  maxValue = maximum $ map snd myOccurrences
```

```
print myOccurrences
print maxValue
```

5 LAB 6 : Write Haskell Programs using functions to perform the following tasks : Higher Order Functions

5.1 Average of the list using fold.

```
avg :: [Int] -> Int

avg xs= foldl(\sum x-> sum+x)0 xs
main :: IO()
main = do
    let arr=[1,2,3,4,5]
    let ans= avg(arr) div (length arr)
    print ans
```

5.2 Switch between different currencies.

```
main :: IO ()
main = do
    putStrLn "Menu:"
    putStrLn "1.-Rupee-to-Dollar"
    putStrLn "2.-Rupee-to-pounds"
    putStrLn "3.-Rupee-to-euro"
    putStrLn "4.-Exit"
    putStrLn "Enter-your-choice-(1-4):"
    choice <- getLine
case choice of
    "1" -> do
        putStrLn "Enter-a-number:"
        n <- getLine
        let r= read n :: Float
        let y= r/83
        print(y)
        main
    "2" -> do
        putStrLn "Enter-a-number:"
        n <- getLine
        let r= read n :: Float
        let y= r/105
        print(y)
        main
    "3" -> do
        putStrLn "Enter-a-number:"
        n <- getLine
        let r= read n :: Float
        let y= r/90
        print(y)
        main
"4" -> putStrLn "Exiting..."
_ -> do
    putStrLn "Invalid-choice!-Please-try-again."
    main
```

5.3 Remove Duplicates in the list.

```
sumOfEvens :: Integer -> Integer
sumOfEvens limit = sum [x | x <- [2,4..limit]]

main :: IO ()
main = do
  putStrLn "Enter the limit:"
  input <- getLine
  let limit = read input :: Integer
  let result = sumOfEvens limit
  putStrLn $ "The sum of even numbers up to " ++ show limit ++ " is: " ++ show result
```