A close-up of a school of engineering

Description automatically generated

**LAB PROGRAMS (21-40)**

**ON**

**ITA0402-Statistics with R Programming for Data Visualization**

**SLOT B**

**Submitted by**

**192321047 – KAUSHIK NARAYANAN.V**

**To**

**DR.P. Jesu Jayarin**

**Saveetha School of Engineering**

**SIMATS, Thandalam.**

**21) Factorial Calculation (Handling Edge Cases)**

**Aim:** Calculate the factorial of a given number using a for loop while handling edge cases like negative numbers or zero.  
**Algorithm:**

1. Prompt the user to enter a number.
2. Handle negative input by displaying an error message.
3. Compute factorial using a for loop if the input is non-negative.
4. Return the result.

**Code:**

num <- as.integer(readline(prompt = "Enter a number: "))

if (num < 0) {

print("Error: Factorial is not defined for negative numbers.")

} else if (num == 0) {

print("The factorial of 0 is 1.")

} else {

factorial <- 1

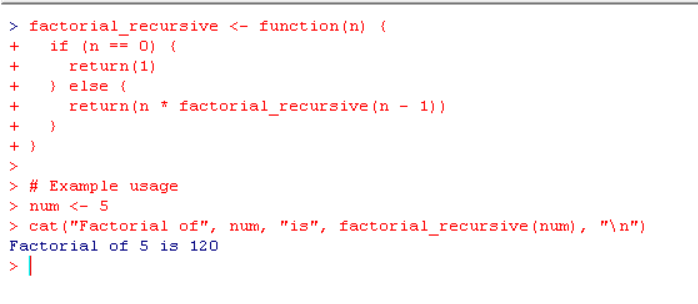
for (i in 1:num) {

factorial <- factorial \* i

}

print(paste("The factorial of", num, "is", factorial))

}



**22. Fibonacci Sequence with Length Calculation**  
**Aim:** Generate a Fibonacci sequence up to a given limit using a while loop.  
**Algorithm:**

1. Initialize the first two numbers of the sequence.
2. Append new numbers while the sequence value is less than the user-specified limit.
3. Print the sequence and its length.

**Code:**

factorial\_iterative <- function(n) {

result <- 1

for (i in 1:n) {

result <- result \* i

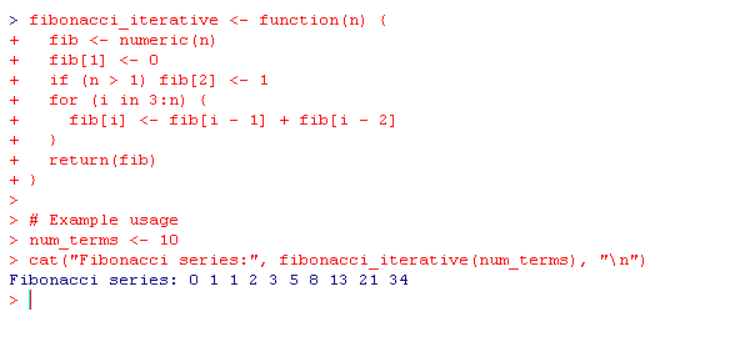
}

return(result)

}

num <- 5

cat("Factorial of", num, "is", factorial\_iterative(num), "\n")

****

**23. Grading System Using Nested If-Else  
Aim:**Assign grades based on the student's score using nested conditions. **Algorithm:**

1. Prompt the user for the score input.
2. Evaluate the score and assign a grade using nested if-else.
3. Display the assigned grade.

**Code:**

# Function to determine grade using nested if-else

get\_grade <- function(score) {

if (score >= 90) {

if (score >= 95) {

return("A+")

} else {

return("A")

}

} else if (score >= 80) {

if (score >= 85) {

return("B+")

} else {

return("B")

}

} else if (score >= 70) {

if (score >= 75) {

return("C+")

} else {

return("C")

}

} else if (score >= 60) {

if (score >= 65) {

return("D+")

} else {

return("D")

}

} else {

return("F")

}

}

score <- as.numeric(readline("Enter your score: "))

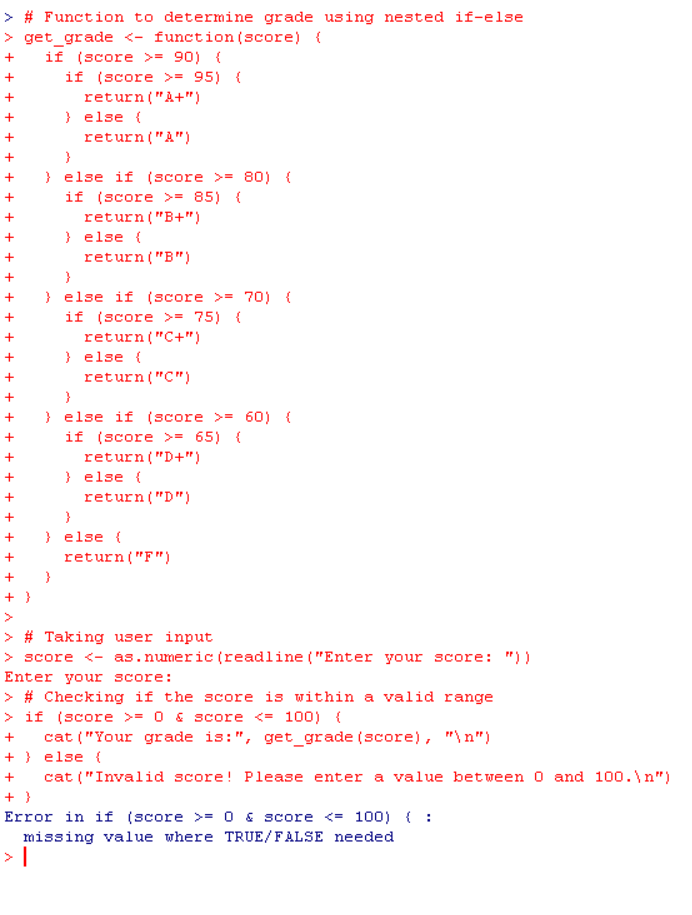
if (score >= 0 & score <= 100) {

cat("Your grade is:", get\_grade(score), "\n")

} else {

cat("Invalid score! Please enter a value between 0 and 100.\n")

}



**24. Mean Calculation Ignoring Non-Numeric Values.  
Aim:**Calculate the mean of numeric vectors, ignoring non-numeric entries. **Algorithm:**

1. Loop over each vector in the list.
2. Filter non-numeric values using is.numeric.
3. Compute and print the mean for valid vectors.

**Code:**

vec\_list <- list(c(1, 2, "A", 4), c(3, 4, 5), c("B", "C", 6))

for (vec in vec\_list) {

numeric\_vec <- as.numeric(vec[!is.na(as.numeric(vec))])

if (length(numeric\_vec) > 0) {

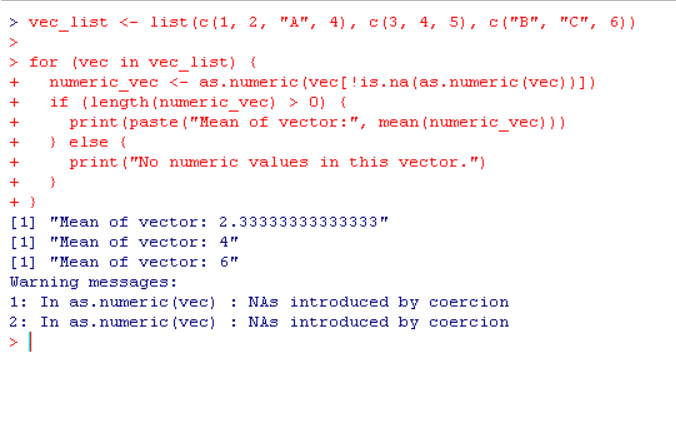
print(paste("Mean of vector:", mean(numeric\_vec)))

} else {

print("No numeric values in this vector.")

}

}



**25. Conditional Row Printing from Data Frame**

**Aim:**Loop over a data frame and print rows satisfying a condition (Age > 30).  
**Algorithm:**

1. Define a sample data frame.
2. Loop over each row using nrow.
3. Print rows meeting the condition.

**Code:**

data <- data.frame(Name = c("Alice", "Bob", "Charlie"), Age = c(25, 35, 40))

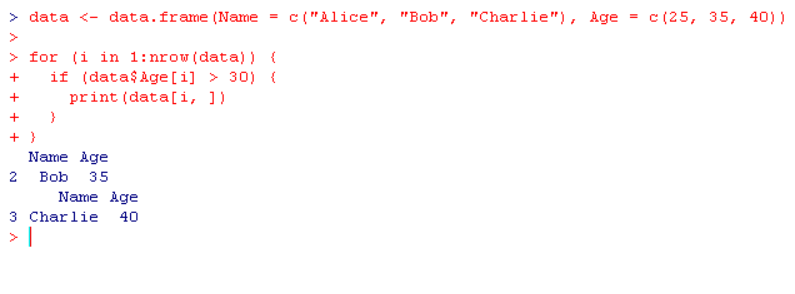
for (i in 1:nrow(data)) {

if (data$Age[i] > 30) {

print(data[i, ])

}

}



**26. Basic Arithmetic Operations**

**Aim:**Perform basic arithmetic operations (+, -, \*, /) on two user-defined numbers. **Algorithm:**

1. Prompt the user to input two numbers.
2. Perform addition, subtraction, multiplication, and division.
3. Display the results.

**Code:**

arithmetic\_operations <- function(a, b, operator) {

if (operator == "+") {

return(a + b)

} else if (operator == "-") {

return(a - b)

} else if (operator == "\*") {

return(a \* b)

} else if (operator == "/") {

if (b == 0) {

return("Error: Division by zero!")

} else {

return(a / b)

}

} else if (operator == "%%") {

return(a %% b)

} else if (operator == "^") {

return(a ^ b)

} else {

return("Invalid operator!")

}

}

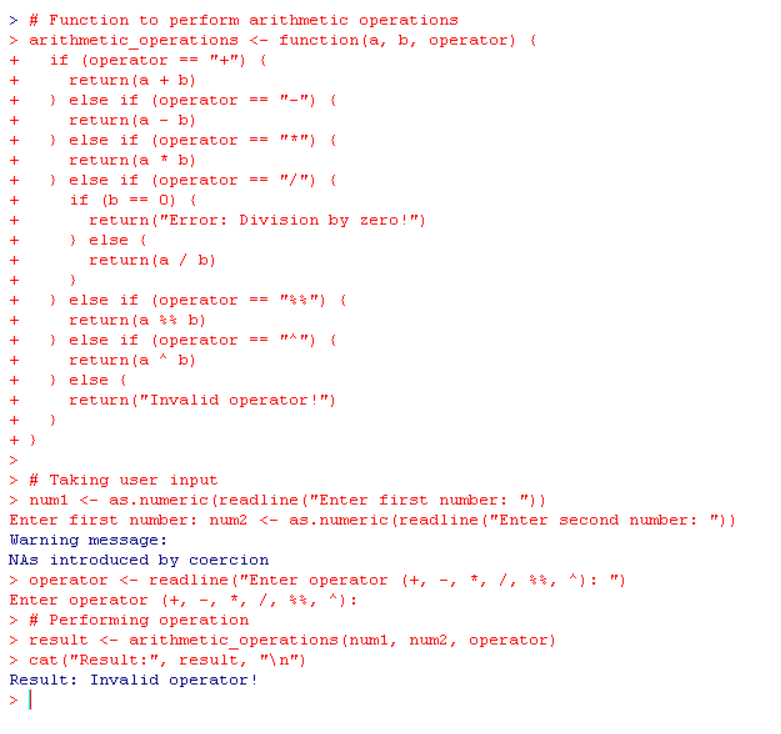
num1 <- as.numeric(readline("Enter first number: "))

num2 <- as.numeric(readline("Enter second number: "))

operator <- readline("Enter operator (+, -, \*, /, %%, ^): ")

result <- arithmetic\_operations(num1, num2, operator)

cat("Result:", result, "\n")



**27. Attendance or Exam Requirement Check**

**Aim:**Identify students who either met attendance requirements or passed exams using the || operator. **Algorithm:**

1. Define a sample data frame with attendance and exam scores.
2. Loop over each student to check whether they meet attendance requirements or pass exams.
3. Print student names satisfying the condition.

**Code:**

students <- data.frame(Name = c("Alice", "Bob", "Charlie"),

Attendance = c(TRUE, FALSE, TRUE),

ExamPassed = c(FALSE, TRUE, FALSE))

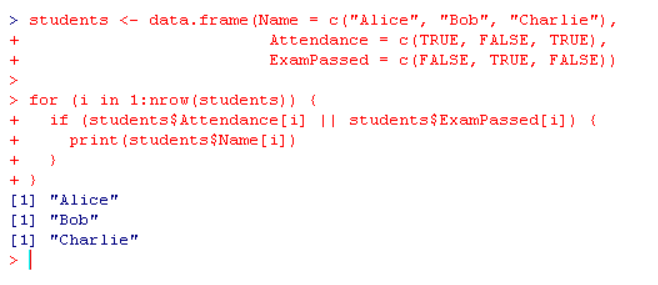
for (i in 1:nrow(students)) {

if (students$Attendance[i] || students$ExamPassed[i]) {

print(students$Name[i])

}

}



**28. Function for Mean, Median, and Mode Calculation**

**Aim:**Create a function that takes a numeric vector and returns the mean, median, and mode. **Algorithm:**

1. Define a function to calculate mean, median, and mode.
2. Return these as a list.
3. Handle non-numeric values if present.

**Code:**

calculate\_stats <- function(vec) {

mode\_calc <- function(x) {

ux <- unique(x)

ux[which.max(tabulate(match(x, ux)))]

}

return(list(

Mean = mean(vec),

Median = median(vec),

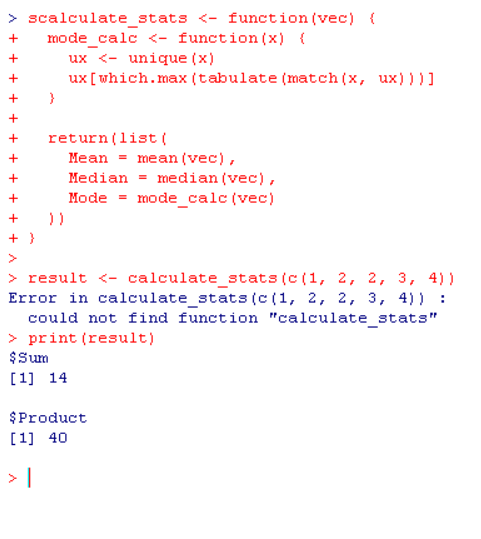
Mode = mode\_calc(vec)

))

}

result <- calculate\_stats(c(1, 2, 2, 3, 4))

print(result)



**29. Recursive Factorial Function**

**Aim:**Calculate the factorial of numbers using a recursive function and apply it to a vector. **Algorithm:**

1. Define a recursive function for factorial calculation.
2. Apply the function to a vector using sapply().

**Code:**

factorial\_recursive <- function(n) {

if (n == 0) return(1)

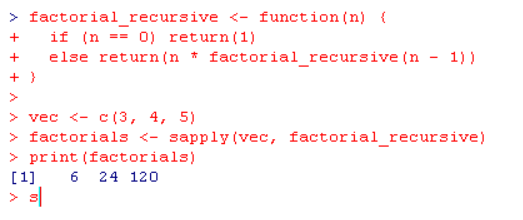
else return(n \* factorial\_recursive(n - 1))

}

vec <- c(3, 4, 5)

factorials <- sapply(vec, factorial\_recursive)

print(factorials)

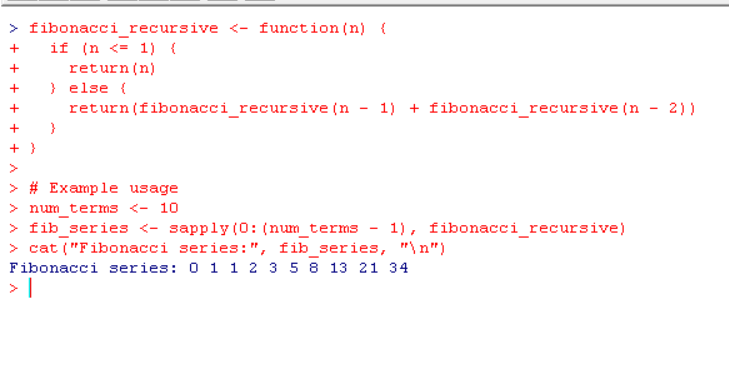


**30. Recursive Fibonacci Function**

**Aim:**Generate the nth Fibonacci number using a recursive function. **Algorithm:**

1. Prompt the user for input n.
2. Define a recursive function to compute the nth Fibonacci number.
3. Display the result.

**Code:**



**31. For Loop with If-Else Statement and Function for Arithmetic Operations**

**Aim:** Iterate over a vector using a for loop, check a condition with an if-else statement, and use a function to perform an arithmetic operation. **Algorithm:**

1. Create a function to perform arithmetic operations with default arguments.
2. Iterate over the vector and check a condition using if-else.
3. Call the function within the loop to perform an operation.
4. Return a complex object (list).

**Code:**

arithmetic\_operation <- function(x = 1, y = 2) {

sum <- x + y

product <- x \* y

result <- list(Sum = sum, Product = product)

return(result)

}

vec <- c(1, 2, 3, 4, 5)

for (num in vec) {

if (num %% 2 == 0) { # Check if the number is even

result <- arithmetic\_operation(num, 10) # Use default value for 10

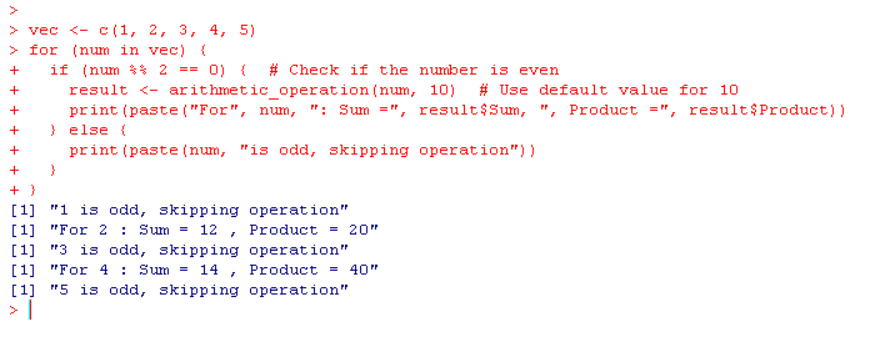
print(paste("For", num, ": Sum =", result$Sum, ", Product =", result$Product))

} else {

print(paste(num, "is odd, skipping operation"))

}

}



**32. Rectangle Area Function with Default Parameters**

**Aim:**Create a function to calculate the area of a rectangle with default values for length and width. **Algorithm:**

1. Define a function to calculate area using length and width.
2. Provide default arguments for length and width.
3. Call the function both with and without arguments.

**Code:**

calculate\_area <- function(length = 5, width = 3) {

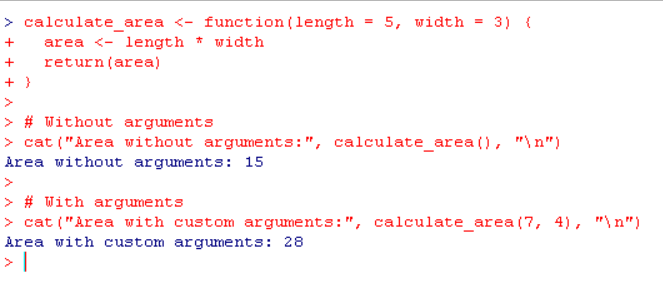
area <- length \* width

return(area)

}

cat("Area without arguments:", calculate\_area(), "\n")

cat("Area with custom arguments:", calculate\_area(7, 4), "\n")



**33. Prime Number Check Using If-Else**

**Aim:**Check if a given number is prime using an if-else statement. **Algorithm:**

1. Take an integer input.
2. Use an if-else statement to check for divisibility from 2 to the square root of the number.
3. Print whether the number is prime or not.

**Code:**

num <- as.integer(readline("Enter an integer: "))’

if (num <= 1) {

print("The number is not prime.")

} else {

is\_prime <- TRUE

for (i in 2:sqrt(num)) {

if (num %% i == 0) {

is\_prime <- FALSE

break

}

}

if (is\_prime) {

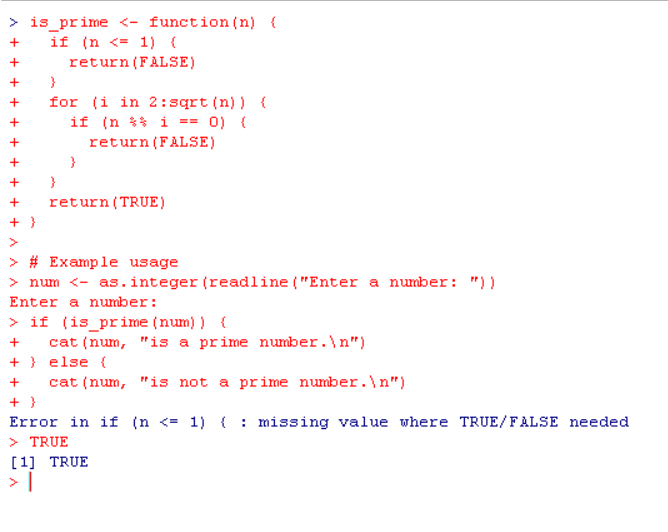
print("The number is prime.")

} else {

print("The number is not prime.")

}

}



**34. Recursive Function to Sum Elements of a Numeric Vector**

**Aim:**Implement a recursive function to sum all the elements of a numeric vector. **Algorithm:**

1. Define a recursive function to sum the vector elements.
2. Base case: If the vector is empty, return 0.
3. Recursive case: Add the first element and call the function again with the rest of the vector.

**Code:**

sum\_recursive <- function(vec) {

if (length(vec) == 0) {

return(0)

} else {

return(vec[1] + sum\_recursive(vec[-1]))

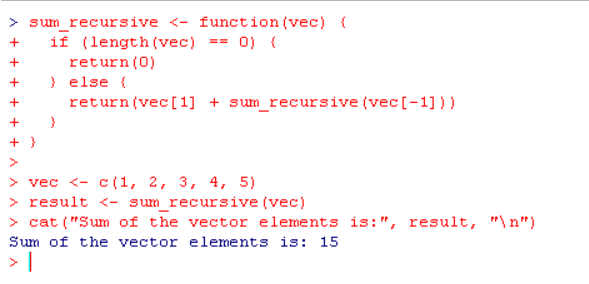
}

}

vec <- c(1, 2, 3, 4, 5)

result <- sum\_recursive(vec)

cat("Sum of the vector elements is:", result, "\n")



**35. Grade Based on Numeric Score**

**Aim:**Print the corresponding grade based on a numeric score using conditional checks. **Algorithm**:

1. Take a numeric score as input.
2. Use if-else to determine the grade based on the score.
3. Print the corresponding grade.

**Code:**

get\_grade <- function(score) {

if (score >= 90) {

return("A")

} else if (score >= 80) {

return("B")

} else if (score >= 70) {

return("C")

} else if (score >= 60) {

return("D")

} else {

return("F")

}

}

score <- as.numeric(readline("Enter your score: "))

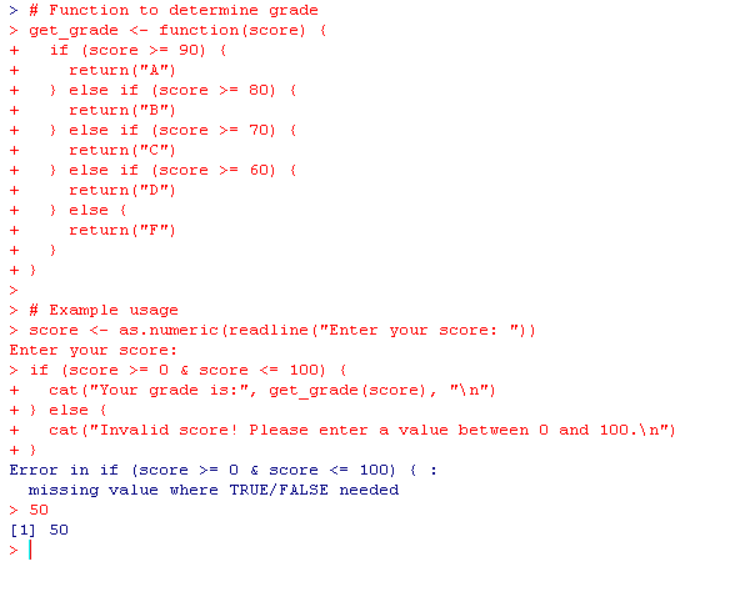
if (score >= 0 & score <= 100) {

cat("Your grade is:", get\_grade(score), "\n")

} else {

cat("Invalid score! Please enter a value between 0 and 100.\n")

}



**36. Replace Vector Elements Based on Sign**

**Aim:**Write a function that replaces each element in a numeric vector with "positive", "negative", or "zero" based on its value. **Algorithm:**

1. Define a function that takes a numeric vector.
2. Loop through the vector and check if the element is positive, negative, or zero using if-else.
3. Return a new vector with the corresponding values.

**Code:**

replace\_sign <- function(vec) {

result <- sapply(vec, function(x) {

if (x > 0) {

return("positive")

} else if (x < 0) {

return("negative")

} else {

return("zero")

}

})

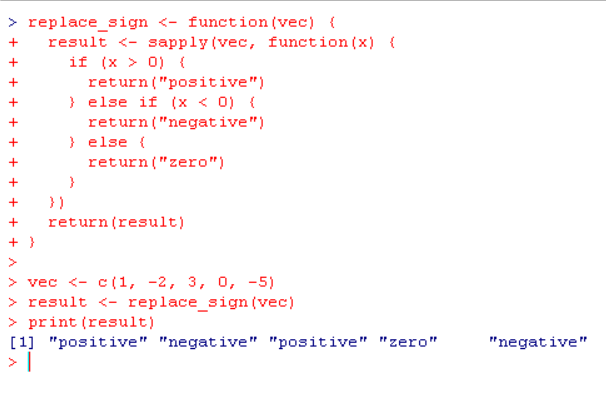
return(result)

}

vec <- c(1, -2, 3, 0, -5)

result <- replace\_sign(vec)

print(result)



**37. Loop Over a List of Character Vectors**

**Aim:** Loop through a list of character vectors representing different categories of items and print the category name along with the number of items. **Algorithm:**

1. Define a list of character vectors representing different categories.
2. Loop through the list, printing the name of each category and the length of each vector.

**Code:**

categories <- list(

fruits = c("apple", "banana", "orange"),

vegetables = c("carrot", "broccoli", "spinach"),

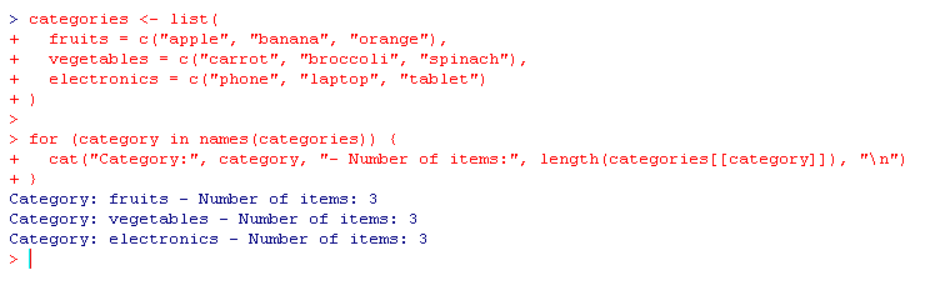
electronics = c("phone", "laptop", "tablet")

)

for (category in names(categories)) {

cat("Category:", category, "- Number of items:", length(categories[[category]]), "\n")

}



**38. Data Frame with Duplicated Product Combinations**

**Aim:**Create a data frame from vectors and display the duplicated product combinations along with the unique customer-product pairs. **Algorithm:**

1. Create vectors representing customers and products.
2. Combine them into a data frame.
3. Use duplicated() to find and display duplicated product combinations and unique customer-product pairs.

**Code:**

customers <- c("John", "Alice", "Bob", "Alice", "John")

products <- c("Laptop", "Phone", "Tablet", "Phone", "Laptop")

df <- data.frame(Customer = customers, Product = products)

duplicated\_combinations <- df[duplicated(df), ]

unique\_pairs <- unique(df)

cat("Duplicated product combinations:\n")

print(duplicated\_combinations)

cat("\nUnique customer-product pairs:\n")

print(unique\_pairs)



**39. Data Frame with Duplicated Patient Treatments**

**Aim:**Create a data frame and identify duplicated treatments for patients, displaying unique patient-treatment combinations**.  
Algorithm:**

1. Create vectors for patient IDs and treatments.
2. Combine them into a data frame.
3. Use duplicated() to find duplicated treatments and unique() for unique patient-treatment pairs.

**Code:**

patients <- c(101, 102, 103, 102, 101)

treatments <- c("Treatment A", "Treatment B", "Treatment A", "Treatment B", "Treatment C")

df <- data.frame(PatientID = patients, Treatment = treatments)

duplicated\_treatments <- df[duplicated(df), ]

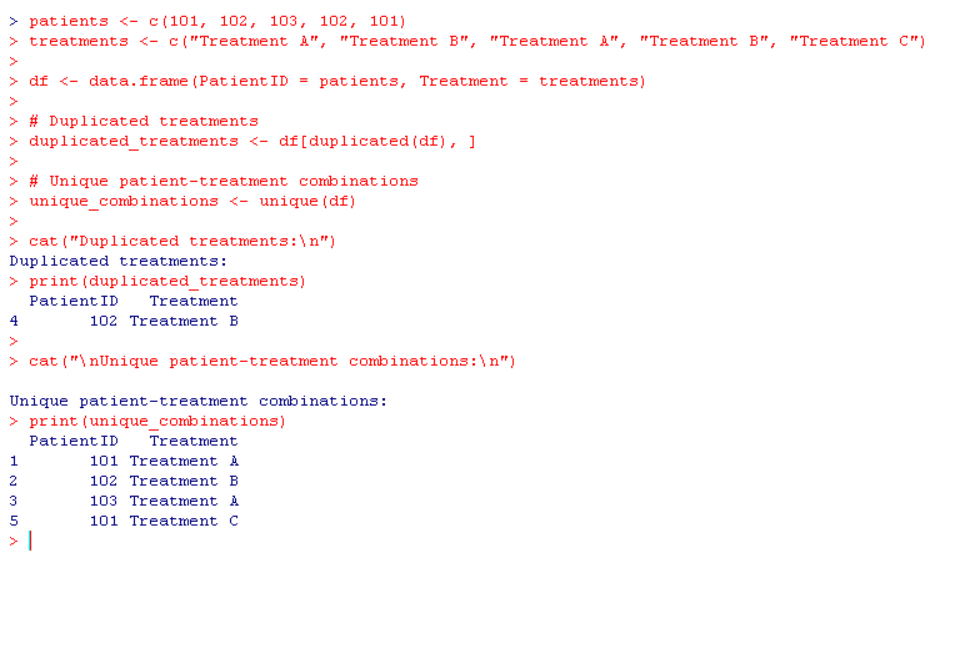
unique\_combinations <- unique(df)

cat("Duplicated treatments:\n")

print(duplicated\_treatments)

cat("\nUnique patient-treatment combinations:\n")

print(unique\_combinations)



**40. Data Frame for Patient-Treatment Combinations**

**Aim:**Create a data frame from patient IDs and treatments, then list duplicated treatments and unique patient-treatment pairs. **Algorithm:**

1. Define vectors for patient IDs and treatments.
2. Create a data frame.
3. Use duplicated() to find duplicated treatments and unique() for unique combinations.

**Code:**

patient\_ids <- c(201, 202, 203, 201, 204)

treatment\_ids <- c("Chemo", "Surgery", "Radiation", "Surgery", "Chemo")

df <- data.frame(PatientID = patient\_ids, TreatmentID = treatment\_ids)

duplicated\_treatments <- df[duplicated(df), ]

unique\_patient\_treatments <- unique(df)

cat("Duplicated treatments for patients:\n")

print(duplicated\_treatments)

cat("\nUnique patient-treatment combinations:\n")

