**Mid-2 Exercises:**

1. **Create a vector and visualize the output using barplot.**

**Source Code:**

# defining vector

x <- c(7, 15, 23, 12, 44, 56, 32)

# output to be present as PNG file

png(file = "barplot.png")

# plotting vector

barplot(x, xlab = "R Programming Lab",

ylab = "Count", col = "white",

col.axis = "darkgreen",

col.lab = "darkgreen")

# saving the file

dev.off()

1. **Write a R program to create a vector and find the length and the dimension of the vector.**

**Source Code:**

# Create a vector 'v' with the elements 1, 3, 5, 7, and 9

v = c(1, 3, 5, 7, 9)

# Print a message indicating the original vector

print("Original vectors:")

# Print the content of the vector 'v'

print(v)

# Print a message indicating the dimension of the vector

print("Dimension of the vector:")

# Print the dimension of the vector 'v' (returns NULL for a 1D vector)

print(dim(v))

# Print a message indicating the length of the vector

print("length of the vector:")

# Print the length of the vector 'v'

print(length(v))

1. **Write a R program to find the sum of all elements in an array using a for loop.**

**Souce Code:**

# Define a function to find the sum of all odd elements in an array using a for loop

sum\_of\_odd\_elements <- function(arr) {

# Initialize a variable to store the sum

sum <- 0

# Iterate through each element in the array

for (element in arr) {

# Check if the element is odd

if (element %% 2 != 0) {

# Add the odd element to the sum

sum <- sum + element

}

}

# Return the sum of odd elements

return(sum)

}

# Test the function with an example array

array <- c(10, 20, 30, 41, 50, 60, 73, 80, 90, 101)

result <- sum\_of\_odd\_elements(array)

# Print the result

cat("Sum of all odd elements in the array is:", result, "\n")

1. **Write a R program function to find the GCD (Greatest Common Divisor) of two numbers using recursion.**

**Source Code:**

# Define a function to find the GCD of two numbers using recursion

gcd <- function(a, b) {

# Base case: If b is 0, GCD is a

if (b == 0) {

return(a)

} else {

# Recursive case: Find GCD using Euclidean algorithm

return(gcd(b, a %% b))

}

}

# Test the function with example inputs

num1 <- 44

num2 <- 16

result <- gcd(num1, num2)

# Print the result

cat("The GCD of", num1, "and", num2, "is:", result, "\n")

1. **Write a R program to check if a given number is Armstrong number using a while loop.**

**Source Code:**

# Define a function to check if a given number is an Armstrong number

is\_armstrong <- function(number) {

# Store the original number in a temporary variable

original\_number <- number

# Initialize variables for storing the sum of cubes of digits and the number of digits

sum\_of\_cubes <- 0

num\_digits <- 0

# Count the number of digits in the given number

while (number > 0) {

num\_digits <- num\_digits + 1

number <- number %/% 10

}

# Reset the number to its original value

number <- original\_number

# Calculate the sum of cubes of digits

while (number > 0) {

digit <- number %% 10

sum\_of\_cubes <- sum\_of\_cubes + digit^num\_digits

number <- number %/% 10

}

# Check if the sum of cubes of digits is equal to the original number

if (sum\_of\_cubes == original\_number) {

return(TRUE) # The number is an Armstrong number

} else {

return(FALSE) # The number is not an Armstrong number

}

}

# Test the function with an example input

number <- 9474 # Example number to check if it's an Armstrong number

is\_armstrong\_number <- is\_armstrong(number)

# Print the result

if (is\_armstrong\_number) {

cat(number, "is an Armstrong number.\n")

} else {

cat(number, "is not an Armstrong number.\n")

}

1. **Write a R program function to check if a given number is a palindrome using recursion.**

**Source Code:**

# Define a function to check if a given number is a palindrome using recursion

is\_palindrome <- function(number) {

# Convert the number to a character string

num\_str <- as.character(number)

# Define a helper function to check if a string is a palindrome

is\_palindrome\_helper <- function(str) {

if (nchar(str) <= 1) {

return(TRUE) # Base case: Single character or empty string is a palindrome

} else {

first\_char <- substr(str, 1, 1) # Get the first character

last\_char <- substr(str, nchar(str), nchar(str)) # Get the last character

# Check if the first and last characters are equal

if (first\_char != last\_char) {

return(FALSE) # If not equal, not a palindrome

} else {

# Recursively check the substring without the first and last characters

return(is\_palindrome\_helper(substr(str, 2, nchar(str) - 1)))

}

}

}

# Call the helper function with the number converted to a string

return(is\_palindrome\_helper(num\_str))

}

# Test the function with example inputs

number1 <- 42324

number2 <- 1234

# Check if the numbers are palindromes and print the result

cat("Number", number1, "is a palindrome!", is\_palindrome(number1), "\n")

cat("Number", number2, "is a palindrome!", is\_palindrome(number2), "\n")

1. **Write a R program to find the reverse of a given number using a while loop.**

**Source Code:**

# Define a function to find the reverse of a given number using a while loop

reverse\_number <- function(number) {

# Initialize variables

reverse <- 0

# Iterate until the number becomes 0

while (number != 0) {

# Extract the last digit of the number

digit <- number %% 10

# Append the digit to the reverse number

reverse <- reverse \* 10 + digit

# Remove the last digit from the number

number <- number %/% 10

}

# Return the reverse of the given number

return(reverse)

}

# Test the function with an example input

number <- 12345678 # Example number to find its reverse

reversed\_number <- reverse\_number(number)

# Print the result

cat("Reverse of", number, "is:", reversed\_number, "\n")

1. **Write a R program to create a vector using : operator and seq() function.**

**Source Code**

# Create a vector 'x' using the : operator, which generates a sequence from 1 to 15

x = 1:15

# Print a message indicating the new vector created with the : operator

print("New vector using : operator-")

# Print the content of the vector 'x'

print(x)

# Print a message indicating the new vector created with the seq() function

print("New vector using seq() function-")

# Print a message specifying that the next vector uses a step size

print("Specify step size:")

# Create a vector 'y' using seq() function from 1 to 3 with a step size of 0.3

y = seq(1, 3, by=0.3)

# Print the content of the vector 'y'

print(y)

# Print a message specifying that the next vector uses a specified length

print("Specify length of the vector:")

# Create a vector 'z' using seq() function from 1 to 5 with a total length of 6 elements

z = seq(1, 5, length.out = 6)

# Print the content of the vector 'z'

print(z)

1. **Write a R program to create an ordered factor from data consisting of the names of months.**

**Source Code:**

# Create a vector of month names

mons\_v = c("March","April","January","November","January",

"September","October","September","November","August","February",

"January","November","November","February","May","August","February",

"July","December","August","August","September","November","September",

"February","April")

# Print the original vector of month names

print("Original vector:")

print(mons\_v)

# Convert the month names into a factor (unordered by default)

f = factor(mons\_v)

# Print the ordered factors of the month names

print("Ordered factors of the said vector:")

print(f)

# Print the frequency table of the factors

print(table(f))

1. **Write a R program to concatenate two given factor in a single factor.**

**Source Code:**

# Create the first factor 'f1' with random samples from LETTERS

f1 <- factor(sample(LETTERS, size=6, replace=TRUE))

# Create the second factor 'f2' with random samples from LETTERS

f2 <- factor(sample(LETTERS, size=6, replace=TRUE))

# Print a message indicating the start of the original factors

print("Original factors:")

# Print the first factor 'f1'

print(f1)

# Print the second factor 'f2'

print(f2)

# Concatenate the levels of both factors and create a new factor 'f'

f = factor(c(levels(f1)[f1], levels(f2)[f2]))

# Print a message indicating the result after concatenation

print("After concatenate factor becomes:")

# Print the concatenated factor 'f'

print(f)