Functions Lab

Assignment Instructions

Complete all questions below. After completing the assignment, knit your document, and download both your .Rmd and knitted output. Upload your files for peer review.

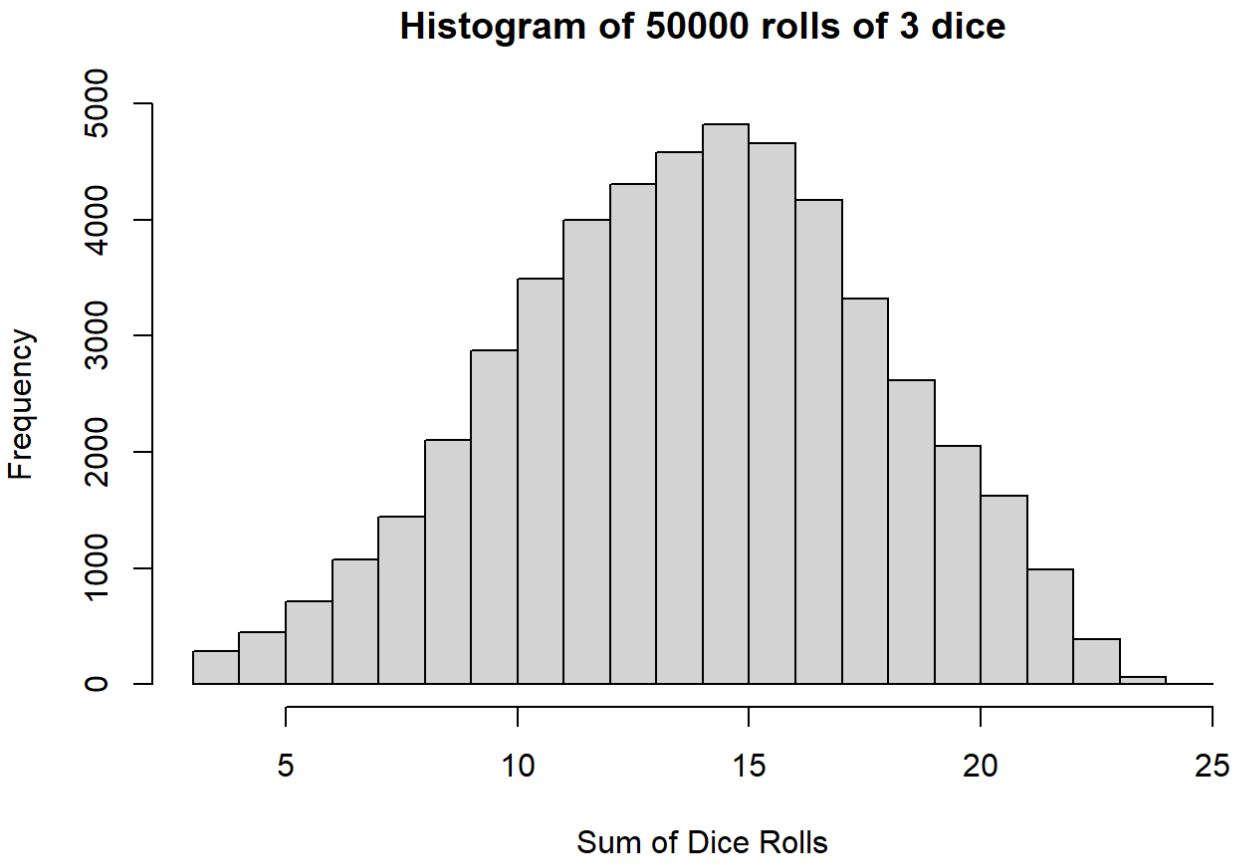
For each response, include comments detailing your response and what each line does. Ensure you test your functions with sufficient test cases to identify and correct any potential bugs.

Question 1.

Review the roll functions from Section 2 in *Hands-On Programming in R*. Using these functions as an example, create a function that produces a histogram of 50,000 rolls of three 8 sided dice. Each die is loaded so that the number 7 has a higher probability of being rolled than the other numbers, assume all other sides of the die have a 1/10 probability of being rolled.

Your function should contain the arguments `max_rolls` , `sides` , and `num_of_dice` . You may wish to set some of the arguments to default values.

```
# Function to simulate loaded dice rolls and produce histogram
simulate_loaded_dice_rolls <- function(max_rolls = 50000, sides = 8, num_of_dice = 3) {

  # Define probabilities for each side of the die
  probabilities <- rep(1/10, sides)
  probabilities[7] <- 2/10  # Setting higher probability for number 7

  # Function to roll a single loaded die
  roll_loaded_die <- function() {
    sample(1:sides, size = 1, prob = probabilities)
  }

  # Function to roll multiple dice and sum their outcomes
  roll_loaded_dice <- function(num_dice) {
    sum(replicate(num_dice, roll_loaded_die()))
  }

  # Simulate rolls
  rolls <- replicate(max_rolls, roll_loaded_dice(num_of_dice))

  # Plot histogram
  hist(rolls, breaks = seq(min(rolls), max(rolls) + 1, by = 1),
       main = paste("Histogram of", max_rolls, "rolls of", num_of_dice, "dice"),
       xlab = "Sum of Dice Rolls",
       ylab = "Frequency")
}

# Example usage:
simulate_loaded_dice_rolls()
```



Question 2.

Write a function, rescale01(), that recieves a vector as an input and checks that the inputs are all numeric. If the input vector is numeric, map any -Inf and Inf values to 0 and 1, respectively. If the input vector is non-numeric, stop the function and return the message “inputs must all be numeric”.

Be sure to thoroughly provide test cases. Additionally, ensure to allow your response chunk to return error messages.

```
# Function to rescale numeric vector to [0, 1]
rescale01 <- function(x) {
  # Check if all elements of x are numeric
  if (!all(is.numeric(x))) {
    stop("inputs must all be numeric")
  }

  # Replace -Inf with 0 and Inf with 1
  x[x == -Inf] <- 0
  x[x == Inf] <- 1

  return(x)
}

# Test cases
# Case 1: Numeric vector with -Inf and Inf
input1 <- c(0.5, 1, -Inf, 2, Inf, 3)
result1 <- rescale01(input1)
print(result1)
```

```
## [1] 0.5 1.0 0.0 2.0 1.0 3.0
```

```
# Expected output: 0.5  1.0  0.0  2.0  1.0  3.0

# Case 2: Numeric vector without -Inf and Inf
input2 <- c(0.1, 0.8, 0.3, 0.9)
result2 <- rescale01(input2)
print(result2)
```

```
## [1] 0.1 0.8 0.3 0.9
```

```
# Expected output: 0.1  0.8  0.3  0.9

# Case 3: Numeric vector with only Inf
input3 <- c(Inf, Inf, Inf)
result3 <- rescale01(input3)
print(result3)
```

```
## [1] 1 1 1
```

```
# Expected output: 1.0  1.0  1.0

# Case 4: Numeric vector with only -Inf
input4 <- c(-Inf, -Inf, -Inf)
result4 <- rescale01(input4)
print(result4)
```

```
## [1] 0 0 0
```

```
# Expected output: 0.0  0.0  0.0

# Case 5: Non-numeric vector
input5 <- c("a", "b", "c")
tryCatch({
  result5 <- rescale01(input5)
  print(result5)
}, error = function(e) {
  print(paste("Error:", e))
})
```

```
## [1] "Error: Error in rescale01(input5): inputs must all be numeric\n"
```

```
# Expected output: Error: inputs must all be numeric
```

Question 3.

Write a function that takes two vectors of the same length and returns the number of positions that have an NA in both vectors. If the vectors are not the same length, stop the function and return the message “vectors must be the same length”.

```r
# Function to count NA values in the same positions of two vectors
count_na_both <- function(vec1, vec2) {
  # Check if vectors are of the same length
  if (length(vec1) != length(vec2)) {
    stop("vectors must be the same length")
  }

  # Count positions where both vectors have NA
  num_na_both <- sum(is.na(vec1) & is.na(vec2))

  return(num_na_both)
}

# Test cases
# Case 1: Vectors with NA at same positions
vec1 <- c(1, NA, 3, NA, 5)
vec2 <- c(NA, 2, NA, 4, NA)
result1 <- count_na_both(vec1, vec2)
print(result1)
```

```
## [1] 0
```

```r
# Expected output: 2 (positions 2 and 5 have NA in both vectors)

# Case 2: Vectors with no NA at same positions
vec3 <- c(1, 2, 3, 4, 5)
vec4 <- c(NA, NA, NA, NA, NA)
result2 <- count_na_both(vec3, vec4)
print(result2)
```

```
## [1] 0
```

```r
# Expected output: 0 (no positions have NA in both vectors)

# Case 3: Vectors of different lengths
vec5 <- c(1, 2, 3)
vec6 <- c(NA, NA, NA, NA)
tryCatch({
  result3 <- count_na_both(vec5, vec6)
  print(result3)
}, error = function(e) {
  print(paste("Error:", e))
})
```

```
## [1] "Error: Error in count_na_both(vec5, vec6): vectors must be the same length\n"
```

```r
# Expected output: Error: vectors must be the same length
```

Question 4

Implement a fizzbuzz function. It takes a single number as input. If the number is divisible by three, it returns

“fizz”. If it’s divisible by five it returns “buzz”. If it’s divisible by three and five, it returns “fizzbuzz”. Otherwise, it returns the number.

```
fizzbuzz <- function(number) {
  if (number %% 3 == 0 && number %% 5 == 0) {
    return("fizzbuzz")
  } else if (number %% 3 == 0) {
    return("fizz")
  } else if (number %% 5 == 0) {
    return("buzz")
  } else {
    return(as.character(number))
  }
}

# Test cases
print(fizzbuzz(3))    # Output: "fizz"
```

```
## [1] "fizz"
```

```
print(fizzbuzz(5))    # Output: "buzz"
```

```
## [1] "buzz"
```

```
print(fizzbuzz(15))   # Output: "fizzbuzz"
```

```
## [1] "fizzbuzz"
```

```
print(fizzbuzz(7))    # Output: "7"
```

```
## [1] "7"
```

Question 5

Rewrite the function below using `cut()` to simplify the set of nested if-else statements.

```
get_temp_desc <- function(temp) {
  if (temp <= 0) {
    "freezing"
  } else if (temp <= 10) {
    "cold"
  } else if (temp <= 20) {
    "cool"
  } else if (temp <= 30) {
    "warm"
  } else {
    "hot"
  }
}
```

```r
get_temp_desc <- function(temp) {
  temperature_labels <- c("freezing", "cold", "cool", "warm", "hot")
  cutpoints <- c(-Inf, 0, 10, 20, 30, Inf)

  desc <- cut(temp, breaks = cutpoints, labels = temperature_labels)

  return(as.character(desc))
}

# Test cases
print(get_temp_desc(-5))   # Output: "freezing"
```

```
## [1] "freezing"
```

```r
print(get_temp_desc(8))    # Output: "cold"
```

```
## [1] "cold"
```

```r
print(get_temp_desc(15))   # Output: "cool"
```

```
## [1] "cool"
```

```r
print(get_temp_desc(25))   # Output: "warm"
```

```
## [1] "warm"
```

```r
print(get_temp_desc(35))   # Output: "hot"
```

```
## [1] "hot"
```