RECT R

(a) Differentiation between "null" and "NA" in the context of missing data in R:

(i) Definition of "null" and its representation in R:

In R, "null" refers to the absence of a value or an object that does not exist. It is used to indicate the lack of any valid data or information. When referring to missing data, "null" is not explicitly used as a representation. Instead, it is typically used in situations where objects or variables have not been assigned any value.

Representation of "null" in R:

When a variable or object doesn't have any assigned value, R will return the keyword "NULL" to represent the absence of data.

Example:

```R
x <- NULL
print(x)
# Output: NULL
```

(ii) Definition of "NA" and its representation in R:

"NA" stands for "Not Available" and is the standard symbol in R to represent missing or undefined values. When data is missing for a particular observation or variable, R uses "NA" to indicate the absence of a valid value. It is essential in data analysis as it helps in handling missing data and avoiding potential issues during computations.

Representation of "NA" in R:

"NA" is used as a special value in R, and it is displayed as such when there are missing values in the data.

Example:

```R
x <- c(1, 2, NA, 4, NA)
print(x)
# Output:  1  2 NA  4 NA
```

(iii) Implications of using "null" and "NA" differently in data analysis:

Using "null" and "NA" differently in data analysis can have significant implications:

1. Handling missing data: "NA" is specifically designed to handle missing data and is widely used in data analysis functions and packages for this purpose. If "null" were used to represent missing data, it might not be recognized by many functions designed to handle "NA," leading to potential errors or incorrect results.

2. Data manipulation and filtering: When filtering or manipulating data, R has built-in functions to deal with "NA" values. If "null" were used, these functions might not work correctly, affecting the integrity of the data analysis.

(iv) Example scenario illustrating the distinction between "null" and "NA" in data analysis:

Let's consider a dataset of students' exam scores. Suppose we have a variable named "math_score," which represents the score of each student in a math exam. Some students did not take the exam, so their scores are missing.

```R
# Using NA to represent missing data
math_score <- c(85, 92, NA, 78, NA, 90)

# Using NULL to represent missing data
math_score_null <- c(85, 92, NULL, 78, NULL, 90)
```

In this scenario, using "NA" correctly indicates that certain students have missing scores. If we use "null" instead, the missing data will not be appropriately recognized by R's functions, potentially leading to incorrect analyses.

(b) Adding new columns to the dataset in R:

(i) Using the `cbind()` function to add a new column named "Gender" with values "Male" and "Female" for each observation:

```R
# Sample dataset
data <- data.frame(Age = c(25, 30, 22, 28, 35),
                   Height = c(170, 165, 180, 160, 175),
                   Weight = c(70, 65, 80, 55, 68))

# Adding a new column "Gender" using cbind()
gender <- c("Male", "Female", "Male", "Female", "Male")
data_with_gender <- cbind(data, Gender = gender)
```

(ii) Calculating "BMI" (Body Mass Index) and adding it as a new column using `cbind()`:

```R
# Assuming "data" already contains "Height" and "Weight" columns
# Calculate BMI as Weight (kg) / Height (m)^2
bmi <- data$Weight / ((data$Height / 100) ^ 2)

# Add BMI as a new column using cbind()
data_with_bmi <- cbind(data, BMI = bmi)
```

(c) Nested lists in R:

(i) Definition and structure of a nested list in R:

A nested list in R is a list that contains multiple elements, and each element can, in turn, be another list or any other data type. It allows you to organize and store complex hierarchical data structures. The structure of a nested list is like a tree, where elements can branch out into sub-elements, forming multiple levels of nesting.

(ii) Creating a nested list in R with at least two levels of nesting:

```R
# Sample nested list
nested_list <- list(
  element1 = "This is a simple element",
  element2 = list(
    sub_element1 = "This is a sub-element within element2",
    sub_element2 = list(
      sub_sub_element1 = "This is a sub-sub-element within element2"
    )
  )
)
```

In the above example, `nested_list` contains two top-level elements: "element1" and "element2." "element2" itself contains two sub-elements: "sub_element1" and "sub_element2," and "sub_element2" contains a sub-sub-element: "sub_sub_element1." This demonstrates two levels of nesting in the list.

(d) Disparities and functional differentiations between lists and data frames in R:

Lists and data frames are both fundamental data structures in R, but they serve different purposes and have distinct characteristics:

1. Lists:

- A list is a collection of elements, and each element can be of any data type (e.g., vectors, matrices, data frames, other lists, etc.).

- Elements in a list can have different lengths.

- Lists are typically used for organizing and storing heterogeneous data and creating complex data structures.

- Accessing elements in a list is done using double brackets `[[ ]]` or the dollar sign `$`.

- Lists are more flexible in terms of structure but might require additional data manipulation for data analysis.

2. Data frames:

- A data frame is a special type of list where each element (column) must have the same length, like a structured table.

- Data frames are commonly used for storing tabular data, where each column represents a variable, and each row represents an observation.

- Data frames have special attributes that make them suitable for data analysis, such as column names and row names (if specified).

- Accessing elements in a data frame is typically done using the column names or indices.

- Data frames support many built-in functions for data manipulation and analysis, such as filtering, merging, and summarizing data.

In summary, lists are more flexible and can hold heterogeneous data with varying lengths, making them suitable for complex data structures. On the other hand, data frames are designed for structured tabular data, providing easier data manipulation and analysis capabilities. Choosing between lists and data frames depends on the nature of the data and the specific requirements of the data analysis task.

(a) Reading and writing files in R:

Reading and writing files in R is an essential part of data manipulation and analysis. R provides several functions to handle different file formats, such as text files, CSV files, Excel files, and more. Here's a step-by-step process for reading and writing files in R:

1. Reading files:

- To read a file, you can use functions like `read.csv()`, `read.table()`, `read.xlsx()` (from the "readxl" package), `readr::read_csv()`, etc., depending on the file format.

Example (reading a CSV file):

```R
# Using read.csv for reading a CSV file
data <- read.csv("data.csv")
```

2. Writing files:

- To write data to a file, use functions like `write.csv()`, `write.table()`, `write.xlsx()` (from the "writexl" package), `writer::write_csv()`, etc.

Example (writing data to a CSV file):

```R
# Using write.csv to write data to a CSV file
write.csv(data, "output.csv", row.names = FALSE)
```

In this example, "data" is the variable containing the data you want to write to the file, and "output.csv" is the name of the file you want to create.

(b) Concepts and techniques for customizing plots in R:

1. Title and axis labels:

- Use the `main`, `xlab`, and `ylab` parameters in plot functions to set the main title, x-axis label, and y-axis label, respectively.

Example:

```R
# Scatter plot with custom title and axis labels
x <- 1:10
y <- x^2
```

```
plot(x, y, main = "Scatter Plot", xlab = "X-axis", ylab = "Y-axis")
```

2. Color selection:

- Colors can be customized using the `col` parameter in plot functions or by specifying color names or hexadecimal color codes.

Example:

```R
# Scatter plot with custom color
x <- 1:10
y <- x^2
plot(x, y, col = "blue")
```

3. Customization of line and point appearances:

- Use parameters like `type`, `lty`, `lwd`, and `pch` to change the type, line style, line width, and point symbol in plot functions, respectively.

Example:

```R
# Line plot with customized appearance
x <- 1:10
y <- x^2
plot(x, y, type = "b", lty = 2, lwd = 2, pch = 16)
```

(c) Significance of "pch" in R for point shapes in graphical representations:

In R, "pch" (plotting character) is a parameter used to specify the shape of points in scatter plots and related graphical representations. The "pch" values are integers representing different point shapes. Here are some important aspects of "pch":

- Specifying "pch" values: You can set the "pch" parameter in plot functions to change the point shape.

- Role in conveying information: Different "pch" values represent different point shapes, and they can be used to differentiate groups or categories in a scatter plot. It allows for visually conveying additional information in a graph.

- Considerations when using "pch": It's essential to choose "pch" values that are easily distinguishable and visually clear. For example, using "pch = 16" for solid circles and "pch = 17" for solid triangles can be helpful when differentiating between two groups.

Example:

```R
# Scatter plot with different "pch" values
x <- 1:10
y <- x^2
plot(x, y, pch = 16)  # Solid circles
points(x, x^2 + 5, pch = 17)  # Solid triangles
```

(d) Features and advantages of ggplot2 in R:

ggplot2 is a powerful data visualization package in R that implements the grammar of graphics. Here are its key features and advantages:

1. Grammar of graphics: ggplot2 follows the grammar of graphics concept, which means it provides a consistent framework for creating plots by combining data, aesthetics, and geometric objects (geoms) with statistical transformations and facetting.

2. Plot creation process: The plot is built step-by-step using layers, where you add data, specify aesthetics (e.g., x and y variables, colors, shapes), choose geoms (e.g., points, lines, bars), and apply statistical transformations if needed.

3. Versatility: ggplot2 is highly versatile and can create a wide range of visualizations, including scatter plots, bar plots, histograms, boxplots, line plots, and more.

4. Aesthetics customization: ggplot2 allows extensive customization of plot aesthetics, such as colors, shapes, sizes, labels, and themes, making it easy to create visually appealing graphs.

5. Geoms: Geoms in ggplot2 define how data points are represented visually. You can add geoms like `geom_point()`, `geom_line()`, `geom_bar()`, etc., to customize the appearance of the plot.

6. Facets: Faceting in ggplot2 enables the creation of small multiples or panels, allowing you to split the data into subsets and create individual plots for each subset.

7. Publication-quality visuals: ggplot2 produces high-quality, publication-ready visuals with default themes and customizable options.

Example:

```R
# Example of ggplot2
library(ggplot2)

# Sample data
data <- data.frame(x = 1:10, y = (1:10)^2)

# Create a scatter plot using ggplot2
ggplot(data, aes(x = x, y = y)) +
  geom_point(shape = 16, size = 3, color = "blue") +
  labs(title = "Scatter Plot", x = "X-axis", y = "Y-axis") +
  theme_minimal()
```

In this example, we create a scatter plot using ggplot2 with customized aesthetics, a title, axis labels, and a minimal theme. The resulting plot is visually appealing and easy to interpret, making ggplot2 a popular choice for data visualization in R.

R CT 2

(a) Concepts of "Int", "NaN", "NULL", and "NA" in R:

1. Int: "Int" is not a specific concept in R. It might refer to integers, which are a data type in R representing whole numbers without decimal points.

Example:

```R
# Integer variable
x <- 10L
```

2. NaN: "NaN" stands for "Not a Number" and is a special floating-point value used to represent undefined or unrepresentable numerical results.

Example:

```R
# NaN value
x <- 0/0
print(x)
# Output: NaN
```

3. NULL: "NULL" represents the absence of a value or an object that doesn't exist. It is used when a variable does not have any assigned value.

Example:

```R
# NULL value
```

```
x <- NULL
print(x)
# Output: NULL
```

4. NA: "NA" is the standard symbol in R to represent missing or undefined values. It is used when data is missing for a particular observation or variable.

Example:

```R
# NA value
x <- c(1, 2, NA, 4, NA)
print(x)
# Output:  1  2 NA  4 NA
```

(b) Creating a Nested list for the table:

```R
nested_list <- list(
  StudentID = c(1, 2, 3, 4),
  Name = c("JohnDoe", "JaneSmith", "AliceBrown", "YourName"),
  Age = c(20, 19, 21, YourAge),
  Gender = c("Male", "Female", "Male", YourGender),
  ContactInfo = list(
    Phone = c("123-456-7890", "123-456-7891", "123-456-7892", "YourPhone"),
    Address = c("123MainSt", "460ElmSt", "789OakSt", "YourAddress"),
    Email = c("john@example.com", "jane@example.com", "alex@example.com", "YourEmail")
  ),
  Courses = list(
```

```
    Compulsory = c("Mathematics", "English", "Chemistry", "YourCompulsory"),
    Optional = c("Physics", "History", "Biology", "YourOptional")
  )
)
```

(c) Creating a data frame and filtering:

```R
# Creating a data frame
df <- data.frame(
  Department = c("HR", "Finance", "Marketing", "YourDepartment"),
  Name = c("John", "Alice", "Jane", "YourName"),
  Age = c(30, 40, 25, YourAge),
  Gender = c("Male", "Female", "Male", YourGender),
  Salary = c(60000, 75000, 50000, YourSalary)
)

# Adding a new column "Department" using column add method
df$Department <- c("HR", "Finance", "Marketing", "YourDepartment")

# Adding a new column "Department" using cbind method
df <- cbind(df, Department = c("HR", "Finance", "Marketing", "YourDepartment"))

# Creating a subset of the data frame
filtered_df <- subset(df, Age >= 35 & Salary < 80000)
```

(d) Recursive function to sum numeric elements in a nested list:

```R
# Recursive function to sum numeric elements in a nested list
sum_numeric_elements <- function(lst) {
  total_sum <- 0
  for (elem in lst) {
    if (is.numeric(elem)) {
      total_sum <- total_sum + elem
    } else if (is.list(elem)) {
      total_sum <- total_sum + sum_numeric_elements(elem)
    }
  }
  return(total_sum)
}

# Example list
myList <- list(1, 2, list(3, 4, list(5, 6, 7)), "Lisa", "Lisa", 7)

# Calling the function
result <- sum_numeric_elements(myList)
print(result)
# Output: 28 (1 + 2 + 3 + 4 + 5 + 6 + 7)
```

PART 2:

(a) Names of the symbols used for point characters values:

1. 1: Empty circle
2. 2: Empty triangle point-up
3. 3: Empty diamond

4. 4: Empty square

(b) Employee Data Visualization using ggplot2:

(i) Apply a query in R to filter employees who have more than 10 years of experience, earn a salary greater than $80,000, and belong to the "Sales" department. Save the filtered data as "filtered_df":

```R
library(dplyr)

# Assuming "df" is the data frame containing employee data
filtered_df <- df %>%
  filter(YearsOfExperience > 10, Salary > 80000, Department == "Sales")
```

(ii) Visualize the distribution of employee ages using a histogram plot:

```R
library(ggplot2)

# Histogram plot
ggplot(filtered_df, aes(x = Age)) +
  geom_histogram(binwidth = 5, fill = "blue", color = "black") +
  labs(title = "Distribution of Employee Ages", x = "Age", y = "Frequency") +
  theme_minimal()
```

(iii) Generate a bar plot showing the average salary for each department in the filtered data. Sort the bars in descending order based on the average salary:

```R
# Bar plot
average_salary <- filtered_df %>%
  group_by(Department) %>%
  summarize(AvgSalary = mean(Salary)) %>%
  arrange(desc(AvgSalary))

ggplot(average_salary, aes(x = reorder(Department, -AvgSalary), y = AvgSalary)) +
  geom_bar(stat = "identity", fill = "green") +
  labs(title = "Average Salary by Department", x = "Department", y = "Average Salary") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

(c) Scatter plot based on the last digit of the roll number:

```R
# Assuming the last digit of your roll number is stored in the variable "last_digit"
last_digit <- 5

# Data for the scatter plot
data <- data.frame(
  LastDigit = c(0:9),
  Color = c("blue", "red", "green", "orange", "purple", "yellow", "pink", "brown", "cyan", "gray"),
  Shape = c(16, 17, 15, 18, 4, 8, 3, 6, 16, 17)
)

# Filter data based on last digit
filtered_data <- data[data$LastDigit == last_digit, ]
```

```
# Scatter plot
ggplot(filtered_data, aes(x = LastDigit, y = LastDigit, color = Color, shape = Shape)) +
  geom_point(size = 5) +
  scale_color_identity() +
  scale_shape_identity() +
  labs(title = "Scatter Plot Based on Last Digit of Roll Number", x = "Last Digit", y = "Last Digit") +
  theme_minimal()
```

(d) Reading and Writing Files in R:

```
# Read the CSV file
students <- read.csv("students.csv")

# View the structure of the data frame
str(students)

# View the first few rows of the data frame
head(students)

# Data manipulation:

# Filter students with a GPA greater than 3.5 and are in grade 12
filtered_students <- subset(students, GPA > 3.5 & Grade == "12th")

# Calculate the average age of the selected students
average_age <- mean(filtered_students$Age)

# Create a new column to indicate if the GPA is excellent (greater than 4.0)
filtered_students$Excellent_GPA <- ifelse(filtered_students$GPA > 4.0, "Yes", "No")
```

Display the modified data frame

filtered_students

Write the modified data frame to a new CSV file

write.csv(filtered_students, "modified_students.csv", row.names = FALSE)```

In this code, we read the "students.csv" file using `read.csv`, performed some data manipulation (filtering students based on GPA and grade level), and then wrote the modified data frame to a new CSV file called "modified_students.csv" using `write.csv`.

CT 1

(a) Fill in the blanks:

1. R uses a **client-server** model, with the client typically being the R console or an IDE, and the server being the **R interpreter**.

2. The R interpreter is written in **C and Fortran** and provides core functionality for data manipulation, statistical analysis, and graphics.

3. R uses a **garbage collection** system to automatically free memory that is no longer needed by the program.

(b) History and Overview of the R programming language:

R is a programming language and open-source software widely used for statistical computing and data analysis. Here is an overview of its key milestones and characteristics:

- R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, in the early 1990s.

- The first version of R, 1.0.0, was released in February 2000.

- R is inspired by the S programming language developed at Bell Laboratories, which is why it has similar syntax and capabilities.

- R is an interpreted language, allowing interactive data analysis and exploration.

- It has a vast and active community of users and developers, leading to a rich ecosystem of packages for various statistical and data analysis tasks.

- R is extensible, and users can create their functions and packages to share their work with others.

- It is platform-independent, supporting Windows, macOS, and Linux.

- R has a strong focus on data visualization, and its "ggplot2" package is widely used for creating publication-quality graphics.

- R's versatility and capabilities have made it a popular choice in various industries, including academia, data science, finance, and bioinformatics.

(c) Data Analysis using R:

```R
# 1. Create a vector containing the weights of the six members.
weights <- c(68, 73, 82, 60, 88, 77)

# 2. Calculate the average weight of the six members.
average_weight <- mean(weights)
print(average_weight)

# 3. Determine the minimum and maximum weights.
min_weight <- min(weights)
max_weight <- max(weights)
print(min_weight)
print(max_weight)
```

```
# 4. Calculate the weight range.
weight_range <- max_weight - min_weight
print(weight_range)

# 5. Assign the weights of the first three members to a new vector.
first_three_weights <- weights[1:3]
print(first_three_weights)

# 6. Extract the length of the new vector.
length_first_three <- length(first_three_weights)
print(length_first_three)
```

(d) Analyzing Taglines using R:

```R
# 1. Create strings for each of the taglines.
tagline1 <- "Be the change"
tagline2 <- "The perfect fit."
tagline3 <- "Taste the difference."

# 2. Concatenate the three taglines into one string, separating each tagline with a newline character.
combined_taglines <- paste(tagline1, tagline2, tagline3, sep = "\n")
print(combined_taglines)

# 3. Extract a substring from the first tagline (characters 4 to 6).
substring_tagline1 <- substr(tagline1, start = 4, stop = 6)
print(substring_tagline1)

# 4. Check if the word "perfect" is present in each of the taglines.
```

```
contains_perfect <- c(
  "perfect" %in% tagline1,
  "perfect" %in% tagline2,
  "perfect" %in% tagline3
)
print(contains_perfect)
```

II. (a) Define a matrix in R and explain how to fill it with values using row and column bindings. Also, provide an example of a matrix filled with random values using these techniques.

In R, a matrix is a two-dimensional data structure with rows and columns. It can be created using the `matrix()` function. Row binding and column binding are techniques to combine multiple matrices into one larger matrix.

- Row Binding: Combining matrices by stacking them vertically, i.e., adding rows.

Example of creating a matrix filled with random values using row binding:

```R
# Create two matrices with random values
mat1 <- matrix(runif(6), nrow = 2)
mat2 <- matrix(runif(6), nrow = 2)

# Row binding to combine the two matrices
combined_mat_row <- rbind(mat1, mat2)
print(combined_mat_row)
```

- Column Binding: Combining matrices by placing them side by side, i.e., adding columns.

Example of creating a matrix filled with random values using column binding:

```R
# Column binding to combine the two matrices
combined_mat_col <- cbind(mat1, mat2)
print(combined_mat_col)
```

(b) Fill in the Blanks:

1. `mat <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)`
2. `row_bind <- rbind(mat1, mat2)`
3. `col_bind <- cbind(mat1, mat2)`
4. `transpose <- t(mat)`
5. `identity_mat <- diag(3)`
6. `addition <- mat1 + mat2`
7. `subtraction <- mat1 - mat2`
8. `multiplication <- mat1 %*% t(mat2)`

(c) Matrix Operations in R:

```R
# Given matrices A and B
A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)
B <- matrix(c(7, 8, 9, 6, 5, 4, 3, 2, 1), nrow = 3, byrow = TRUE)

# 1. Create a new matrix C by horizontally concatenating A and B.
C <- cbind(A, B)

# 2. Extract the first row and third column of C and calculate their sum.
```

```
first_row <- C[1, ]

third_column <- C[, 3]

sum_row_column <- sum(first_row, third_column)


# 3. Replace the diagonal elements of A with the diagonal elements of the 3x3 identity matrix.

A[cbind(1:3, 1:3)] <- diag(3)


# 4. Calculate the transpose of the modified A.

transpose_A <- t(A)


# 5. Perform element-wise addition and subtraction of A and B, and then multiply the resulting matrices.

addition <- A + B

subtraction <- A - B

multiplication <- A * B

# 6. Invert the resulting matrix from the previous step, if possible.

inverse_multiplication <- solve(multiplication)

# 7. Create a 3x3x2 multidimensional array using A and B as the first and second slices, respectively.

multi_array <- array(c(A, B), dim = c(3, 3, 2))


# 8. Extract the element in the first row, second column, and second slice of the multidimensional array.

element <- multi_array[1, 2, 2]
```

(d) Explanation of Terms in the context of R:

a) Matching: Matching is the process of searching for specific patterns or values in

a vector or data frame. It is often used to extract or replace specific elements that match the given pattern.

Example:

```R
# Vector
colors <- c("red", "blue", "green", "yellow", "purple")
# Matching to find elements containing "blue"
matches <- grep("blue", colors)
print(matches) # Output: 2 (index of the element "blue" in the vector)
```

b) Factors: Factors are used to represent categorical data in R. They are useful for handling discrete data with predefined levels or categories. Factors are created using the `factor()` function.

Example:

```R
# Create a factor vector
gender <- factor(c("Male", "Female", "Male", "Male", "Female"))

# View the levels of the factor
print(levels(gender)) # Output: "Female" "Male"
```

c) Identifying Categories: Identifying categories refers to determining the unique values or categories present in a vector or data frame column.

Example:

```R
# Vector
ages <- c(25, 30, 22, 30, 25, 22, 28)

# Identifying unique categories (ages)
unique_ages <- unique(ages)
```

print(unique_ages) # Output: 25 30 22 28

```

d) Defining and Ordering Levels: When creating a factor, you can define and order its levels. This allows you to specify the desired order of the categories.

Example:

```R

# Create a factor with specified levels and order

grade <- factor(c("A", "B", "C", "B", "A"), levels = c("A", "B", "C"), ordered = TRUE)

# View the levels and order of the factor

print(levels(grade)) # Output: "A" < "B" < "C"

```

e) Combining and Cutting: Combining and cutting are techniques used to group continuous data into intervals or bins. It is useful for creating histograms or analyzing data in categories.

Example:

```R

# Vector of ages

ages <- c(25, 30, 22, 28, 33, 40, 27, 38, 23, 31)

# Cut the ages into three intervals: "Young", "Middle-aged", "Old"

age_groups <- cut(ages, breaks = c(20, 30, 40, 50), labels = c("Young", "Middle-aged", "Old"))

# View the result

print(age_groups)

```

In this example, we used `cut()` to group ages into three intervals: "Young" (20-29), "Middle-aged" (30-39), and "Old" (40-49). The result is a factor with the corresponding labels.
```