Unit-1

1. What are data types used in R? Explain.

Ans

In R, data types are used to categorize and represent different types of data. Understanding data types is crucial for working with data effectively and performing operations on them. R offers several fundamental data types, which can be categorized into the following groups:

1. *Numeric:* Numeric data types represent numerical values. They can be further divided into integers and doubles (floating-point numbers).

- *Integer*: Integers are whole numbers without a fractional or decimal component. In R, you can create integers using the `as.integer()` function or by specifying a number without a decimal point.

Example:

R

x <- as.integer(5)

- **Double (Numeric)**: Doubles are numbers with decimal points or fractions. They are the default numeric data type in R.

Example:

R

y <- 3.14

2. *Character:* Character data types represent text or string values. You can create character vectors using single or double quotes

Example:

R

name <- "John"

3. *Logical:* Logical data types represent binary values, either `TRUE` or `FALSE`, which are used for logical operations and conditional statements.

Example:

R

```
is_student <- TRUE
```

4. *Factor:* Factors are used to represent categorical data, such as gender, color, or product categories. Factors are particularly useful for statistical modeling.

Example:

R

```
gender <- factor(c("Male", "Female", "Male", "Female"))
```

5. *Date and Time:* R has data types to handle dates and times. The `Date`, `POSIXct`, and `POSIXlt` classes are commonly used for this purpose.

Example (Date):

R

```
birthday <- as.Date("1990-05-15")
```

Example (POSIXct):

R

```
timestamp <- as.POSIXct("2023-09-08 15:30:00")
```

6. *Complex:* Complex data types represent complex numbers in the form `a + bi`, where `a` and `b` are real numbers, and `i` is the imaginary unit.

Example:

R

```
z <- 2 + 3i
```

7. *Raw:* The raw data type is used to store raw binary data. It is not commonly used in everyday data analysis but can be useful in certain specialized situations.

Example:

R

```
raw_data <- charToRaw("Hello, world!")
```

8. *Lists and Data Frames:* Lists and data frames are more complex data structures in R that can store a combination of different data types, including vectors, matrices, and other lists. Data frames are particularly useful for handling structured data.

Example (List):

R

```
my_list <- list(name = "Alice", age = 30, is_student = TRUE)
```

Example (Data Frame):

R

```
df <- data.frame(name = c("John", "Alice", "Bob"), age = c(25, 30, 22))
```

These are some of the fundamental data types in R. Understanding how to work with these data types is essential for data manipulation, analysis, and visualization in R.

1B. Why use R for your statistical work?

Ans

R is a popular programming language and environment for statistical work, and it offers several compelling reasons why it is a preferred choice for statisticians, data scientists, and researchers:

1. *Rich Statistical Libraries:* R provides a vast array of specialized packages and libraries for statistical analysis. These packages cover a wide range of statistical methods, from basic descriptive statistics to advanced modeling techniques. Some of the most widely used statistical packages include `stats`, `lme4`, `glmnet`, `survival`, and `caret`.

2. *Data Manipulation and Transformation:* R excels in data manipulation, transformation, and cleansing tasks. It offers powerful tools for data reshaping, subsetting, merging, and filtering, making it easier to prepare data for analysis. The `dplyr` and `tidyr` packages are especially popular for data manipulation tasks.

3. *Data Visualization:* R provides excellent data visualization capabilities through packages like `ggplot2`, which allows users to create highly customizable and publication-quality graphs and plots. The ability to visualize data effectively is crucial for understanding patterns and conveying results to others.

4. *Statistical Modeling:* R is well-suited for building and evaluating statistical models. You can easily perform linear and nonlinear regression, generalized linear models, mixed-effects models, time series analysis, and more. Packages like `glm` and `lmer` are widely used for modeling.

5. *Data Integration:* R can seamlessly integrate with various data sources and file formats, including CSV, Excel, databases (e.g., MySQL, PostgreSQL), and web data. This flexibility allows you to work with data from diverse sources.

6. *Open Source and Community-Driven:* R is open-source software, which means it's freely available for anyone to use, modify, and distribute. Its active and vibrant user community continuously develops and maintains packages, ensuring that R stays up-to-date with the latest statistical techniques and best practices.

7. *Reproducibility:* R promotes reproducible research and analysis. By using R Markdown or Jupyter Notebooks with R, you can create documents that combine code, explanations, and visualizations. This makes it easy to share your work with others and reproduce your results.

8. *Cross-Platform Compatibility:* R is available on multiple operating systems (Windows, macOS, Linux), making it accessible to a wide range of users.

9. *Extensibility:* R allows you to write custom functions and packages, giving you the flexibility to adapt it to your specific needs. You can also interface with other programming languages like C++ for performance-critical tasks.

10. *Community Support:* R has a large and active online community of users and experts. This means you can find help, tutorials, and solutions to common problems easily through forums, mailing lists, and social media groups.

11. *Statistical Learning and Education:* R is often used in educational settings for teaching statistics and data analysis. Its syntax is straightforward and conducive to learning for beginners, and it remains a valuable tool for more experienced statisticians.

In summary, R is a versatile and powerful language for statistical work due to its extensive statistical libraries, data manipulation capabilities, visualization tools, and strong community support. It's an ideal choice for those who prioritize statistical analysis and data science tasks in their work.

2. Explain the concept of data frames with examples. Create any dataframe with some synthetic data and write code for different operations that can be applied on data frames.

Ans

A data frame is a fundamental data structure in R used to store tabular data, similar to a spreadsheet or a SQL table. It is a two-dimensional object where data is organized into rows and columns. Each column can have a different data type (e.g., numeric, character, factor), making data frames versatile for handling a wide range of data. Data frames are commonly used for data manipulation, analysis, and visualization in R.

Here's how you can create a data frame in R with some synthetic data and perform various operations on it:

Creating a Data Frame:

```R
# Creating a data frame
df <- data.frame(
```

```
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age = c(25, 30, 22, 28, 21),
  Gender = c("Female", "Male", "Male", "Male", "Female"),
  Score = c(85, 92, 78, 88, 95)
)

# Display the data frame
print(df)
```

This code creates a data frame `df` with four columns: "Name," "Age," "Gender," and "Score."

Operations on Data Frames:

1. **Accessing Data:**

   - To access a specific column, use the `$` operator or square brackets `[]`. For example:

     ```R
     # Access the "Age" column
     ages <- df$Age
     ```

   - To access a specific row, use square brackets `[]`. For example:

```R
# Access the second row
row2 <- df[2, ]
```

2. **Summary Statistics:**

   - You can compute summary statistics for numeric columns using the `summary()` function:

```R
# Summary statistics for the "Score" column
summary(df$Score)
```

3. **Filtering Data:**

   - You can filter rows based on conditions using logical indexing:

```R
# Filter rows where Age is greater than 25
filtered_df <- df[df$Age > 25, ]
```

4. **Sorting Data:**

- You can sort the data frame based on a column using the `order()` function:

```R
# Sort the data frame by "Score" in descending order
sorted_df <- df[order(-df$Score), ]
```

5. **Adding New Columns:**

- You can add new columns to the data frame easily:

```R
# Add a new column "Grade" based on the "Score" column
df$Grade <- ifelse(df$Score >= 90, "A", "B")
```

6. **Grouping and Aggregating Data:**

- You can group data by a specific column and compute aggregate statistics using the `aggregate()` function:

```R
# Group by "Gender" and calculate average age and score
```

summary_by_gender <- aggregate(cbind(Age, Score) ~ Gender, data = df, FUN = mean)

```

7. **Merging Data Frames:**

- You can merge two data frames based on common columns using functions like `merge()` or `cbind()`:

```R
# Merge two data frames by a common column (e.g., "Name")
merged_df <- merge(df1, df2, by = "Name")
```

8. **Subsetting Data:**

- You can extract a subset of the data frame based on specific rows and columns:

```R
# Select specific columns (e.g., "Name" and "Score") for a subset of rows
subset_df <- df[c(1, 3), c("Name", "Score")]
```

These are just a few examples of the operations you can perform on data frames in R. Data frames are highly versatile and provide a wide range of functions and methods to manipulate, analyze, and visualize data efficiently.

3. Create the following vectors in R.

a = (5, 10, 15, 20, ..., 160) b = (87, 86, 85, ..., 56)
```

Use vector arithmetic to multiply these vectors and call the result ‘d’.

Select

subsets of d to identify the following.

i. What are the 19th, 20th, and 21st elements of d?

ii. What are all of the elements of d which are less than 2000?

iii. How many elements of d are greater than 6000?

Ans

You can create vectors 'a' and 'b' using the `seq()` function in R and then perform vector arithmetic to create 'd' and answer the questions. Here's the code to do that:

```R
# Create vector 'a' from 5 to 160 with increments of 5
a <- seq(5, 160, by = 5)

# Create vector 'b' from 87 to 56 with decrements of 1
b <- seq(87, 56, by = -1)

# Multiply vectors 'a' and 'b' to create 'd'
d <- a * b

# i. What are the 19th, 20th, and 21st elements of d?
elements_19_to_21 <- d[19:21]

# ii. What are all of the elements of d which are less than 2000?
elements_less_than_2000 <- d[d < 2000]
```

```
# iii. How many elements of d are greater than 6000?
elements_greater_than_6000 <- sum(d > 6000)

# Print the results
print("19th, 20th, and 21st elements of d:")
print(elements_19_to_21)

print("Elements of d less than 2000:")
print(elements_less_than_2000)

print("Number of elements of d greater than 6000:")
print(elements_greater_than_6000)
```

```
#OUTPUT
[1] "19th, 20th, and 21st elements of d:"
[1] 6555 6800 7035
[1] "Elements of d less than 2000:"
[1]  435  860 12751680
[1] "Number of elements of d greater than 6000:"
[1] 16
```

In this code:

- Vector 'a' is created from 5 to 160 with increments of 5.
- Vector 'b' is created from 87 to 56 with decrements of 1.

- Vector 'd' is created by multiplying 'a' and 'b'.

- We use indexing to find the 19th, 20th, and 21st elements of 'd'.

- We use conditional indexing to find all elements of 'd' that are less than 2000.

- We use the `sum()` function to count the number of elements in 'd' that are greater than 6000.

4. Explain in detail about vectors in R. Mention its characteristics with proper examples.

Ans

In R, a vector is a fundamental data structure that is used to store a sequence of values. Vectors can hold data of the same type, and they are the building blocks for more complex data structures like lists and data frames. Understanding vectors is crucial in R programming as they play a central role in data manipulation and analysis. Here are the key characteristics and examples of vectors in R:

1. **Homogeneous Data:** Vectors in R can only contain elements of the same data type. This means that all elements within a vector must be either numeric, character, logical, etc. You cannot mix different data types in a single vector.

```R
# Numeric vector
numeric_vector <- c(1, 2, 3, 4, 5)

# Character vector
character_vector <- c("apple", "banana", "cherry")

# Logical vector
```

```
logical_vector <- c(TRUE, FALSE, TRUE)
```

2. **Creation of Vectors:** There are various ways to create vectors in R.

- Using the `c()` function: The `c()` function is used to combine values into a vector.

```R
numeric_vector <- c(1, 2, 3)
```

- Using a sequence: The `seq()` function generates a sequence of values.

```R
sequence <- seq(1, 10, by = 2)  # Creates a vector from 1 to 10 with a step of 2
```

- Using repetition: The `rep()` function repeats a value or a sequence.

```R
repetition <- rep(0, times = 5)  # Creates a vector of five 0s
```

3. **Vector Operations:** You can perform various operations on vectors, such as addition, subtraction, multiplication, and division. These operations are applied element-wise.

```R
a <- c(1, 2, 3)
b <- c(4, 5, 6)

# Addition
result_addition <- a + b  # Result: [5, 7, 9]

# Subtraction
result_subtraction <- a - b  # Result: [-3, -3, -3]

# Multiplication
result_multiplication <- a * b  # Result: [4, 10, 18]

# Division
result_division <- a / b  # Result: [0.25, 0.4, 0.5]
```

4. **Vector Indexing:** You can access individual elements of a vector using indexing. In R, indexing starts at 1 (not 0 as in some other programming languages).

```R
my_vector <- c(10, 20, 30, 40, 50)
```

```
# Accessing the first element
first_element <- my_vector[1]  # Result: 10

# Accessing a range of elements
sub_vector <- my_vector[2:4]  # Result: [20, 30, 40]
```

5. **Vector Functions:** R provides various functions for working with vectors, including `length()`, `sum()`, `mean()`, `min()`, `max()`, and many more.

```R
my_vector <- c(10, 20, 30, 40, 50)

vector_length <- length(my_vector)  # Number of elements: 5
vector_sum <- sum(my_vector)       # Sum of elements: 150
vector_mean <- mean(my_vector)     # Mean of elements: 30
vector_min <- min(my_vector)       # Minimum element: 10
vector_max <- max(my_vector)       # Maximum element: 50
```

6. **Vector Names:** You can assign names to elements in a vector using the `names()` function.

```R
```

```
my_vector <- c(10, 20, 30)
names(my_vector) <- c("A", "B", "C")
```

7. **Vector Recycling:** In operations involving vectors of different lengths, R recycles the shorter vector to match the longer one, repeating its elements as needed.

```R
a <- c(1, 2, 3)
b <- c(4, 5)

result <- a + b  # Result: [5, 7, 7] (b is recycled)
```

8. **Vector Attributes:** Vectors can have attributes like names, dimensions, and data types.

```R
my_vector <- c(1, 2, 3)
attributes(my_vector)  # Displays attributes of the vector
```

These are some of the fundamental characteristics and operations related to vectors in R. Vectors are versatile and form the foundation for more advanced data structures and data manipulation tasks in R.

5. Discuss about matrices in R. Demonstrate various operations on matrices in R.

Ans

In R, a matrix is a two-dimensional data structure that consists of rows and columns, similar to a mathematical matrix. Matrices are used to store and manipulate data when you need a structured grid-like format. Here, I'll discuss matrices in R and demonstrate various operations you can perform on them.

Creating Matrices:

You can create matrices in R using the `matrix()` function. Here's how to create a simple matrix:

```R
# Create a 3x3 matrix
mat <- matrix(1:9, nrow = 3, ncol = 3)
print(mat)
```

This code creates a 3x3 matrix with values from 1 to 9.

Matrix Operations:

1. **Matrix Addition and Subtraction:**

You can perform element-wise addition and subtraction on matrices of the same dimensions.

```R
mat1 <- matrix(1:9, nrow = 3, ncol = 3)
mat2 <- matrix(9:1, nrow = 3, ncol = 3)

# Matrix addition
result_add <- mat1 + mat2

# Matrix subtraction
result_sub <- mat1 - mat2

print("Matrix Addition:")
print(result_add)

print("Matrix Subtraction:")
print(result_sub)
```

```
     [,1] [,2] [,3]
[1,]   10   10   10
[2,]   10   10   10
[3,]   10   10   10

     [,1] [,2] [,3]
[1,]   -8   -8   -8
```

[2,] -8 -8 -8

[3,] -8 -8 -8

2. **Matrix Multiplication:**

   You can perform matrix multiplication using `%*%` for inner products.

   ```R
   mat1 <- matrix(1:6, nrow = 2)
   mat2 <- matrix(7:12, nrow = 2)

   # Matrix multiplication
   result_mult <- mat1 %*% t(mat2)  # Transpose mat2 to match dimensions

   print("Matrix Multiplication:")
   print(result_mult)
   ```

    [,1] [,2]

[1,] 32 77

[2,] 39 92

3. **Matrix Transposition:**

   You can transpose a matrix using the `t()` function.

   ```R
   mat <- matrix(1:6, nrow = 2)
   ```

```
# Transpose the matrix
transposed_mat <- t(mat)

print("Transposed Matrix:")
print(transposed_mat)
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    3    5
```

4. **Matrix Inversion:**

   You can find the inverse of a square matrix using the `solve()` function.

```R
mat <- matrix(c(1, 2, 2, 3), nrow = 2)

# Find the inverse of the matrix
inverse_mat <- solve(mat)

print("Inverse Matrix:")
print(inverse_mat)
```

```
     [,1] [,2]
```

[1,] -3   2

[2,]   2 -1

5. **Element-Wise Operations:**

You can perform element-wise operations, such as exponentiation or taking the square root of a matrix.

```R
mat <- matrix(1:9, nrow = 3, ncol = 3)

# Element-wise square root
sqrt_mat <- sqrt(mat)

# Element-wise exponentiation
exp_mat <- exp(mat)

print("Element-Wise Square Root:")
print(sqrt_mat)

print("Element-Wise Exponentiation:")
print(exp_mat)
```

```
     [,1] [,2] [,3]
[1,]    1  1.41 1.73
[2,] 1.15 1.41 1.65
```

[3,] 1.24 1.48 1.73

6. **Matrix Indexing:**

You can access specific elements or subsets of a matrix using indexing.

```R
mat <- matrix(1:9, nrow = 3, ncol = 3)

# Access the element in the first row and second column
element <- mat[1, 2]

# Access the second row
row2 <- mat[2, ]

print("Accessed Element:")
print(element)

print("Accessed Second Row:")
print(row2)
```

These are some of the basic operations you can perform on matrices in R. Matrices are versatile data structures that are widely used in mathematics, statistics, and data analysis for various purposes, including linear algebra, regression analysis, and more.

6. What is a list? Explain the concept of lists in R with examples.

Ans

In R, a list is a versatile and fundamental data structure that can hold a collection of objects, including vectors, matrices, data frames, other lists, and even functions. Lists are used to organize and store heterogeneous data elements in a structured manner. Unlike vectors or matrices, lists can contain elements of different data types. Here's a detailed explanation of lists in R with examples:

Creating Lists:

You can create a list in R using the `list()` function. Elements within a list are separated by commas and enclosed within parentheses. Here's how to create a simple list:

```R
# Create a list with different types of elements
my_list <- list(
  name = "John",
  age = 30,
  scores = c(85, 92, 78),
  is_student = TRUE
)
```

```
$name
[1] "John"
```

$age

[1] 30

$scores

[1] 85 92 78

$is_student

[1] TRUE

In this example, `my_list` is a list containing four elements: a character element "name," a numeric element "age," a numeric vector "scores," and a logical element "is_student."

Accessing List Elements:

You can access individual elements of a list using double square brackets `[[]]` or by using the dollar sign `$`. The double square brackets are used when you know the name or position of the element, while the dollar sign is used when you know the name of the element.

```R
# Access list elements
name <- my_list$name
age <- my_list[["age"]]
scores <- my_list$scores
is_student <- my_list[["is_student"]]
```

```
[1] "Name:"
[1] "John"
[1] "Age:"
[1] 30
[1] "Scores:"
[1] 85 92 78
[1] "Is Student:"
[1] TRUE
```

Adding Elements to a List:

You can add elements to an existing list using the list indexing notation or the `$` operator.

```R
# Add a new element to the list
my_list$city <- "New York"
my_list[["hobbies"]] <- c("reading", "hiking")
```

```
$name
[1] "John"

$age
[1] 30

$scores
```

[1] 85 92 78

$is_student
[1] TRUE

$city
[1] "New York"

$hobbies
[1] "reading" "hiking"
Nested Lists:

Lists in R can be nested, meaning you can have lists within lists. This allows you to create hierarchical or structured data.

```R
# Create a nested list
nested_list <- list(
  person = my_list,
  address = list(
    street = "123 Main St",
    city = "Los Angeles"
  )
)
```

$person

```
$person$name
[1] "John"

$person$age
[1] 30

$person$scores
[1] 85 92 78

$person$is_student
[1] TRUE

$person$city
[1] "New York"

$person$hobbies
[1] "reading" "hiking"


$address
$address$street
[1] "123 Main St"

$address$city
[1] "Los Angeles"
```

In this example, `nested_list` contains a list of a person's information and another list for their address.

List of Lists:

You can create a list that contains multiple lists, making it useful for organizing and managing complex data structures.

```R
# Create a list of lists
list_of_lists <- list(
  list1 = list(a = 1, b = 2),
  list2 = list(x = "apple", y = "banana")
)
```

Lists as Data Frames:

Lists can be converted into data frames, and vice versa, allowing you to work with structured data.

```R
# Convert a list to a data frame
df_from_list <- data.frame(my_list)
```

Lists of Functions:

Lists can also store functions or references to functions, making it convenient for creating dynamic code structures.

```R
# Create a list of functions
function_list <- list(
  square = function(x) x^2,
  cube = function(x) x^3
)

# Call a function from the list
result <- function_list$square(5)  # Result: 25
```

In summary, lists in R are flexible and versatile data structures that can store heterogeneous data elements, making them a powerful tool for organizing and managing complex data, hierarchical structures, and even functions. Lists are commonly used in R for various data manipulation, analysis, and programming tasks.

7A. Explain R sessions in detail.

Ans

In R, a session refers to the period during which you are actively working with the R environment. It encompasses the time from when you start R to

when you exit R. During an R session, you can interactively enter and execute R commands, load data, perform data analysis, create visualizations, and more. Let's explore R sessions in detail with examples, including their inputs and outputs.

Starting an R Session:

You typically start an R session by launching the R console or an integrated development environment (IDE) like RStudio. Once you have started R, you can begin interacting with it.

Executing R Commands:

During an R session, you can execute R commands interactively. For example:

Input:
```R
# Assign values to variables
x <- 5
y <- 10

# Perform arithmetic operations
sum_result <- x + y

# Print the result
print(sum_result)
```

```

*Output:*
```
[1] 15
```

In this example, we assigned values to variables `x` and `y`, performed addition, and printed the result. This interactive nature of R allows for quick experimentation and exploration of data.

**Loading Data:**

You can load data into an R session using functions like `read.csv()` to read CSV files, `read.table()` for tabular data, or database connectors for database interactions.

*Input:*
```R
# Load a CSV file
data <- read.csv("data.csv")

# View the first few rows of the data
head(data)
```
```

Output:

```
  Name Age Gender Score
1 John  25   Male    85
2 Alice 30 Female    92
3   Bob  22   Male    78
4 David 28   Male    88
5   Eve  21 Female    95
```

Here, we loaded a CSV file named "data.csv" into the R session and displayed the first few rows of the data using the `head()` function.

Data Analysis:

R is widely used for data analysis. You can perform various data analysis tasks, such as calculating summary statistics, creating plots, and conducting statistical tests.

Input:

```R
# Calculate summary statistics
summary(data$Score)

# Create a histogram of scores
hist(data$Score)
```

```
# Perform a t-test
t_test_result <- t.test(data$Score ~ data$Gender)
print(t_test_result)
```

Output:

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  78.00   85.00   88.00   87.60   92.00   95.00

   Welch Two Sample t-test

data:  data$Score by data$Gender
t = -1.6329, df = 3.6356, p-value = 0.1849
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -12.468927   2.268927
sample estimates:
mean in group Female   mean in group Male
                  93                  82
```

In this example, we calculated summary statistics, created a histogram of scores, and conducted a t-test to compare scores by gender.

Creating Visualizations:

R is known for its powerful data visualization capabilities. You can create various types of plots and charts to visualize data.

Input:

```R
# Create a scatter plot
plot(data$Age, data$Score, xlab = "Age", ylab = "Score", main = "Scatter Plot")

# Create a bar chart of gender distribution
barplot(table(data$Gender), main = "Gender Distribution")
```

Output:

(Outputs are visual plots or charts displayed in a separate graphics window or saved as image files.)

In this example, we created a scatter plot to visualize the relationship between age and score and a bar chart to display the distribution of genders in the data.

Saving Your Work:

You can save your R session or your script to preserve your work for future sessions.

Input:

```R
# Save the current workspace
save.image(file = "my_workspace.RData")
```

In this example, we saved the current workspace to a file named "my_workspace.RData." This allows you to load it in a future session using `load()`.

Ending an R Session:

To end an R session, you can simply close the R console or the IDE you are using.

In summary, an R session is the period during which you work with the R environment, performing tasks such as executing commands, loading data, conducting data analysis, creating visualizations, and saving your work. It provides an interactive and exploratory environment for data analysis and statistical computing.

7B. Why R? Give features of R.

Ans

R is a popular open-source programming language and environment specifically designed for statistical analysis, data visualization, and data science. It has gained immense popularity in academia and industry due to its powerful features and extensive ecosystem. Here are some key features of R that make it a preferred choice for statistical work and data analysis:

1. **Open Source and Free:** R is open-source software, which means it's freely available to anyone. This accessibility has contributed to its widespread adoption in both academia and industry.

2. **Rich Statistical Functionality:** R is equipped with a vast library of statistical and mathematical functions. It covers a wide range of statistical techniques, including linear and nonlinear modeling, time-series analysis, hypothesis testing, and more.

3. **Data Visualization:** R provides excellent data visualization capabilities. It has a robust graphics system that allows users to create a wide variety of static and interactive plots and charts. Popular visualization packages like ggplot2 make complex visualizations relatively straightforward.

4. **Data Manipulation:** R excels in data manipulation tasks. It offers powerful tools for data cleaning, transformation, aggregation, and reshaping. The `dplyr` and `tidyr` packages are particularly popular for data wrangling.

5. **Data Import and Export:** R supports a wide range of data formats, including CSV, Excel, SQL databases, JSON, and more. This makes it easy to import and export data from various sources.

6. **Extensive Package Ecosystem:** R has a vast ecosystem of packages contributed by the R community. These packages extend R's functionality and cover a wide array of domains, including machine learning, text mining, spatial analysis, and more.

Overall, R is a versatile and powerful tool for statistical analysis and data science that caters to the needs of both beginners and experts in the field. Its rich ecosystem of packages and active community support make it an attractive choice for anyone working with data.

UNIT-2

1A. What are basic operations in R? Explain with examples.

Ans

Basic operations in R encompass a wide range of tasks, from arithmetic operations and data manipulation to logical operations and control structures. Here, I'll explain some of the fundamental basic operations in R with examples and provide their outputs.

Arithmetic Operations:

1. **Addition and Subtraction:**

```R
```

```
a <- 5
b <- 3

# Addition
result_addition <- a + b  # Result: 8

# Subtraction
result_subtraction <- a - b  # Result: 2
```

2. **Multiplication and Division:**

```R
x <- 10
y <- 2

# Multiplication
result_multiplication <- x * y  # Result: 20

# Division
result_division <- x / y  # Result: 5
```

3. **Exponentiation and Modulo:**

```R
```

```
base <- 2
exponent <- 3

# Exponentiation
result_exponentiation <- base ^ exponent  # Result: 8

# Modulo (remainder)
result_modulo <- 10 %% 3  # Result: 1
```

Logical Operations:

4. **Logical Operators:**

```R
p <- TRUE
q <- FALSE

# Logical AND
result_and <- p & q  # Result: FALSE

# Logical OR
result_or <- p | q  # Result: TRUE

# Logical NOT
result_not <- !p  # Result: FALSE
```

```

**Comparison Operations:**

5. **Comparison Operators:**

```R
x <- 5
y <- 3

# Equality
result_equal <- x == y  # Result: FALSE

# Inequality
result_inequal <- x != y  # Result: TRUE

# Greater than
result_greater <- x > y  # Result: TRUE

# Less than or equal to
result_less_equal <- x <= y  # Result: FALSE
```

Data Manipulation:

6. **Creating Vectors:**
```

```R
numeric_vector <- c(1, 2, 3, 4, 5)
character_vector <- c("apple", "banana", "cherry")
```

7. **Indexing and Slicing:**

```R
my_vector <- c(10, 20, 30, 40, 50)

# Accessing the first element
first_element <- my_vector[1]  # Result: 10

# Accessing a range of elements
sub_vector <- my_vector[2:4]  # Result: [20, 30, 40]
```

Control Structures:

8. **Conditional Statements:**

```R
x <- 5

if (x > 0) {
```

```
  message("x is positive")
} else {
  message("x is non-positive")
}
```

Output:

```
x is positive
```

9. **Loops (for and while):**

```R
# For loop
for (i in 1:5) {
  cat("Iteration:", i, "\n")
}
```

Output:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

```
Iteration: 5
```

These are some of the basic operations in R, including arithmetic operations, logical and comparison operations, data manipulation, and control structures. R's interactive nature allows you to experiment and see the results immediately, making it a powerful tool for data analysis and programming.

1B. Describe loops in R programming.

In R programming, loops are control structures that allow you to execute a block of code repeatedly. Loops are essential when you need to perform a specific task multiple times, such as iterating through a sequence of values, processing elements in a data structure, or carrying out simulations. R supports several types of loops, including for loops, while loops, and repeat loops. Let's dive into each type of loop and explain their use:

1. **For Loops:**

A for loop is used when you know in advance how many times you want to execute a block of code. It iterates over a sequence of values, such as a numeric vector or a list.

```R
# Example of a for loop
for (i in 1:5) {
  cat("Iteration:", i, "\n")
```

}
```

**Output:**
```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

In this example, the for loop iterates over the values 1 through 5, and for each iteration, it executes the code inside the loop, which prints the iteration number.

2. **While Loops:**

A while loop is used when you want to repeat a block of code as long as a specified condition is true. The loop continues executing until the condition becomes false.

```R
# Example of a while loop
x <- 1
while (x <= 5) {
```
```

```
  cat("Value of x:", x, "\n")
  x <- x + 1
}
```

Output:

```
Value of x: 1
Value of x: 2
Value of x: 3
Value of x: 4
Value of x: 5
```

In this example, the while loop continues as long as `x` is less than or equal to 5. The code inside the loop increments `x` by 1 in each iteration.

3. **Repeat Loops:**

A repeat loop is used when you want to repeatedly execute a block of code without any specified condition. The loop continues until you explicitly use the `break` statement to exit the loop.

```R
# Example of a repeat loop with break
x <- 1
```

```
repeat {
  cat("Value of x:", x, "\n")
  x <- x + 1
  if (x > 5) {
    break
  }
}
```

Output:

```
Value of x: 1
Value of x: 2
Value of x: 3
Value of x: 4
Value of x: 5
```

In this example, the repeat loop continues indefinitely until `x` exceeds 5. When `x` becomes greater than 5, the `break` statement is used to exit the loop.

These are the main types of loops in R. You can choose the appropriate loop type based on your specific task and requirements. Loops are commonly used in R for tasks such as iterating through data, simulations, and solving optimization problems. It's essential to be cautious when using loops to avoid

infinite loops and to optimize code for efficiency, especially when working with large datasets.

2. What is recursion in R? Why use recursive functions? Write and explain any R recursive program.

Ans

Recursion in R:

Recursion is a programming technique in which a function calls itself in order to solve a problem. In R, like in many other programming languages, you can create recursive functions. Recursive functions consist of two parts: the base case(s) and the recursive case(s).

- **Base case:** The base case is the condition under which the recursion stops. It provides the termination point for the recursive calls.

- **Recursive case:** The recursive case is where the function calls itself with a modified argument(s). The idea is that each recursive call moves closer to the base case, eventually leading to the base case being reached and the recursion ending.

Why Use Recursive Functions in R:

Recursive functions are particularly useful when dealing with problems that can be broken down into smaller, similar subproblems. They can make the code more concise and elegant in such cases. Recursive functions are

commonly used in mathematics, computer science, and various algorithmic problems.

Here's an example of a simple recursive function in R:

```R
# Recursive function to calculate the factorial of a number
factorial <- function(n) {
  if (n == 0) {
    return(1)  # Base case: factorial of 0 is 1
  } else {
    return(n * factorial(n - 1))  # Recursive case
  }
}

# Calculate the factorial of 5
result <- factorial(5)
print(result)
```

Output:

```
[1] 120
```

In this example, the `factorial` function calculates the factorial of a number `n`. The base case is when `n` equals 0, in which case the function returns 1 (since the factorial of 0 is defined as 1). In the recursive case, the function multiplies `n` by the factorial of `n-1`, effectively reducing the problem to a smaller subproblem. The recursion continues until it reaches the base case, and then it starts "unwinding" or returning the results back up the call stack.

In this case, `factorial(5)` is computed as follows:
- `factorial(5)` calls `factorial(4)`
- `factorial(4)` calls `factorial(3)`
- `factorial(3)` calls `factorial(2)`
- `factorial(2)` calls `factorial(1)`
- `factorial(1)` calls `factorial(0)`
- `factorial(0)` returns 1

Then, the results are multiplied back up the call stack:
- `factorial(1)` returns 1 * 1 = 1
- `factorial(2)` returns 2 * 1 = 2
- `factorial(3)` returns 3 * 2 = 6
- `factorial(4)` returns 4 * 6 = 24
- `factorial(5)` returns 5 * 24 = 120

This demonstrates the concept of recursion and how it can be used to solve problems by breaking them down into smaller, similar subproblems.

3A. Explain if-else statements with examples.

Ans

In R, if-else statements are used to control the flow of a program based on a condition. These statements allow you to specify different code blocks to execute depending on whether a condition is true or false. Here's the basic structure of an if-else statement:

```R
if (condition) {
  # Code to execute if the condition is true
} else {
  # Code to execute if the condition is false
}
```

- `condition`: A logical expression that evaluates to either `TRUE` or `FALSE`.
- The code block inside the `if` block is executed if the condition is `TRUE`.
- The code block inside the `else` block is executed if the condition is `FALSE`.

Here are some examples of if-else statements in R:

Example 1: Simple if-else statement

```R
# Check if a number is even or odd
num <- 7
```

```
if (num %% 2 == 0) {
  cat(num, "is even.")
} else {
  cat(num, "is odd.")
}
```

Output:

```
7 is odd.
```

In this example, we check if the number `num` is even or odd. The condition `num %% 2 == 0` checks if `num` is divisible by 2 without a remainder. If it is, the code inside the `if` block is executed, which prints that the number is even. Otherwise, the code inside the `else` block is executed, indicating that the number is odd.

Example 2: Nested if-else statements

```R
# Determine the category of a student based on their score
score <- 85

if (score >= 90) {
```

```
  cat("Excellent")
} else if (score >= 80) {
  cat("Good")
} else if (score >= 70) {
  cat("Average")
} else {
  cat("Below Average")
}
```

Output:

```
Good
```

In this example, we determine the category of a student based on their exam score. The code uses a series of `if-else if` statements to check different score ranges and print the corresponding category. In this case, the student's score of 85 falls into the "Good" category.

Example 3: Using if-else within a function

```R
# Function to check if a number is positive, negative, or zero
classify_number <- function(x) {
  if (x > 0) {
```

```
    return("Positive")
  } else if (x < 0) {
    return("Negative")
  } else {
    return("Zero")
  }
}

result <- classify_number(-5)
cat("The number is", result)
```

Output:

```
The number is Negative
```

In this example, we define a function `classify_number` that takes a number `x` as input and returns whether it's positive, negative, or zero. The function uses if-else statements to determine the category and returns the result.

These examples illustrate how if-else statements are used to make decisions in R based on conditions, allowing your code to be more flexible and responsive to different scenarios.

3B. Discuss about return values in R.

Ans

In R, return values are the values that a function can send back to the caller. When you call a function, you often want to receive some result or output from that function. This result or output is typically referred to as the "return value" of the function. Here are some important points to understand about return values in R:

1. **Purpose of Return Values:**

   - **Communication:** Return values allow functions to communicate information or results to the code that called them. It's a way for a function to provide its output or answer to the calling code.

   - **Reuse:** Return values make it possible to reuse the result of a function in different parts of your code. You can store, manipulate, or display the result as needed.

2. **Use of `return()` Function:**

   In R, you use the `return()` function to specify the value that a function should return. When `return()` is called within a function, it immediately exits the function and sends the specified value back to the caller.

   ```R
   my_function <- function() {
     result <- 42
     return(result)
```

```
}
```

In this example, the `return()` function is used to send the value `42` back to the caller.

3. **Implicit Return:**

In R, a function will also implicitly return the value of the last evaluated expression if there is no explicit `return()` statement. This means that the return value is the result of the last expression in the function's body.

```R
my_function <- function() {
  result <- 42
  result * 2  # Implicit return value
}
```

In this case, the function `my_function()` will return `84` because `result * 2` is the last expression evaluated.

4. **Multiple Return Values:**

R functions can return multiple values in the form of a list, vector, or data frame. This allows you to package and return multiple results.

```R
get_statistics <- function(data) {
  mean_value <- mean(data)
  median_value <- median(data)
  return(list(mean = mean_value, median = median_value))
}
```

In this example, the `get_statistics()` function returns a list containing the mean and median of the input data.

5. **Capture Return Values:**

When you call a function that returns a value, you can capture and store that value in a variable.

```R
result <- my_function()
```

Here, `result` will contain the value returned by `my_function()`.

6. **Handling Return Values:**

Depending on the function and its purpose, you may need to check, process, or manipulate the return values. For example, you might use conditional statements to make decisions based on return values, store them in variables for further analysis, or display them to the user.

Return values are a fundamental concept in R (and programming in general) that allow you to create reusable and flexible code by encapsulating logic and providing a means of passing information between different parts of your program. Understanding how functions return values is crucial for effective R programming.

4. Describe quicksort implementation using recursion.

Ans

Quicksort is a popular sorting algorithm that uses a divide-and-conquer strategy to sort an array or list of elements. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Here's an implementation of the Quicksort algorithm in R using recursion:

```R
# Quicksort function using recursion
quicksort <- function(arr) {
  # Base case: If the input array has one or zero elements, it's already sorted
```

```
  if (length(arr) <= 1) {
    return(arr)
  }

  # Select a pivot element (in this example, we choose the first element)
  pivot <- arr[1]

  # Partition the array into two sub-arrays
  less_than_pivot <- arr[arr < pivot]
  equal_to_pivot <- arr[arr == pivot]
  greater_than_pivot <- arr[arr > pivot]

  # Recursively sort the sub-arrays
  sorted_less <- quicksort(less_than_pivot)
  sorted_greater <- quicksort(greater_than_pivot)

  # Concatenate the sorted sub-arrays and the equal_to_pivot array
  result <- c(sorted_less, equal_to_pivot, sorted_greater)

  return(result)
}

# Example usage:
unsorted_array <- c(6, 2, 8, 3, 1, 7, 4, 5)
sorted_array <- quicksort(unsorted_array)
cat("Sorted array:", sorted_array, "\n")
```

```

**Output:**

```
Sorted array: 1 2 3 4 5 6 7 8
```

Here's how the Quicksort implementation works:

1. The `quicksort` function takes an array `arr` as input.

2. The base case checks if the length of the input array is one or zero. If it is, the array is already considered sorted, and it is returned as is.

3. If the base case is not met, the function selects a pivot element. In this example, we choose the first element as the pivot.

4. The array is partitioned into three sub-arrays: `less_than_pivot` (elements less than the pivot), `equal_to_pivot` (elements equal to the pivot), and `greater_than_pivot` (elements greater than the pivot).

5. The `quicksort` function is recursively called on the `less_than_pivot` and `greater_than_pivot` sub-arrays.

6. Finally, the sorted sub-arrays and the `equal_to_pivot` array are concatenated to produce the sorted result, which is then returned.
```

The Quicksort algorithm continues to divide the array into smaller sub-arrays until the base case is met for each sub-array. This divide-and-conquer approach efficiently sorts the entire array. Quicksort is known for its average-case time complexity of O(n log n), making it a popular choice for sorting large datasets.

5A. Justify the following: “R functions are first-class objects”.

Ans

In programming, the concept of "first-class objects" refers to the ability to treat functions as values or entities that can be assigned to variables, passed as arguments to other functions, and returned as results from functions. In the context of R, the statement "R functions are first-class objects" is indeed justified, and here's why:

1. **Functions Can Be Assigned to Variables:**

In R, you can assign a function to a variable just like any other data type. This means you can create a variable that references a function and use that variable to call the function.

```R
# Define a function
add <- function(a, b) {
  return(a + b)
}
```

```
# Assign the function to a variable
my_function <- add

# Use the variable to call the function
result <- my_function(3, 4)  # Equivalent to add(3, 4)
```

2. **Functions Can Be Passed as Arguments:**

R allows you to pass functions as arguments to other functions. This feature is commonly used in functional programming and is particularly powerful for creating higher-order functions.

```R
# Define a function that takes another function as an argument
apply_operation <- function(operation, a, b) {
  return(operation(a, b))
}

# Define a function to add two numbers
add <- function(a, b) {
  return(a + b)
}

# Use apply_operation to apply the add function
result <- apply_operation(add, 3, 4)  # Equivalent to add(3, 4)
```

```

3. **Functions Can Be Returned as Results:**

Functions in R can also be returned as results from other functions. This is a key concept in functional programming, where functions can generate and return new functions.

```R
# Define a function that generates and returns a new function
create_multiplier <- function(factor) {
  return(function(x) {
    return(x * factor)
  })
}

# Create a new function that multiplies by 5
multiply_by_5 <- create_multiplier(5)

# Use the generated function
result <- multiply_by_5(7)  # Result: 35
```

4. **Functions Can Be Stored in Data Structures:**
```

You can store functions in data structures like lists, vectors, or data frames. This capability allows you to organize and manipulate functions programmatically.

```R
# Store functions in a list
function_list <- list(
  add = function(a, b) { return(a + b) },
  subtract = function(a, b) { return(a - b) }
)

# Access and use the functions from the list
result_add <- function_list$add(3, 4)       # Equivalent to add(3, 4)
result_subtract <- function_list$subtract(7, 2)  # Equivalent to subtract(7, 2)
```

In summary, in R, functions are indeed first-class objects because they can be assigned to variables, passed as arguments to other functions, returned as results from functions, and stored in data structures. This flexibility and capability to treat functions as data enable powerful programming techniques, such as functional programming and the creation of higher-order functions, making R a versatile language for a wide range of applications, including data analysis and statistical computing.

5B. How to pass default values for arguments in R.

Ans

In R, you can pass default values for function arguments by specifying those default values within the function definition. When a function is called and an argument is not provided by the caller, the default value you specify in the function definition will be used. Here's how you can pass default values for function arguments in R:

```R
# Define a function with default arguments
greet <- function(name = "Guest", time = "morning") {
  message("Good ", time, ", ", name, "!")
}

# Call the function with both arguments provided
greet("Alice", "evening")  # Output: Good evening, Alice!

# Call the function with only one argument provided
greet("Bob")  # Output: Good morning, Bob!

# Call the function with no arguments provided
greet()  # Output: Good morning, Guest!
```

In the `greet` function above, we have two arguments, `name` and `time`, and we have provided default values for both arguments in the function definition. Here's a breakdown of how the function behaves:

- When you call `greet("Alice", "evening")`, both `name` and `time` are provided, so the provided values are used.

- When you call `greet("Bob")`, only `name` is provided, so the default value for `time` ("morning") is used.

- When you call `greet()` without providing any arguments, both `name` and `time` use their default values ("Guest" and "morning" respectively).

By specifying default values for function arguments, you make the function more flexible, as callers can choose to provide their own values or rely on the defaults. This can be particularly useful when creating functions that have optional or configurable parameters.

6. Write R code to return a complex object.

Ans

In R, you can create and return complex objects, such as lists, data frames, or custom objects, from functions. Here's an example of how to write R code to return a complex object, specifically a list, from a function:

```R
# Define a function that returns a complex object (a list)
create_person <- function(name, age, gender) {
  person <- list(
    Name = name,
    Age = age,
    Gender = gender
```

```
  )
  return(person)
}

# Call the function to create a person object
person_object <- create_person("Alice", 30, "Female")

# Access and print the attributes of the person object
cat("Name:", person_object$Name, "\n")
cat("Age:", person_object$Age, "\n")
cat("Gender:", person_object$Gender, "\n")
```

In this example, we have a function called `create_person` that takes three arguments: `name`, `age`, and `gender`. Inside the function, we create a list called `person` with these attributes and their respective values. Then, we use the `return()` function to return the `person` list as the result of the function.

When we call `create_person("Alice", 30, "Female")`, it returns a list containing the person's information. We can access and print the attributes of the person object using the `$` operator.

You can create more complex objects, such as data frames or custom classes, in a similar manner. Just define the structure and attributes of the object

within the function and return it as the result. This allows you to encapsulate and organize data and functionality in a reusable and structured manner.

7. Explain looping over non vector sets with examples.

Ans

In R, looping over non-vector sets typically involves iterating through elements that are not stored in a vector-like data structure, such as lists, data frames, or other collections of objects. This kind of looping is often necessary when you need to perform operations on elements that are not arranged in a regular sequence like vectors or matrices. You can use various types of loops and control structures to iterate over these non-vector sets. Here, we'll explore some examples of looping over non-vector sets in R.

Example 1: Looping Over a List of Names

In this example, we have a list of names, and we want to print a greeting for each name.

```R
# Create a list of names
names_list <- list("Alice", "Bob", "Charlie", "David")

# Loop over the list and print greetings
for (name in names_list) {
  cat("Hello, ", name, "!\n")
```

}
```

**Output:**
```
Hello, Alice !
Hello, Bob !
Hello, Charlie !
Hello, David !
```

In this case, we used a `for` loop to iterate over the elements of the `names_list` (which is a non-vector set). The loop variable `name` takes on each name in the list one by one, and we print a greeting for each name.

**Example 2: Looping Over Rows in a Data Frame**

Suppose you have a data frame containing information about individuals, and you want to perform some operation on each row of the data frame.

```R
# Create a data frame
data_df <- data.frame(
  Name = c("Alice", "Bob", "Charlie", "David"),
  Age = c(30, 25, 35, 28),
  Gender = c("Female", "Male", "Male", "Male")
```
```

```
)

# Loop over the rows of the data frame and print information
for (i in 1:nrow(data_df)) {
  cat("Name:", data_df$Name[i], "\n")
  cat("Age:", data_df$Age[i], "\n")
  cat("Gender:", data_df$Gender[i], "\n")
  cat("\n")
}
```

Output:

```
Name: Alice
Age: 30
Gender: Female

Name: Bob
Age: 25
Gender: Male

Name: Charlie
Age: 35
Gender: Male

Name: David
```

Age: 28

Gender: Male

```

In this example, we used a `for` loop to iterate over the rows of the data frame `data_df`. We accessed the values in each row using indexing, and then we printed the information.

**Example 3: Looping Over a List of Custom Objects**

Suppose you have a list of custom objects, and you want to perform some operation on each object in the list.

```R
# Define a custom class for a person
Person <- function(name, age) {
  return(list(Name = name, Age = age))
}

# Create a list of person objects
people_list <- list(
  Person("Alice", 30),
  Person("Bob", 25),
  Person("Charlie", 35),
  Person("David", 28)
)

```
# Loop over the list of person objects and print information
for (person in people_list) {
  cat("Name:", person$Name, "\n")
  cat("Age:", person$Age, "\n")
  cat("\n")
}
```

Output:

```
Name: Alice
Age: 30

Name: Bob
Age: 25

Name: Charlie
Age: 35

Name: David
Age: 28
```

In this example, we created a custom class `Person` and defined a list of person objects. We then used a `for` loop to iterate over the list of custom objects and printed the information for each person.

These examples demonstrate how you can loop over non-vector sets, such as lists, data frames, or custom objects, using `for` loops or other control structures to perform operations on their elements. Depending on your specific data and requirements, you may need to choose the most appropriate looping technique and indexing method to access and process the elements of your non-vector sets.

8. Implement binary search tree with R

Ans

A binary search tree (BST) is a binary tree data structure where each node has at most two child nodes, referred to as the left child and the right child. In a BST, the left child contains values less than or equal to the parent node, and the right child contains values greater than the parent node. Implementing a binary search tree in R involves creating functions to insert nodes, search for values, and perform tree traversal operations like in-order, pre-order, and post-order traversal.

Here's an implementation of a binary search tree in R:

```R
# Define a structure for a binary tree node
createNode <- function(key) {
  return(list(
```

```
    key = key,
    left = NULL,
    right = NULL
  ))
}

# Define a function to insert a value into the BST
insert <- function(root, key) {
  if (is.null(root)) {
    return(createNode(key))
  }
  if (key < root$key) {
    root$left <- insert(root$left, key)
  } else if (key > root$key) {
    root$right <- insert(root$right, key)
  }
  return(root)
}

# Define a function to search for a value in the BST
search <- function(root, key) {
  if (is.null(root) || root$key == key) {
    return(root)
  }
  if (key < root$key) {
    return(search(root$left, key))
```

```
  }
  return(search(root$right, key))
}

# In-order traversal (prints values in ascending order)
inOrderTraversal <- function(root) {
  if (!is.null(root)) {
    inOrderTraversal(root$left)
    cat(root$key, " ")
    inOrderTraversal(root$right)
  }
}

# Create an empty BST and insert values
bst <- NULL
bst <- insert(bst, 50)
bst <- insert(bst, 30)
bst <- insert(bst, 20)
bst <- insert(bst, 40)
bst <- insert(bst, 70)
bst <- insert(bst, 60)
bst <- insert(bst, 80)

# Search for a value in the BST
result <- search(bst, 40)
if (!is.null(result)) {
```

```
  cat("Value 40 found in the BST.\n")
} else {
  cat("Value 40 not found in the BST.\n")
}

# Perform in-order traversal to print values in ascending order
cat("In-order traversal (sorted values): ")
inOrderTraversal(bst)
cat("\n")
```

This code defines a binary search tree with functions to insert nodes, search for values, and perform in-order traversal. You can create an empty BST (`bst`), insert values into it, search for values, and print the sorted values using the in-order traversal function.

Please note that this is a basic implementation, and in practice, you may want to add additional functionality, error handling, and balancing to ensure the tree remains efficient for various operations.

9. Explain the functioning of lapply() and tapply() in a R program with proper examples

Ans

`lapply()` and `tapply()` are two useful functions in R for applying a function to elements of a list or a vector and aggregating data by a factor, respectively. Let's explore how each of these functions works with examples:

1. `lapply()` Function:

`lapply()` stands for "list apply" and is used to apply a specified function to each element of a list or vector. It returns a list where each element is the result of applying the function to the corresponding element of the input list or vector.

Syntax:

```R
lapply(X, FUN, ...)
```

- `X`: The list or vector to apply the function to.
- `FUN`: The function to apply to each element of `X`.
- `...`: Additional arguments to pass to `FUN`.

Example: Applying `lapply()` to a List

Let's say we have a list of numbers and we want to square each number in the list using `lapply()`:

```R
# Create a list of numbers
numbers <- list(1, 2, 3, 4, 5)

# Define a function to square a number
square <- function(x) {
  return(x^2)
}

# Apply the square function to each element of the list using lapply()
squared_numbers <- lapply(numbers, square)

# Print the squared numbers
squared_numbers
```

Output:
```
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16

[[5]]
[1] 25
```

In this example, `lapply()` applies the `square()` function to each element of the `numbers` list, and the result is a list of squared numbers.

2. `tapply()` Function:

`tapply()` stands for "table apply" and is used to apply a function to subsets of a vector or data frame, grouped by one or more factors. It is commonly used for aggregating data based on factors.

Syntax:

```R
tapply(X, INDEX, FUN, ...)
```

- `X`: The vector or data frame to apply the function to.
- `INDEX`: A list of factors or a formula specifying the grouping.
- `FUN`: The function to apply to each subset.

- `...`: Additional arguments to pass to `FUN`.

Example: Using `tapply()` to Aggregate Data

Suppose we have a data frame containing scores of students by subject, and we want to calculate the average score for each subject using `tapply()`:

```R
# Create a data frame
scores_df <- data.frame(
  Subject = c("Math", "Math", "Science", "Science", "Math", "Science"),
  Score = c(90, 85, 78, 92, 88, 95)
)

# Use tapply() to calculate the average score by subject
avg_scores <- tapply(scores_df$Score, scores_df$Subject, mean)

# Print the average scores
avg_scores
```

Output:
```
   Math Science
   87.67    88.33
```

In this example, we used `tapply()` to calculate the average score for each subject in the `scores_df` data frame. The `INDEX` argument specifies the grouping by the "Subject" column, and the `mean` function is applied to calculate the average score for each subject.

These examples illustrate the functioning of `lapply()` and `tapply()` in R. `lapply()` is used for applying a function to each element of a list or vector, while `tapply()` is used for aggregating data by one or more factors. These functions are handy for various data manipulation tasks in R.

UNIT-3

1. **Describe linear algebra operations on vectors and matrices.**

   Ans

   Certainly, let's explore linear algebra operations on vectors and matrices with examples and outputs.

   1. Vector Operations:

   Vector Addition and Subtraction:

   - **Addition:** Given two vectors `u` and `v` of the same dimension, you can add them component-wise.

   Example:

   ```R
   u <- c(2, 4, 6)
   v <- c(1, 3, 5)
   result <- u + v
   result
   ```

   Output:

   ```
   [1] 3 7 11
   ```

   - **Subtraction:** Vector subtraction is similar to addition.

   Example:

   ```R
   result <- u - v
   result
   ```

   Output:

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

Scalar Multiplication:
- You can multiply a vector by a scalar.

Example:
```R
a <- 2
result <- a * u
result
```

Output:
```
[1]  4  8 12
```

Dot Product (Inner Product):
- The dot product of two vectors `u` and `v` is the sum of the products of their corresponding components.

Example:
```R
dot_product <- sum(u * v)
dot_product
```

Output:
```
[1] 44
```

Norm (Magnitude) of a Vector:
- The norm of a vector `u` represents its length or magnitude.

Example:
```R
norm_u <- sqrt(sum(u^2))
norm_u
```

Output:
```
[1] 7.483315
```

2. Matrix Operations:

Matrix Addition and Subtraction:
- Matrix addition and subtraction are performed element-wise.

Example:
```R
A <- matrix(1:6, nrow = 2)
B <- matrix(7:12, nrow = 2)
result_addition <- A + B
result_subtraction <- A - B
result_addition
result_subtraction
```

Output (Addition):
```
  [,1] [,2] [,3]
[1,]    8   10   12
[2,]   10   12   14
```

Output (Subtraction):
```
```

```
     [,1] [,2] [,3]
[1,]   -6   -6   -6
[2,]   -6   -6   -6
```

Scalar Multiplication of a Matrix:
- You can multiply a matrix by a scalar.

Example:
```R
c <- 3
result_multiplication <- c * A
result_multiplication
```

Output:
```
     [,1] [,2] [,3]
[1,]    3    6    9
[2,]   12   15   18
```

Matrix Multiplication (Dot Product):
- Matrix multiplication is performed by taking the dot product of rows from the first matrix and columns from the second matrix.

Example:
```R
C <- matrix(c(1, 2, 3, 4), nrow = 2)
D <- matrix(c(5, 6, 7, 8), nrow = 2)
result_multiplication <- C %*% D
result_multiplication
```

Output:
```
```

```
     [,1] [,2]
[1,]   19   22
[2,]   43   50
```

Matrix Transposition:
- Matrix transposition involves interchanging rows and columns.

Example:
```R
E <- matrix(1:6, nrow = 2)
result_transpose <- t(E)
result_transpose
```

Output:
```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    5    6
```

These are some common linear algebra operations on vectors and matrices in R, along with examples and outputs. Linear algebra operations are fundamental in various fields, including mathematics, physics, engineering, and data analysis.

2. **What is cumulative sum,product,min,max? Explain with an example? Write R functions used for this purpose?**

Ans

Cumulative sum, cumulative product, minimum (min), and maximum (max) are common operations applied to sequences of numbers, such as vectors or lists. These operations provide useful insights into the data and can help in various analytical and statistical tasks. Let's explore each of these operations with examples and provide R functions for each purpose.

1. Cumulative Sum (`cumsum()`):

The cumulative sum of a sequence of numbers is a new sequence where each element is the sum of all the preceding elements in the original sequence.

R Function: `cumsum()`

Example:
```R
# Original vector
numbers <- c(1, 2, 3, 4, 5)

# Calculate the cumulative sum
cumulative_sum <- cumsum(numbers)
cumulative_sum
```

Output:
```
[1]  1  3  6 10 15
```

In this example, the cumulative sum of the vector `[1, 2, 3, 4, 5]` is `[1, 3, 6, 10, 15]`. Each element in the cumulative sum is the sum of all previous elements in the original vector.

2. Cumulative Product (`cumprod()`):

The cumulative product of a sequence of numbers is a new sequence where each element is the product of all the preceding elements in the original sequence.

R Function: `cumprod()`

Example:
```R
# Original vector
```

```
numbers <- c(1, 2, 3, 4, 5)

# Calculate the cumulative product
cumulative_product <- cumprod(numbers)
cumulative_product
```

Output:
```
[1]   1   2   6  24 120
```

In this example, the cumulative product of the vector `[1, 2, 3, 4, 5]` is `[1, 2, 6, 24, 120]`. Each element in the cumulative product is the product of all previous elements in the original vector.

3. Minimum (`min()`):

The minimum function returns the smallest value among a sequence of numbers.

R Function: `min()`

Example:
```R
# Original vector
numbers <- c(5, 2, 9, 1, 8)

# Find the minimum value
minimum_value <- min(numbers)
minimum_value
```

Output:
```
[1] 1
```

In this example, the minimum value among the numbers `[5, 2, 9, 1, 8]` is `1`.

4. Maximum (`max()`):

The maximum function returns the largest value among a sequence of numbers.

R Function: `max()`

Example:
```R
# Original vector
numbers <- c(5, 2, 9, 1, 8)

# Find the maximum value
maximum_value <- max(numbers)
maximum_value
```

Output:
```
[1] 9
```

In this example, the maximum value among the numbers `[5, 2, 9, 1, 8]` is `9`.

These operations and their corresponding R functions are valuable for data analysis and manipulation tasks. They allow you to summarize data, calculate running totals, and find extremum values within a sequence of numbers.

3. **Explain about sort(), order() and rank() functions with examples.**

Ans

In R, the `sort()`, `order()`, and `rank()` functions are used for sorting data, obtaining the order of elements, and ranking elements within a vector or data frame. Let's explore each of these functions with examples:

1. `sort()` Function:

The `sort()` function is used to sort the elements of a vector or data frame in ascending or descending order. By default, it sorts in ascending order, but you can specify `decreasing = TRUE` to sort in descending order.

Syntax:

```R
sort(x, decreasing = FALSE)
```

- `x`: The vector or data frame to be sorted.
- `decreasing`: A logical value indicating whether to sort in descending order (default is `FALSE`).

Example: Sorting a Vector in Ascending Order

```R
# Original vector
numbers <- c(5, 2, 9, 1, 8)

# Sort the vector in ascending order
sorted_numbers <- sort(numbers)
sorted_numbers
```

Output:

```
[1] 1 2 5 8 9
```

Example: Sorting a Vector in Descending Order

```R
# Sort the vector in descending order
sorted_numbers_desc <- sort(numbers, decreasing = TRUE)
sorted_numbers_desc
```

Output:

```
[1] 9 8 5 2 1
```

2. `order()` Function:

The `order()` function returns a permutation that rearranges its input into ascending or descending order. It's often used to obtain the order of elements that would sort a vector.

Syntax:

```R
order(..., na.last = NA, decreasing = FALSE)
```

- `...`: A sequence of numeric, complex, character, or logical vectors, or a data frame.
- `na.last`: A logical value indicating whether to treat missing values as greater (default is `NA`).
- `decreasing`: A logical value indicating whether to sort in descending order (default is `FALSE`).

Example: Getting the Order of Elements in a Vector

```R
# Original vector
numbers <- c(5, 2, 9, 1, 8)

# Get the order of elements
order_indices <- order(numbers)
order_indices
```

Output:

```
[1] 4 2 1 5 3
```

In this example, `order_indices` contains the indices that would rearrange the `numbers` vector in ascending order.

3. `rank()` Function:

The `rank()` function computes the sample ranks of the elements in a vector, indicating the positions of elements when they are sorted in ascending order. Ties can be handled using various methods.

Syntax:

```R
rank(x, na.last = "keep", ties.method = "average")
```

- `x`: The vector or data frame for which ranks are computed.
- `na.last`: A character string indicating how missing values should be treated. Options are "keep," "last," or "first."
- `ties.method`: A character string indicating how ties should be handled. Options are "average," "min," "max," "first," and "random."

Example: Computing Ranks of Elements in a Vector

```R
# Original vector with ties
scores <- c(85, 90, 75, 85, 80)
```

```
# Compute ranks with tie-breaking method "average"
ranked_scores <- rank(scores, ties.method = "average")
ranked_scores
```

Output:

```
[1] 3.5 5.0 1.0 3.5 2.0
```

In this example, the `ranked_scores` vector contains the ranks of the elements in the `scores` vector, with tie-breaking based on the "average" method.

These functions (`sort()`, `order()`, and `rank()`) are essential for data manipulation and analysis in R, as they allow you to arrange data in a meaningful way, find order information, and rank elements based on their values.

4. Develop a function to find the cross product

Ans

The cross product is an operation between two vectors in three-dimensional space, resulting in a third vector that is orthogonal (perpendicular) to the plane formed by the original vectors. The cross product is defined as:

$$\mathbf{a} \times \mathbf{b} = \left(a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1 \right)$$

In R, you can create a function to calculate the cross product of two vectors using their components. Here's a function to find the cross product:

```R
cross_product <- function(a, b) {
  if (length(a) != 3 || length(b) != 3) {
    stop("Both input vectors must have exactly 3 components.")
  }

  result <- c(
    a[2] * b[3] - a[3] * b[2],
    a[3] * b[1] - a[1] * b[3],
    a[1] * b[2] - a[2] * b[1]
  )

  return(result)
}
```

You can use this `cross_product` function by passing two vectors with three components each. It calculates and returns the cross product as a vector.

Example: Calculating the Cross Product of Two Vectors

```R
```

```
# Define two 3D vectors
vector_a <- c(1, 2, 3)
vector_b <- c(4, 5, 6)

# Calculate the cross product
result_cross_product <- cross_product(vector_a, vector_b)
result_cross_product
```

Output:

```
[1] -3  6 -3
```

In this example, the `cross_product` function is used to calculate the cross product of `vector_a` and `vector_b`, resulting in the vector `[-3, 6, -3]`.**Explain any six math functions in R with proper example code**

Ans

Certainly! R provides a wide range of mathematical functions that allow you to perform various mathematical operations. Here are six commonly used math functions in R with examples:

1. `sqrt()`: Square Root Function

The `sqrt()` function is used to calculate the square root of a number.

Example:

```R
# Calculate the square root of a number
x <- 16
sqrt_x <- sqrt(x)
sqrt_x
```

Output:
```
[1] 4
```

2. `abs()`: Absolute Value Function

The `abs()` function returns the absolute value of a number.

Example:

```R
# Calculate the absolute value of a number
y <- -5
abs_y <- abs(y)
abs_y
```

Output:

```
[1] 5
```

3. `exp()`: Exponential Function

The `exp()` function calculates the exponential value of a number.

Example:

```R
# Calculate e raised to the power of a number
z <- 2
exp_z <- exp(z)
exp_z
```

Output:

```
[1] 7.389056
```

4. `log()`: Natural Logarithm Function

The `log()` function calculates the natural logarithm of a number.

Example:

```R
# Calculate the natural logarithm of a number
w <- 10
log_w <- log(w)
log_w
```

Output:

```
[1] 2.302585
```

5. `sin()`: Sine Function

The `sin()` function calculates the sine of an angle in radians.

Example:

```R
# Calculate the sine of an angle in radians
angle <- pi/4
sin_angle <- sin(angle)
sin_angle
```

```

**Output:**

```
[1] 0.7071068
```

**6. `cos()`: Cosine Function**

The `cos()` function calculates the cosine of an angle in radians.

**Example:**

```R
# Calculate the cosine of an angle in radians
angle <- pi/3
cos_angle <- cos(angle)
cos_angle
```

Output:

```
[1] 0.5
```
```

These are just a few examples of the mathematical functions available in R. R provides a comprehensive set of mathematical functions for a wide range of calculations, making it a powerful tool for data analysis and scientific computing.