

Unit – I

INTRODUCTION TO R PROGRAMMING

1. What is R? Briefly describe the history and development of the R programming language.

The **R programming language** is used for statistical analysis, data visualization, and data science. It's popular among researchers, data scientists, and statisticians for its powerful tools and packages.

The R Language stands out as a powerful tool in the modern era of statistical computing and data analysis. Widely embraced by statisticians, data scientists, and researchers, the R Language offers an extensive suite of packages and libraries tailored for data manipulation, statistical modelling, and visualization. In this article, we explore the features, benefits, and applications of the R Programming Language, shedding light on why it has become an indispensable asset for data-driven professionals across various industries.

R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. Moreover, the project was conceived in 1992, with an initial version released in 1995 and a stable beta version in 2000.

The R language came to use quite a bit after S had been developed. One key limitation of the S language was that it was only available in a commercial package, S-PLUS. In 1991, R was created by Ross Ihaka and Robert Gentleman in the Department of Statistics at the University of Auckland. In 1993 the first announcement of R was made to the public.

In 1995, Martin Mächler made an important contribution by convincing Ross and Robert to use the GNU General Public License to make R free software. This was critical because it allowed for the source code for the entire R system to be accessible to anyone who wanted to tinker with it (more on free software later).

In 1996, a public mailing list was created (the R-help and R-devel lists) and in 1997 the R Core Group was formed, containing some people associated with S and S-PLUS. Currently, the core group controls the source code for R and is solely able to check in changes to the main R source tree. Finally, in 2000 R version 1.0.0 was released to the public.

2.a) Explain the different ways to run R code, including using the R console, R scripts, and R Markdown.

R operates in two modes: interactive and batch. The one typically used is interactive mode. In this mode, you type in commands, R displays results, you type in more commands, and so on. On the other hand, batch mode does not require interaction with the user. It's useful for

production jobs, such as when a program must be run periodically, say once per day, because you can automate the process.

Interactive Mode

On a Linux or Mac system, start an R session by typing R on the command line in a terminal window. On a Windows machine, start R by clicking the R icon. The result is a greeting and the R prompt, which is the > sign. The screen will look something like this:

```
R version 2.10.0 (2009-10-26)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

You can then execute R commands. The window in which all this appears is called the R console.

Example:

```
> mean(abs(rnorm(100)))
[1] 0.7194236
```

Batch Mode

Sometimes it's convenient to automate R sessions. For example, you may wish to run an R script that generates a graph without needing to bother with manually launching R and executing the script yourself. Here you would run R in batch mode. As an example, let's put our graph-making code into a file named z. R with the following contents:

```
pdf("xh.pdf") # set graphical output file
hist(rnorm(100)) # generate 100 N(0,1) variates and plot their histogram
dev.off() # close the graphical output file
```

The items marked with # are comments. They're ignored by the R interpreter. Comments serve as notes to remind us and others what the code is doing, in a human-readable format.

We could run this code automatically, without entering R's interactive mode, by invoking R with an operating system shell command (such as at the \$ prompt commonly used in Linux systems)

2.b) Classify the advantages and disadvantages of each method for running R code.

Advantages of Script mode:

Script mode is a great way to automate tasks and run commands on a remote server. Script mode can also be used to create files that will execute certain commands when they are run. This can be very helpful in automating tasks or setting up a development environment.

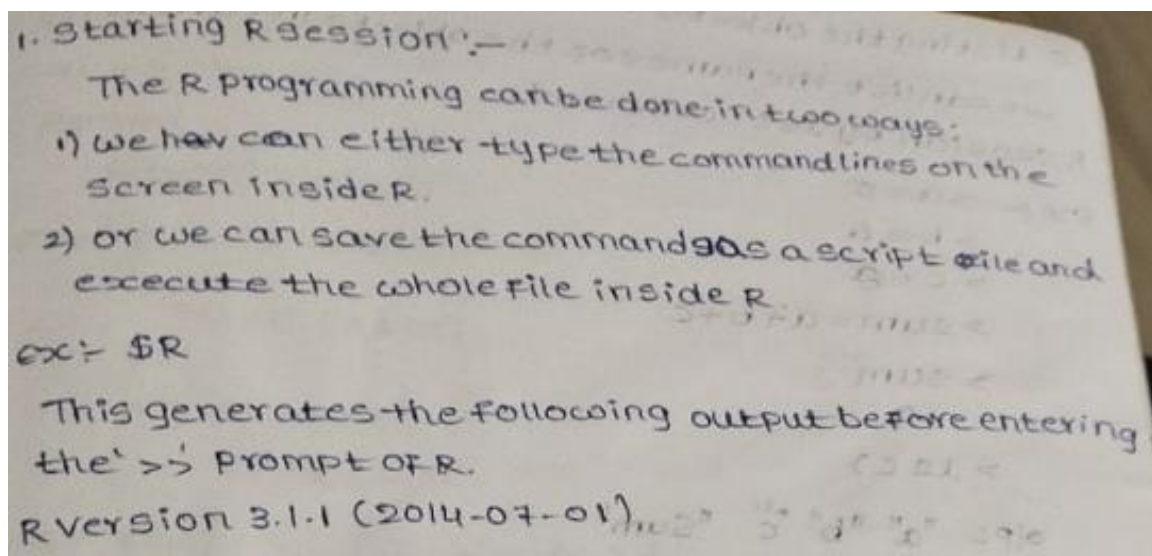
Disadvantages of script mode:

There are some disadvantages to using script mode, however. First, if something goes wrong with the script, it can be difficult to troubleshoot. Second, if you are not familiar with scripting languages, it can be difficult to write a script that does what you want it to do. Finally, scripts can be slow because they have to execute all of their commands serially.

Advantages and Disadvantages of interactive mode:

The interactive mode is great for testing out commands and getting immediate feedback. It can also be used to quickly execute commands on a remote server. The main disadvantage of interactive mode is that it can be difficult to automate tasks. For larger programs, the interactive mode is not suitable. Moreover, editing the program in interactive mode is a tedious task.

3.a) What are R sessions, and how do you manage them effectively?



2. working with R session :-

once we are inside the R session, we can directly execute R language commands by type them line by line.

```
ex:-> a = 5
```

```
> b = 6
```

```
> c = a + b
```

```
c
```

The value of the variable 'c' is printed as

11.

3. Get help inside R session :-

To get help on any function of R, type help in R prompt. For example, if we need help on if logic type

```
help("if")
```

then, help line for the if statement is printed.

4. Exit the R session :-

to exit the R session, type `quit()` in R prompt and say n(no) for saving the workspace image.

```
> quit()
```

```
> save workspace image? [y/n] : n
```

5. Listing the objects in current session
we can list the names of the objects in current session by ls() command

```
ex:- >a=5  
>b=6  
>c=8  
>sum=a+b+c  
>sum  
[1] 19  
>ls()
```

```
olp:- "a" "b" "c" "sum"
```

b. Removing the objects from the current session

Specific objects created in the current session can be removed using rm() command.

If we specify the name of the object, it will be removed.

```
>a=5  
>b=6  
>c=8  
>sum=a+b+c  
>sum
```

```
19  
>ls()  
"a" "b" "c" "sum"
```

```
>rm(list=c(sum))  
>ls()
```

```
"a" "b" "c"
```

3.b) Illustrate the concept of functions in R with a program, and how to define and use them?

A function is a block or chunk of code having a specific structure, which is often singular or atomic nature, and can be reused to accomplish a specific nature. A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

structure of a function

```
function_name <- function(arguments)

{
statements
}
```

- The word `_function` is a keyword which is used to specify the statements enclosed within the curly braces are part of the function.
- `function_name` is used to identify the function
- Function consists of formal arguments and body
- The function is called using the following statement: `function_name(arguments)`.

Example:

```
# function to add 2 numbers
add_num <- function(a,b)
{
  sum_result <- a+b
  return(sum_result)
}

# calling add_num function
sum = add_num(35,34)

#printing result
print(sum)
```

Default Arguments:-

R also makes frequent use of default arguments. Consider a function definition like this:

```
> g <- function(x,y=2,z=T) { ... }
```

Here y will be initialized to 2 if the programmer does not specify y in the call. Similarly, z will have the default value TRUE.

4.a) Demonstrate basic math operations with program, such as addition, subtraction, multiplication, and division.

Addition:

The values at the corresponding positions of both operands are added.

Ex: a <- c (1, 2)

b <- c (2, 4)

print (a+b)

output:

3 6

Subtraction:

The second operand values are subtracted from the first.

Ex: a <- 6

b <- 8.4

print (a-b)

output:

-2.4

Multiplication:

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the '*' operator.

Ex: B= c (4,4)

C= c (5,5)

print (B*C)

output:

20 20

Division:

The first operand is divided by the second operand with the use of the '/' operator.

Ex: a <- 10

b <- 5

print (a/b)

output:

2

4.b) Categorize the order of operations in R and how to use parentheses to control the evaluation of expressions with an example program.

Operator	Description	Associativity
^	Exponent	Right to Left
-x, +x	Unary minus, Unary plus	Left to Right
%%	Modulus	Left to Right
*, /	Multiplication, Division	Left to Right
+, -	Addition, Subtraction	Left to Right
, =, ==, !=	Comparisons	Left to Right
!	Logical NOT	Left to Right
&, &&	Logical AND	Left to Right
,	Logical OR	Left to Right

->, ->>	Rightward assignment	Left to Right
<<-	Leftward assignment	Right to Left
=	Leftward assignment	Right to Left

Ex: $(2 + 6) * 5$

Here, the `*` operator gets higher priority than `+` and hence $2 + 6 * 5$ is interpreted as $2 + (6 * 5)$. This order can be changed with the use of parentheses `()`.

5.b) Discuss the different naming conventions and rules for variable naming, each with example in R.

Variable:

A variable is used to store data that can be accessed later by subsequent code. In R, there are no variable “declaration” commands. Instead, they are created with the assignment operator.

```
variable <- "value"
```

The assignment operator can be chained together to initialize multiple variables at once:

```
var1<- var2 <- var3 <- 0
```

R variables must adhere to the following naming conventions:

- The name can be a combination of letters, digits, period(.) and underscore (_).
- It must start with a letter or a period.
- If it starts with a period, the second character cannot be a number.
- It cannot start with a number or an underscore.
- Variable names are case-sensitive.
- Reserved words (TRUE, FALSE, print, etc.) cannot be used as variable names.

Example:

```
foo <- 1
```

```
foo <- 2 # A different variable from "foo" above
```

```
.bar <- TRUE
```

```
foo_bar <- "value"
```

```
3.5 -> Foo123.4 # Rightward assignment.
```

6.a) Classify different data types in R with example.

Datatype:

R Data types are used to specify the kind of data that can be stored in a variable.

1. numeric
2. Integer
3. logical
4. complex
5. character

1. Numeric:

Decimal values are called numeric in R. It is the default R data type for numbers in R.

If you assign a decimal value to a variable x as follows, x will be of numeric type.

Example:

```
x = 5.6
```

```
print(class(x))
```

```
print ( type of(x))
```

output:

```
"numeric"
```

```
"double"
```

2.Integer:

You can create as well as convert a value into an integer type using the **as.integer()** function.

You can also use the capital 'L' notation as a suffix to denote that a particular value is of the integer R data type.

Example:

```
x = as.integer(5)
```

```
print(class(x))
```

```
y = 5L
```

```
print(typeof(y))
```

output:

```
“integer”
```

```
“integer”
```

3.Logical:

Boolean values, which have two possible values, are represented by this R data type: FALSE or TRUE.

Example:

```
x = 4
```

```
y = 3
```

```
z = x > y
```

```
print(z)
```

```
print(typeof(z))
```

output:

```
TRUE
```

```
“logical”
```

4.Complex:

R supports complex data types that are set of all the complex numbers. The complex data type is to store numbers with an imaginary component.

Example:

```
x = 4 + 3i
```

```
print(class(x))
```

output:

```
“complex”
```

5.Character:

R supports character data types where you have all the alphabets and special characters. It stores character values or strings. Strings in R can contain alphabets, numbers, and symbols.

Example:

```
char = "Geeksforgeeks"  
print(type of(char))
```

output:

```
“character”
```

6.b) Conclude how to check the data type of a variable and convert between different data types.

Data Types in R:

It is useful to know the data type in order to know what functions can be performed on the object. To determine the type of data, you can use the `class()`, `mode()` or `typeof()` functions. The following commands create different variables and check their type using the `class()` function.

Datatype Conversion:

Data Type conversion is the process of converting one type of data to another type of data.

as. datatype(value)

Example:

```
age <- 20  
pwd <- FALSE  
# Converting type  
as.character(age)  
as.character(pwd)
```

output:

```
“20”  
“FALSE”
```

7. Classify different data structures in R with examples.

Data structures are a specific way of organizing data in a specialized format on a computer so that the information can be organized, processed, stored, and retrieved quickly and effectively.

The most essential data structures used in R include:

- Vectors
- Lists
- Data frames
- Matrices
- Arrays

Vectors

A vector is an ordered collection of basic data types of a given length. The only key thing here is all the elements of a vector must be of the identical data type e.g homogeneous data structures. Vectors are one-dimensional data structures.

Example:

```
X = c(1, 3, 5, 7, 8)
```

```
Print(x)
```

Output:

```
1 3 5 7 8
```

Lists:

A list is a generic object consisting of an ordered collection of objects. Lists are heterogeneous data structures. These are also one-dimensional data structures. A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.

Example:

```
EmpId = c(1, 2, 3, 4)
```

```
EmpName = c("Debi", "Sandeep", "Subham", "Shiba")
```

```
numberOfEmp = 4
```

```
EmpList = list(EmpId, EmpName, numberOfEmp)
```

```
print(EmpList)
```

output:

```
[1]
```

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

```
[[2]]
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
[[3]]
```

```
[1] 4
```

Data Frames:

Data frames are generic data objects of R which are used to store the tabular data. Data frames are the foremost popular data objects in R programming because we are comfortable in seeing the data within the tabular form. They are two-dimensional, heterogeneous data structures. These are lists of vectors of equal lengths.

Data frames have the following constraints placed upon them:

- A data-frame must have column names and every row should have a unique name.
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.

To create a data frame we use the `data.frame()` function.

Example:

```
Name = c("Amiya", "Raj", "Asish")
```

```
Language = c("R", "Python", "Java")
```

```
df = data.frame(Name, Language, Age)
```

```
print(df)
```

output:

	Name	Language	Age
1	Amiya	R	22
2	Raj	Python	25
3	Asish	Java	45

Matrices:

A matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. Matrices are two-dimensional, homogeneous data structures.

Example:

```
A = matrix(
```

```
  # Taking sequence of elements
```

```
  c(1, 2, 3, 4, 5, 6, 7, 8, 9),
```

```
  nrow = 3, ncol = 3,
```

```
byrow = TRUE
)
```

```
print(A)
```

output:

```
  [,1] [,2] [,3]
[1,]   1   2   3
[2,]   4   5   6
[3,]   7   8   9
```

Arrays:

Arrays are the R data objects which store the data in more than two dimensions. Arrays are n-dimensional data structures. For example, if we create an array of dimensions (2, 3, 3) then it creates 3 rectangular matrices each with 2 rows and 3 columns. They are homogeneous data structures.

Example:

```
A= array(
  # Taking sequence of elements
  c(1, 2, 3, 4, 5, 6, 7, 8),

  # Creating two rectangular matrices
  # each with two rows and two columns
  dim = c(2, 2, 2)
)
print(A)
```

output:

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

, , 2
  [,1] [,2]
[1,]   5   7
[2,]   6   8
```

8.a) i) Create a program for manipulating data frames in R.

```
# R program to rename a Data Frame
# Adding Package
df <- library(plyr)
# Creating a Data Frame
df<-data.frame(row1 = 0:2, row2 = 3:5, row3 = 6:8)
print("Original Data Frame")
print(df)
print("Modified Data Frame")
# Renaming Data Frame
rename(df, c("row1"="one", "row2"="two", "row3"="three"))
```

output:

```
[1] "Original Data Frame"
```

```
  row1 row2 row3
1    0    3    6
2    1    4    7
3    2    5    8
```

```
[1] "Modified Data Frame"
```

```
  one two three
1    0    3    6
2    1    4    7
3    2    5    8
```

ii. Create a program for manipulating lists in R.

```
Vec<-c(3,4,5,6)
Char_vec<-("Shubham", " nishika", "gunjan", "sumit")
logic_vec<-c(TRUE,FALSE,FALSE,TRUE)
out_list<-list(vec,char_vec,logic_vec)
out_list
```


output:

```
[[1]]
```

```
[1] 3 4 5 6
```

```
[[2]]
```

```
[1] "shubham" "nishka" "gunjan" "sumit"
```

```
[[3]]
```

```
[1] TRUE FALSE FALSE TRUE
```

8.b) Determine Vector and its functions with examples.

Vectors:

A vector is an ordered collection of basic data types of a given length. The only key thing here is all the elements of a vector must be of the identical data type e.g homogeneous data structures. Vectors are one-dimensional data structures.

Example:

```
X = c(1, 3, 5, 7, 8)
```

```
Print(x)
```

Output:

```
1 3 5 7 8
```

Using the seq() function

In R, we can create a vector with the help of the seq() function. A sequence function creates a sequence of elements as a vector. The seq() function is used in two ways, i.e., by setting step size with 'by' parameter or specifying the length of the vector with the 'length.out' feature.

Example:

1. seq_vec<-seq(1,4,by=0.5)
2. seq_vec
3. class(seq_vec)

Output

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

9.a) Differentiate between matrices and data frames?

Matrix in R:

It's a homogeneous collection of data sets which is arranged in a two dimensional rectangular organisation. It's a $m \times n$ array with similar data type. It is created using a vector input. It has a fixed number of rows and columns. You can perform many arithmetic operations on R matrix like – addition, subtraction, multiplication, and divisions.

Example:

```
A = matrix (c(11, 22, 33, 44, 55, 66),
```

```
        nrow = 2, ncol = 3, byrow = 1)
```

```
# Printing Matrix
```

```
print(A)
```

output:

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

```
[1,] 11 22 33
```

```
[2,] 44 55 66
```

Data Frames in R:

It is used for storing data tables. It can contain multiple data types in multiple columns called fields. It is a list of vector of equal length. It is a generalized form of a matrix. It is like a table in excel sheets. It has column and row names. The name of rows are unique with no empty columns. The data stored must be numeric, character or factor type. Data Frames are heterogeneous.

Example:

```
Name = c("Amiya", "Raj", "Asish")
```

```
Language = c("R", "Python", "Java")
```

```
df = data.frame(Name, Language, Age)
```

```
print(df)
```

output:

```
  Name Language Age
```

```
1 Amiya      R  22
```

```
2  Raj Python  25
```

```
3 Asish  Java  45
```

9.b) Create a program for 2x3 matrix and accessing its second row?

```
row_names = c("row1", "row2", "row3", "row4")
col_names = c("col1", "col2", "col3", "col4")
M = matrix(c(1:16), nrow = 4, byrow = TRUE, dimnames = list((row_names,col_names))
Print ("Original Matrix:")
print(M)
print ("Access only the 2nd row:")
print (M [3,])
```

10.a) Discriminate the following data structures with syntax and example.

i)Arrays

ii)Matrices

Arrays:

Arrays are the R data objects which store the data in more than two dimensions. Arrays are n-dimensional data structures. For example, if we create an array of dimensions (2, 3, 3) then it creates 3 rectangular matrices each with 2 rows and 3 columns. They are homogeneous data structures.

Example:

```
A=array(
  # Taking sequence of elements
  C (1, 2, 3, 4, 5, 6, 7, 8),
  # Creating two rectangular matrices
  # each with two rows and two columns
  dim = c (2, 2, 2)
)
print(A)
output:
, 1
[1] [2]
```

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

```
      [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Matrices:

A matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. Matrices are two-dimensional, homogeneous data structures.

Example:

```
A = matrix(
  # Taking sequence of elements
  c(1, 2, 3, 4, 5, 6, 7, 8, 9),
  nrow = 3, ncol = 3,
  byrow = TRUE
)
print(A)
```

output:

```
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

10.b) Demonstrate how to create, manipulate, and perform operations on these data structures.

i)Lists

ii)Data Frames

Lists:

A list is a generic object consisting of an ordered collection of objects. Lists are heterogeneous data structures. These are also one-dimensional data structures. A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.

Example:

```
EmpId = c(1, 2, 3, 4)
EmpName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4
EmpList = list(EmpId, EmpName, numberOfEmp)
print(EmpList)
```

output:

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

[[2]]
[1] "Debi" "Sandeep" "Subham" "Shiba"

[[3]]
[1] 4
```

Data Frames:

Data frames are generic data objects of R which are used to store the tabular data. Data frames are the foremost popular data objects in R programming because we are comfortable in seeing the data within the tabular form. They are two-dimensional, heterogeneous data structures. These are lists of vectors of equal lengths.

Data frames have the following constraints placed upon them:

- A data-frame must have column names and every row should have a unique name.
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.

To create a data frame we use the `data.frame()` function.

Example:

```
Name = c("Amiya", "Raj", "Asish")
```

```
Language = c("R", "Python", "Java")  
df = data.frame(Name, Language, Age)  
print(df)
```

output:

	Name	Language	Age
1	Amiya	R	22
2	Raj	Python	25
3	Asish	Java	45

Unit – II

R PROGRAMMING STRUCTURES, OPERATORS AND FUNCTIONS

1. Illustrate different conditional statements in R with appropriate syntax and examples.

Conditional control structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

- i. If
- ii. If-else
- iii. If-else ladder
- iv. Nested if-else
- v. Switch

If:

An "if statement" is written with the if keyword, and it is used to specify a block of code to be executed if a condition is TRUE.

Syntax:

```
if(condition)
{
    Statement
}
```

Example: Check for Positive Number

```
a = 10
if(a > 0)
{
    print("Positive Number")
}
```

Output:

“Positive Number”

If-else:

It is similar to if condition but when the test expression in if condition fails, then statements in else condition are executed.

Syntax:

```
if(condition) {  
    Statement1  
}  
else {  
    Statement 2  
}
```

Example: Check for Odd or Even Number

```
a = 10  
if(a %% 2 == 0)  
{  
    print("Even number")  
}  
else {  
    print("Odd number")  
}
```

Output:

"Even number"

If-Else if Ladder:

If-Else statements can be chained using If-Else if Ladder.

Syntax:

```
if(condition)  
{  
    Statement1  
}  
else  
if {  
    Statement 2  
}  
else {  
    Statement 3  
}
```

Example:

```
a <- 67
```



```

b <- 76
c <- 99
if(a > b && b > c)
{
  print("condition a > b > c is TRUE")
} else if(a < b && b > c)
{
  print("condition a < b > c is TRUE")
} else if(a < b && b < c)
{
  print("condition a < b < c is TRUE")
}

```

Output:

"condition a < b < c is TRUE"

Nested if-else:

In R, nested if-else statements allow you to evaluate multiple conditions sequentially within each other. This structure is useful when you have multiple criteria to evaluate and different actions to take based on those criteria.

Syntax:

```

if(condition) {
  if(condition 1) {
    statement
  } else {
    statement
  }
} else {
  if(child condition 2 is true) {
    execute this statement
  } else {
    execute this statement
  }
}

```

```
}
```

Example:

```
a <- 10
```

```
b <- 11
```

```
if(a == 10)
```

```
{
```

```
  if(b == 10)
```

```
  {
```

```
    print("a:10 b:10")
```

```
  } else
```

```
  {
```

```
    print("a:10 b:11")
```

```
  }
```

```
} else
```

```
{
```

```
  if(a == 11)
```

```
  {
```

```
    print("a:11 b:10")
```

```
  } else
```

```
  {
```

```
    print("a:11 b:11")
```

```
  }
```

Output:

```
"a:10 b:11"
```

Switch:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

```
switch (expression, case1, case2, case3,...,case n )
```

Example:

```
x <- switch(2, "CSE", "IT", "ECE")  
print(x)
```

Output:

"IT"

2.a) Explain working of switch case in R with an example program.

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. In R, the switch function is used to handle multiple conditions by selecting one of several possible actions based on the value of a variable.

Syntax:

```
switch (expression, case1, case2, case3, ..., case n)
```

Example:

```
val1 = as.integer(readline(prompt = "Enter a number"))  
val2 = as.integer(readline(prompt = "Enter a number"))  
val3 = readline(prompt="Enter any operation i.add\nii.sub\niii.div\niv.mul\nv.mod\nvi.pow")  
result = switch(  
  val3,  
  "add"= cat("Addition =", val1 + val2),  
  "sub"= cat("Subtraction =", val1 - val2),  
  "div"= cat("Division = ", val1 / val2),  
  "mul"= cat("Multiplication =", val1 * val2),  
  "mod"= cat("Modulus =", val1 %% val2),  
  "pow"= cat("Power =", val1 ^ val2)  
)  
cat(result)
```

Output:

Enter a number4

Enter a number4

Enter any operation i.add

ii.sub

iii.div

iv.mul

v.mod

vi.pow

add

Addition = 8

2.b) Develop a R program to check for leap year or not.

```
year = as.integer(readline(prompt = "Enter a year:"))
```

```
if((year %% 4) == 0){
```

```
  if((year %% 100) == 0){
```

```
    if((year %% 400) == 0){
```

```
      print(paste(year, "is a leap year"))
```

```
    }else{
```

```
      print(paste(year, "is not a leap year"))
```

```
    }
```

```
  }else{
```

```
    print(paste(year, "is a leap year"))
```

```
  }
```

```
}else{
```

```
  print(paste(year,"is not a leap year"))
```

```
}
```

Output:

Enter a year:2004

"2004 is a leap year"

3.a) Illustrate for loop in R and demonstrate its usage.

For loop:

A For loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
for(value in vector)
```

```
{
```

```
    Statements
```

```
}
```

Example:

```
a <- c(6,4,3,2,7)
```

```
for(i in a)
```

```
{
```

```
    print(i)
```

```
}
```

Output:

6

4

3

2

7

Usage of *for* loop:

Efficiency: for loops can be slower for large datasets. Consider using vectorized operations or the apply family of functions (lapply, sapply, etc.) for better performance.

Readability: Keep loops simple. For complex tasks, consider breaking them into smaller functions to enhance readability.

Pre allocate Storage: When building up a result, preallocate storage for objects that grow in each iteration to avoid inefficient memory allocation.

3.b) Explain the while loop and create a program for finding the sum of natural numbers.

While loop:

The While loop executes the same code again and again until as top condition is met.

Syntax:

```
while(condition)
{
    Statement
}
```

Program:

```
n = as.integer(readline(prompt = "Enter a number:"))
if(n < 0){
    print("Enter a positive number")
}else{
    sum = 0
    while(n>0){
        sum = sum + n
        n=n-1
    }
    print(paste("The sum of numbers up to given limit is", sum))
}
```

Output:

Enter a number:10

[1] "The sum of numbers up to given limit is 55"

4.a) Describe how to iterate over a list or a data frame using a loop.

Iterating Over a List in R:

A list in R can contain elements of different types and lengths. You can iterate over each element of a list using a for loop.

Example:

```
# Define a list

my_list1 <- list(1:5)

my_list2 <- list(c("one", "two", "three"))

mylist3 <- list(TRUE)

# Use a for loop to iterate over each element in the list

for (i in my_list1)
{
  print(i)
}

for (j in my_list2)
{
  print(j)
}

for (k in my_list3)
{
  print(k)
}
```

Output:

```
1 2 3 4 5
"one" "two" "three"
TRUE
```

Iterating Over a Data Frame in R:

A data frame in R is a list of vectors of equal length, and you can iterate over rows, columns, or both.

Example:

```
# Define a data frame

df <- data.frame(Name = c("Alice", "Bob", "Charlie"),
                  Age = c(25, 30, 35),
                  Score = c(85, 90, 88),
                  stringsAsFactors = FALSE)

# Use a for loop to iterate over each row of the data frame
for (i in 1:nrow(df)) {
  cat("Row", i, ":\n")
  print(df[i, ])
}
```

Output:

Row 1 :

	Name	Age	Score
1	Alice	25	85

Row 2 :

	Name	Age	Score
2	Bob	30	90

Row 3 :

	Name	Age	Score
3	Charlie	35	88

4.b) Explain the *lapply* and *sapply* functions and their use cases and create a program that uses *lapply* to apply a function to each element of a list and returns a new list.

***Lapply()*:**

The *lapply* function in R is used to apply a function to each element of a list (or vector) and returns a list of the same length as the input list.

Syntax:

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


Sapply():

The `sapply` function in R is a variant of `lapply` that simplifies the output if possible. It tries to return a vector or matrix if appropriate, instead of a list.

Syntax:

```
sapply(X, FUN, simplify = TRUE, ...)
```

Use Cases

lapply: Use `lapply` when you want to apply a function to each element of a list and get a list as the result. It's useful for tasks like applying data transformations across multiple elements.

sapply: Use `sapply` when you expect the result to be simplified to a vector or matrix. It's handy for converting lists into simpler data structures.

Example:

```
# Define a list
```

```
my_list <- list(1, 2, 3, 4, 5)
```

```
# Define a function to square a number
```

```
square_function <- function(x) {
```

```
  return(x^2)
```

```
}
```

```
# Apply the square_function to each element of my_list using lapply
```

```
result_list <- lapply(my_list, square_function)
```

```
# Print the original and squared lists
```

```
print("Original list:")
```

```
print(my_list)
```

```
print("Squared list:")
```

```
print(result_list)
```

Output:

```
[1] "Original list:"
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 2
```

```

[[3]]
[1] 3
[[4]]
[1] 4
[[5]]
[1] 5
[1] "Squared list:"
[[1]]
[1] 1
[[2]]
[1] 4
[[3]]
[1] 9
[[4]]
[1] 16
[[5]]
[1] 25

```

5.a) Compare the functions next and break with example program.

Feature	next	break
Purpose	Skips current iteration and moves to the next one.	Terminates the loop and exits its execution.
Usage	Inside loops (for, while, etc.)	Inside loops (for, while, etc.)
Effect	Skips the code below next for the current iteration.	Immediately exits the loop, skipping subsequent iterations.
Use Case	Filtering or skipping specific iterations based on a condition.	Finding specific conditions and stopping iteration early.
Example	<pre> for (i in 1:10) { if (i %% 2 != 0) { </pre>	<pre> for (i in 1:10) { if (i > 5 && i %% 2 == 0) { </pre>

	<pre> next # Skip to the next iteration if i is odd } print(i) } </pre>	<pre> print(paste("First even number greater than 5:", i)) break # Exit the loop once the condition is met } } </pre>
--	---	--

5.b) Explain the concept of *nested if-else* statements and prepare an example that uses the *ifelse()* function to create a new vector based on conditions.

Nested if-else Statements:

In R, nested if-else statements allow you to evaluate multiple conditions sequentially within each other. This structure is useful when you have multiple criteria to evaluate and different actions to take based on those criteria.

Syntax:

```

if(condition) {
    if(condition 1) {
        statement
    } else {
        statement
    }
} else {
    if(child condition 2 is true) {
        execute this statement
    } else {
        execute this statement
    }
}

```

Example using if-else():

```

# Example: Using ifelse() to categorize exam scores into Pass/Fail

scores <- c(85, 70, 92, 60, 78)

pass_fail <- ifelse(scores >= 70, "Pass", "Fail")

```

```
print(pass_fail)
```

Output:

```
"Pass" "Pass" "Pass" "Fail" "Pass"
```

6.a) Classify various types of operators in R and write about any two operators.**Operator:**

R has many operators to carry out different mathematical and logical operations. Operators in R can mainly be classified into the following categories.

- Arithmetic operators.
- Relational operators.
- Logical operators.
- Assignment operators.
- Miscellaneous Operators.

Arithmetic operators: These operators are used to carry out mathematical operations like addition, subtraction, division and multiplication.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

Example:

```
vec1 <- c(0, 2)
```

```
vec2 <- c(2, 3)
```

```
# Performing operations on Operands
```

```
cat ("Addition of vectors :", vec1 + vec2, "\n")
```

```
cat ("Subtraction of vectors :", vec1 - vec2, "\n")
cat ("Multiplication of vectors :", vec1 * vec2, "\n")
cat ("Division of vectors :", vec1 / vec2, "\n")
cat ("Modulo of vectors :", vec1 %% vec2, "\n")
cat ("Power operator :", vec1 ^ vec2)
```

Output:

```
Addition of vectors : 2 5
Subtraction of vectors : -2 -1
Multiplication of vectors : 0 6
Division of vectors : 0 0.6666667
Modulo of vectors : 0 2
Power operator : 0 8
```

Miscellaneous Operators: Miscellaneous operators are the mixed operators in R that simulate the printing of sequences and assignment of vectors, either left or right-handed.

Operator	Description
:	Colon Operator. It creates a sequence of numbers.
%in%	This operator is used to identify if an element belongs to a vector or not.
%*%	This operator is used for matrix multiplication.

6.b) Write a program that demonstrates the use of various arithmetic and Boolean operators.

Program for Arithmetic Operators:

```
vec1 <- c(0, 2)
vec2 <- c(2, 3)

# Performing operations on Operands

cat ("Addition of vectors :", vec1 + vec2, "\n")
cat ("Subtraction of vectors :", vec1 - vec2, "\n")
cat ("Multiplication of vectors :", vec1 * vec2, "\n")
cat ("Division of vectors :", vec1 / vec2, "\n")
```

```
cat ("Modulo of vectors :", vec1 %% vec2, "\n")
cat ("Power operator :", vec1 ^ vec2)
```

Output:

Addition of vectors : 2 5
Subtraction of vectors : -2 -1
Multiplication of vectors : 0 6
Division of vectors : 0 0.6666667
Modulo of vectors : 0 2
Power operator : 0 8

Program for Boolean Operator:

```
vec1 <- c(0,2)
vec2 <- c(TRUE,FALSE)
# Performing operations on Operands
cat ("Element wise AND :", vec1 & vec2, "\n")
cat ("Element wise OR :", vec1 | vec2, "\n")
cat ("Logical AND :", vec1[1] && vec2[1], "\n")
cat ("Logical OR :", vec1[1] || vec2[1], "\n")
cat ("Negation :", !vec1)
```

Output:

Element wise AND : FALSE FALSE
Element wise OR : TRUE TRUE
Logical AND : FALSE
Logical OR : TRUE
Negation : TRUE FALSE

7.a) Explain how to set default values for function arguments in R.

Default arguments:

We can assign default values to arguments in a function in R. This is done by providing an appropriate value to the formal argument in the function declaration. When a default value is set for an argument, the function will use that value if the argument is not explicitly provided when the function is called.

Here's how you can set default values for function arguments in R:

Define the function: Specify the default values in the function definition.

Call the function: You can call the function without providing values for the arguments with defaults, and R will use the default values.

Syntax:

```
function_name <- function(arg1 = default1, arg2 = default2, ...) {  
  # Function body  
}
```

Example:

Define a function to calculate the area of a rectangle with default arguments

```
calculate_area <- function(length = 1, width = 1) {  
  area <- length * width  
  return(area)  
}
```

Function call without arguments

```
print(calculate_area())
```

function call with one argument

```
print(calculate_area(5))
```

function call with both arguments

```
print(calculate_area(5, 3))
```

Output:

1

5

15

7.b) Justify the importance of default values and their use cases and create a function that takes two arguments with default values and returns their sum.

```
# Define a function with default values
```

```
sum <- function(a = 10, b = 5) {
```

```
  result <- a + b
```

```
  return(result)
```

```
}
```

```
# Function call without arguments, should return the sum of default values
```

```
print(sum())
```

```
# Function call with one argument, should use default value for the second argument
```

```
print(sum(5))
```

```
# Function call with both arguments
```

```
print(sum(5, 3))
```

Output:

15

10

8

8.a) Describe the *return()* function in R and its purpose.

Return():

The *return()* function in R is used to control the output of a function and terminate its execution. It allows you to specify a single value or object that the function will pass back to the code that called it.

Implicit Return: If you don't explicitly use *return()*, R will automatically return the value of the last evaluated expression in the function. This can be useful for simple functions where the desired output is the result of the final calculation.

Early Termination: You can use *return()* anywhere within the function's body to exit the function immediately and return the specified value. This is useful for conditional statements where you want to return different results based on certain conditions.

Returning Multiple Values: While *return()* can only return a single object, you can use R's list data structure to return multiple values. By creating a list and populating it with the desired values, you can return them as a single unit from the function.

Syntax:

```
return(value)
```

Example:

```
# Define a function to add two numbers and return the result
```

```
add <- function(x, y) {
```

```
  result <- x + y
```

```
  return(result)
```

```
}
```

```
# function call and print the result
```

```
print(add(3, 5))
```

Output:

```
8
```

8.b) Illustrate the concept of implicit return in R functions.

Implicit Return in R Functions

In R, functions have an implicit return behaviour, meaning that they automatically return the value of the last evaluated expression, even if no explicit *return()* statement is used. This feature simplifies function definitions by removing the need to explicitly specify what should be returned, provided it is the final result of the function.

Example:

```
# Define a function to add two numbers without using return()
```

```
add <- function(x, y) {
```

```
  x + y
```

```
}
```

```
# Call the function and print the result
```

```
print(add(3, 5))
```

Output:

8

```
# Define a function with multiple expressions
```

```
calculate <- function(a, b) {
```

```
  sum <- a + b
```

```
  product <- a * b
```

```
  product # The last evaluated expression
```

```
}
```

```
# Call the function and print the result
```

```
print(calculate(3, 5))
```

Output:

15

9.a) Discuss the factors to consider when deciding whether to use an explicit `return ()` statement.

The factors to consider when deciding whether to use an explicit `return()` statement in R functions:

Complex Functions: In functions with multiple calculations, conditional statements, or loops, explicit `return()` statements become more important. They improve code clarity by explicitly stating which value is returned under different conditions. This makes the code easier to understand and maintain for yourself and others.

Error Handling: Using explicit `return()` statements allows you to return specific error messages or values in case of errors. This helps with better error handling and makes debugging easier.

Early Termination: If a function needs to exit based on certain conditions and return a specific value, explicit `return()` is necessary. This allows you to control the flow of the function and return different outputs based on the situation.

Returning from Loops: While loops in R typically iterate until the loop condition is no longer met, you might want to exit the loop prematurely based on certain criteria. Explicit `return()` within the loop allows for this controlled early termination.

9.b) Explain how to return complex objects, such as lists or data frames, from R functions.

Returning complex objects, such as lists or data frames, from R functions is a common practice in R programming. This allows you to encapsulate multiple values or structures within a single function return, making your functions more versatile and powerful.

Returning Lists from R Functions

Lists are a flexible way to return multiple values or a collection of heterogeneous objects from a function.

Example:

```
# Define a function that returns a list
calculate_stats <- function(numbers) {
  sum_val <- sum(numbers)
  mean_val <- mean(numbers)
  sd_val <- sd(numbers)

  # Create a list containing the calculated statistics
  result <- list(sum = sum_val, mean = mean_val, sd = sd_val)

  # Return the list
  return(result)
}

# Call the function
stats <- calculate_stats(c(1, 2, 3, 4, 5))

# Print the returned list
print(stats)
```

Output:

```
$sum
[1] 15
$mean
```

```
[1] 3
```

```
$sd
```

```
[1] 1.581139
```

Returning Data Frames from R Functions

Data frames are a convenient way to return tabular data from a function.

Example:

```
# Define a function that returns a data frame
```

```
create_summary_df <- function(numbers) {
```

```
  sum_val <- sum(numbers)
```

```
  mean_val <- mean(numbers)
```

```
  sd_val <- sd(numbers)
```

```
# Create a data frame containing the calculated statistics
```

```
result_df <- data.frame(
```

```
  Statistic = c("Sum", "Mean", "Standard Deviation"),
```

```
  Value = c(sum_val, mean_val, sd_val)
```

```
)
```

```
# Return the data frame
```

```
return(result_df)
```

```
}
```

```
# Call the function
```

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

```
# Print the returned data frame
```

```
print(summary_df)
```

Output:

	Statistic	Value
1	Sum	15.000000
2	Mean	3.000000
3	Standard Deviation	1.581139

10.a) Illustrate the concept of functions being first-class citizens in R.

R treats functions as first-class citizens, meaning they can be treated just like any other data type. This allows for powerful and flexible programming by enabling you to:

1. Assigning Functions to Variables
2. Passing Functions as Arguments
3. Returning Functions from Other Functions

Assigning Functions to Variables:

You can assign a function to a variable, which allows you to use the variable name to call the function.

Example:

```
square <- function(x) { return(x * x) }
```

```
# Assign the function to a variable
```

```
my_squaring_function <- square
```

```
# Use the variable to call the function
```

```
result <- my_squaring_function(5)
```

```
print(result)
```

Output:

```
25
```

Passing Functions as Arguments:

Functions can be passed as arguments to other functions, enabling higher-order functions that operate on or produce other functions.

Example:

```
# Function to apply another function to a list
```

```
apply_function <- function(func, my_list) {  
  # Loop through the list and apply the passed function  
  for (i in seq_along(my_list)) {  
    my_list[i] <- func(my_list[i])  
  }  
  return(my_list)  
}
```

```
# Square each element in a list using apply_function  
number_list <- c(2, 4, 6)  
squared_list <- apply_function(square, number_list)  
print(squared_list)
```

Output:

```
4 16 36
```

Returning Functions from Other Functions:

A function can return another function, which is useful for creating function factories or closures.

Example:

```
# Function to create an adder function based on a given value
```

```
create_adder <- function(x) {  
  # Create a function that adds 'x' to a value  
  adder <- function(y) {  
    return(x + y)  
  }  
  return(adder)  
}
```

```
# Generate an adder function that adds 5
```

```
add_5 <- create_adder(5)
```

```
# Use the generated function
```

```
result <- add_5(10)
```

```
print(result)
```

Output:

15

10.b) Discuss the absence of pointers in R. Create a recursive function that calculates the *nth* Fibonacci number.

Absence of Pointers in R

R is an interpreted language and does not inherently support pointers like C or C++. This means you cannot directly manipulate memory addresses. R manages memory allocation and deallocation automatically, which simplifies memory management for the programmer but removes some of the low-level control.

Example:

```
# Define a recursive function to calculate the nth Fibonacci number
```

```
fibonacci <- function(n) {
```

```
  if (n == 0) {
```

```
    return(0)
```

```
  } else if (n == 1) {
```

```
    return(1)
```

```
  } else {
```

```
    # Recursive case
```

```
    return(fibonacci(n - 1) + fibonacci(n - 2))
```

```
  }
```

```
}
```

```
# Test the function
```

```
print(fibonacci(10))
```

Output:

55

UNIT – III

MATH FUNCTIONS, SIMULATION IN R AND EXTENDED EXAMPLE CALCULATING PROBABILITY

1. Identify different built-in mathematical functions in R with for example for each.

Math Function: R contains built-in functions for various math operations and for statistical distributions.

Function	Explanation	Example
exp()	Exponential function, base e	> exp(2) [1] 7.389056
log()	Natural logarithm	> log(10) [1] 2.302585
log10()	Logarithm base 10	> log10(10) [1] 1
sqrt()	Square root	> sqrt(16) [1] 4
abs()	Absolute value	> abs(-12.5) [1] 12.5
sin()	Trigonometric functions	> sin(40) [1] 0.7451132
cos()	Trigonometric functions	> cos(12) [1] 0.843854
min()	Minimum value within a vector	> x <- c(1,4,-423,8,-2,23) > min(x) [1] -423
max()	Maximum value within a vector	> x <- c(1,4,-423,8,-2,23) > max(x) [1] 23
which.min()	Index of minimal element of the vector	> x <- c(1,4,-423,8,-2,23) > which.min(x) [1] 3
which.max()	Index of maximal element of the vector	> x <- c(1,4,-423,8,-2,23) > which.max(x)

		[1] 6
pmin()	Element-wise minima of several vectors	<pre>> x <- c(4,-5,56) > y <- c(3,2,7) > pmin(x,y) [1] 3 -5 7</pre>
pmax()	Element-wise maxima of several vectors	<pre>> x <- c(4,-5,56) > y <- c(3,2,7) > pmax(x,y) [1] 4 2 56</pre>
sum()	Sum of the elements of the vector	<pre>1.1 > x <- c(4,-5,56) 1.2 > sum(x) 1.3 [1] 55</pre>
prod()	Product of the elements of the vector	<pre>> y <- 1:3 > prod(y) [1] 6</pre>
cumsum()	Cumulative sum of the elements of a vector	<pre>> y <- 1:3 > cumsum(y) [1] 1 3 6</pre>
cumprod()	Cumulative product of the elements of a vector	<pre>> z <- c(2,5,3) > cumprod(z) [1] 2 10 30</pre>
round()	Round of the closest integer	<pre>2 > round(12.4) 3 [1] 12 4 > round(2.43,digits=1) 5 [1] 2.4</pre>

floor()	Round of the closest integer below	> floor(12.4) [1] 12
ceiling()	Round of the closest integer above	> ceiling(12.4) [1] 13
factorial()	Factorial function	> factorial(5) [1] 120

sin(), cos(), tan() and so on: Trig functions, the arguments will be in radians ,asin(), acos(), atan() inverse trigonometry functions.

```
R 4.3.3 ~/  
> tan(45*pi/180)  
[1] 1  
> a<-tan(45*pi/180)  
> b<-atan(a)  
> b  
[1] 0.7853982  
> b*180/pi  
[1] 45
```

sum(): sum returns the sum of all the values present in its arguments.

Syntax: sum(..., na.rm = FALSE)

- ... : numeric or complex or logical vectors.
- na.rm : logical. Should missing values (including NaN) be removed?

Examples:

```
R 4.3.3 ~/  
> x <- c(1,3,5)  
> sum(x)  
[1] 9
```


```
R 4.3.3 ~/  
> y <- c(2,3,NA,1)  
> sum(y)  
[1] NA  
> sum(y, na.rm=TRUE)  
[1] 6
```


prod(): prod returns the product of all the values present in its arguments. prod(..., na.rm = FALSE)

Syntax: prod(..., na.rm = FALSE)

- ... : numeric or complex or logical vectors.
- na.rm : logical. Should missing values (including NaN) be removed?

Examples:


```
R 4.3.3 · ~/ 
> x <- c(1,3,5)
> prod(x)
[1] 15
```

```
R 4.3.3 · ~/ 
> y <- c(2,3,NA,1)
> prod(y)
[1] NA
> prod(y, na.rm=TRUE)
[1] 6
```


2.a) Categorize the different methods for calculating minimum, maximum, and cumulative sum statistics on vectors in R.

Cumulative Sums and Products:

A cumulative sum is a sequence of partial sum of a given sequence. For example, the cumulative sum of the sequence {a,b,c,.....} are a,ab,abc , Returns a vector whose elements are the cumulative sum.

```
R 4.3.3 · ~/ 
> x <- c(2,4,3)
> cumsum(x)
[1] 2 6 9
```

A cumulative product is a sequence of partial products of a given sequence. For example, the cumulative products of the sequence {a,b,c,.....} are a,ab,abc , Returns a vector whose elements are the cumulative product.


```
R 4.3.3 · ~/ 
> x <- c(2,4,3)
> cumprod(x)
[1] 2 8 24
```

Minima and maxima:

max() function computes the maximum value of a vector. **min()** function computes the minimum value of a vector.

- ✓ **x:** number vector
- ✓ **na.rm:** whether NA should be removed, if not, NA will be returned

1. Syntax: max(, na.rm = FALSE)

```
R 4.3.3 · ~/ 
> max(c(12,4,6,NA,34))
[1] NA
> max(c(12,4,6,NA,34), na.rm=FALSE)
[1] NA
> max(c(12,4,6,NA,34), na.rm=TRUE)
[1] 34
> x <- c(2,-4,6,-34)
> min(x[1],x[4])
[1] -34
```

2. Syntax: min(., na.rm = FALSE)

```
R 4.3.3 · ~/
> min(c(12,4,6,NA,34))
[1] NA
> min(c(12,4,6,NA,34), na.rm=TRUE)
[1] 4
> min(c(12,4,6,NA,34), na.rm=FALSE)
[1] NA
> x <- c(2,-4,6,-34)
> max(x[2],x[3])
[1] 6
```

which.min() and which.max(): Index of the minimal element and maximal element of a vector.

```
R 4.3.3 · ~/
> x <- c(1,4,-423,8,-2,23)
> which.min(x)
[1] 3
> which.max(x)
[1] 6
```

pmin() and pmax(): Element-wise minima and maxima of several vectors.

There is quite a difference between `min()` and `pmin()`. The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if `pmin()` is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name `pmin`.

✓ The `max()` and `pmax()` functions act analogously to `min()` and `pmin()`.

1. **Syntax:** `pmax(..., na.rm = FALSE)`

```
R 4.3.3 · ~/
> x <- c(12,4,6,NA)
> y <- c(2,34,56,1)
> pmax(x,y)
[1] 12 34 56 NA
> pmax(x,y, na.rm=TRUE)
[1] 12 34 56 1
```

2. **Syntax:** `pmin(..., na.rm = FALSE)`

```
R 4.3.3 · ~/
> x <- c(12,4,6,NA)
> y <- c(2,34,56,1)
> pmin(x,y)
[1] 2 4 6 NA
> pmin(x,y, na.rm=TRUE)
[1] 2 4 6 1
```

Function minimization/maximization can be done via `nlm()` and `optim()`. For example, let's find the smallest value of $f(x) = x^2 - \sin(x)$.

Here, the minimum value was found to be approximately -0.23 , occurring at $x = 0.45$. A Newton Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case. The second argument specifies the initial guess, which we set to be 8.

```
R 4.3.3 ~ /
> nlm(function(x) return(x^2-sin(x)),8)
$minimum
[1] -0.2324656

$estimate
[1] 0.4501831

$gradient
[1] 4.024558e-09

$code
[1] 1

$iterations
[1] 5
```

2.b) Differentiate between cumulative sums and products in the context of numerical analysis within R.

Cumulative Sums:

Definition: Cumulative sum refers to a sequence where each element is the sum of all previous elements in the sequence, including the current element. For a given vector xxx , the cumulative sum yyy is defined as: $y[i] = x[1] + x[2] + \dots + x[i]$

Function in R: The function used in R to calculate the cumulative sum is `cumsum()`.

Usage Example:

```
x <- c(1, 2, 3, 4)
```

```
y <- cumsum(x)
```

```
print(y) # Output: 1 3 6 10
```

Applications:

- **Time Series Analysis:** To analyze the running total of a series of data points over time.
- **Probability:** To calculate cumulative distribution functions.
- **Economics:** To compute cumulative revenue or expenses over time.

Cumulative Products:

Definition: Cumulative product refers to a sequence where each element is the product of all previous elements in the sequence, including the current element. For a given vector xxx , the cumulative product yyy is defined as: $y[i] = x[1] \times x[2] \times \dots \times x[i]$

Function in R: The function used in R to calculate the cumulative product is `cumprod()`.

Usage Example:

```
x <- c(1, 2, 3, 4)
```

```
y <- cumprod(x)
print(y) # Output: 1 2 6 24
```

Applications:

- **Compound Interest:** To calculate the growth of investments over time.
- **Population Growth Models:** To model exponential growth processes.
- **Geometric Brownian Motion:** Used in financial mathematics to model stock prices.

3.a) Create a R program for calculating the probability?

Calculating a probability :

Now we see how to find the probability that exactly one event occur: If three friends x, y, z appeared for an examination x has 17% chance of failure, y has 7% chance of failure, and Z has 26% chance of failure. What is the probability that exactly one of them will fail in the exams?

$$P(X \text{ fails, but not others}) = 0.17 * 0.93$$

$$* 0.74, P(Y \text{ fails, but not others}) = 0.83$$

$$* 0.07 * 0.74, P(Z \text{ fails, but not others})$$

$$= 0.83 * 0.93 * 0.26.$$

The probability can be calculated using the prod() function. Let us assume that there are n's independent events with the ith event having the pi probability of occurrence. What is the probability of exactly one of these events occurring?

Considering an example where the value of n is 3. The events are named A, B, and C. Then we break down the computation as follows:

$$\underline{P(\text{exactly one event occurs}) = P(A \text{ and not } B \text{ and not } C) + P(\text{not } A \text{ and } B \text{ and not } C) + P(\text{not } A \text{ and not } B \text{ and } C)}$$

$$P(A \text{ and not } B \text{ and not } C) \text{ would be } p_A (1 - p_B) (1 - p_C),$$

and so on. For general n, that is calculated as follows:

$$\sum_{i=1}^n p_i (1 - p_1) \dots (1 - p_{i-1}) (1 - p_{i+1}) \dots (1 - p_n)$$

(The ith term inside the sum is the probability that event i occurs and all the others do not occur.) Here's code to compute this, with our probabilities pi contained in the vector p:

```
R 4.3.3 · ~/
> exactlyone <- function(p) {
  notp <- 1 - p
  tot <- 0.0
  for (i in 1:length(p))
    tot <- tot + p[i] * prod(notp[-i])
  return(tot)
}
```

notp <- 1 - p :- creates a vector of all the —not occur probabilities $1 - p_j$, using recycling. The expression `notp[-i]` computes the product of all the elements of `notp`, except the *i*th.

3.b) Explain following functions with example

i) **dnorm** ii) **qchisq** iii) **qbinom** iv) **rnorm**.

i) **dnorm**:

Definition: The `dnorm` function in R computes the density of the normal distribution (i.e., the value of the probability density function) at a given point.

Usage:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
```

- **x:** The point at which the density is to be calculated.
- **mean:** The mean of the normal distribution (default is 0).
- **sd:** The standard deviation of the normal distribution (default is 1).
- **log:** If TRUE, probabilities are given as $\log(p)$ (default is FALSE).

Example:

```
# Calculate the density of the normal distribution at x = 1
```

```
density <- dnorm(1, mean = 0, sd = 1)
```

```
print(density)
```

Output:

```
[1] 0.2419707
```

This means the density of the normal distribution with mean 0 and standard deviation 1 at $x=1$ is approximately 0.242.

ii) **qchisq**:

Definition: The `qchisq` function in R computes the quantile function (inverse of the cumulative distribution function) for the chi-squared distribution.

Usage:

```
qchisq(p, df, lower.tail = TRUE, log.p = FALSE)
```

- **p:** The probability for which the quantile is to be calculated.

- df: Degrees of freedom.
- lower.tail: If TRUE (default), probabilities are $P(X \leq x)$, otherwise $P(X > x)$.
- log.p: If TRUE, probabilities p are given as $\log(p)$.

Example:

```
# Calculate the 95th percentile of the chi-squared distribution with 5 degrees of freedom
quantile <- qchisq(0.95, df = 5)
print(quantile)
```

Output:

```
[1] 11.0705
```

This means the 95th percentile of the chi-squared distribution with 5 degrees of freedom is approximately 11.071.

iii) qbinom:

Definition: The qbinom function in R computes the quantile function for the binomial distribution.

Usage:

```
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
```

- p: The probability for which the quantile is to be calculated.
- size: Number of trials.
- prob: Probability of success on each trial.
- lower.tail: If TRUE (default), probabilities are $P(X \leq x)$, otherwise $P(X > x)$.
- log.p: If TRUE, probabilities p are given as $\log(p)$.

Example:

```
# Calculate the quantile for a binomial distribution with 10 trials, probability of success 0.5,
and cumulative probability 0.75
```

```
quantile <- qbinom(0.75, size = 10, prob = 0.5)
print(quantile)
```

Output:

```
[1] 6
```

This means the 75th percentile of a binomial distribution with 10 trials and probability of success 0.5 is 6.

iv) rnorm:

Definition: The rnorm function in R generates random numbers from a normal distribution.

Usage:

`rnorm(n, mean = 0, sd = 1)`

- `n`: Number of random numbers to generate.
- `mean`: The mean of the normal distribution (default is 0).
- `sd`: The standard deviation of the normal distribution (default is 1).

Example:

Generate 5 random numbers from a normal distribution with mean 0 and standard deviation 1

```
random_numbers <- rnorm(5, mean = 0, sd = 1)
```

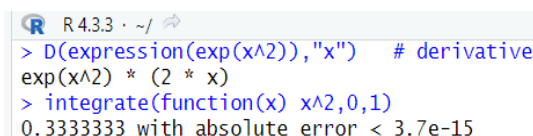
```
print(random_numbers)
```

Output (will vary each time you run the code):

```
[1] -0.5604756 -0.2301775 1.5587083 0.0705084 0.1292877
```

4. Evaluate the capabilities of R for performing basic calculus operations (e.g., differentiation, integration).

Calculus: R also has some calculus capabilities, including symbolic differentiation and numerical integration.



```
R 4.3.3 ~ /
> D(expression(exp(x^2)), "x") # derivative
exp(x^2) * (2 * x)
> integrate(function(x) x^2, 0, 1)
0.3333333 with absolute error < 3.7e-15
```

Here, R reported $\frac{d}{dx} e^{x^2} = 2xe^{x^2}$ and $\int_0^1 x^2 dx = 0.3333333$

R packages for differential equations, for interfacing R with the Yacas symbolic math system (ryacas), and for other calculus operations. These packages, and thousands of others, are available from the Comprehensive R Archive Network (CRAN)

(or)

R, while primarily known for its strengths in statistical computing and data analysis, offers a variety of capabilities for performing basic calculus operations. Let's evaluate these capabilities in more detail:

Differentiation:

1. Numerical Differentiation:

- R allows for numerical differentiation using finite difference methods.

- The `diff()` function computes first differences of a numeric vector.
- Custom functions can be created to approximate derivatives numerically using small increments (h).

- **Example:**

```
f <- function(x) x^2 + 3*x + 2
derivative_approx <- function(x, f, h = 1e-6) {
  (f(x + h) - f(x)) / h
}
result <- derivative_approx(2, f)
print(result) # Approximate derivative of f(x) at x=2
```

2. Symbolic Differentiation:

- R itself does not have built-in support for symbolic differentiation.
- Packages like Ryacas provide interfaces to computer algebra systems (CAS) like Yacas for symbolic differentiation.

- **Example using Ryacas:**

```
library(Ryacas)
yacas("D(x^2 + 3*x + 2, x)")
```

Integration:

1. Numerical Integration:

- R offers numerical integration capabilities through the `integrate()` function.
- Supports both definite and indefinite integrals over finite or infinite ranges.

- **Example:**

```
g <- function(x) sin(x)
result <- integrate(g, lower = 0, upper = pi)
print(result)
```

2. Symbolic Integration:

- Similar to differentiation, R does not natively support symbolic integration.
- External packages or interfaces to CAS like Ryacas can be used for symbolic integration.

- **Example using Ryacas:**

```
library(Ryacas)
yacas("Integrate(sin(x), x)")
```

Summary of Capabilities:

- **Differentiation:**

- R supports numerical differentiation using finite differences and custom functions.
- Symbolic differentiation is achievable through packages like Ryacas that interface with external CAS.

- **Integration:**


- Numerical integration is straightforward with the `integrate()` function in R.
- Symbolic integration can be performed using external packages or interfaces to CAS like Ryacas.

5. Explain efficient techniques for sorting data vectors in R based on specific criteria.


Sorting: Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order.

Syntax: `sort(x, decreasing, na.last)`

Example:


```
R 4.3.3 · ~/ 
> x <- c(12,4,25,4)
> sort(x)
[1] 4 4 12 25
> x
[1] 12 4 25 4
```

The vector `x` did not change actually as printed in the very last line of the code. In order to sort the indexes as such, the `order` function is used in the following manner.

```
R 4.3.3 · ~/ 
> order(x)
[1] 2 4 1 3
```

The console represents that there are two smallest values in vector `x`. The third smallest value being `x[1]`, and so on. The same function `order` can also be used along with indexing for sorting data frames. This function can also be used to sort the characters as well as numeric values.

Another function which specifies the rank of every single element present in a vector is called `rank()`

```
R 4.3.3 · ~/ 
> x
[1] 12 4 25 4
> rank(x)
[1] 3.0 1.5 4.0 1.5
```

The above console demonstrates that the value 12 lies at rank 4th, which means that the 3rd smallest element in x is 12. Now, 4 number appears two times in the vector x. So, the rank 1.5 is allocated to both the numbers.

Example: using order function on a dataframe.

```
R 4.3.3 · ~/
> age <- c(12,4,34,14)
> names <- c("A","B","C","D")
> df <- data.frame(age,names)
> df
  age names
1  12     A
2   4     B
3  34     C
4  14     D
> df[order(df$age),]
  age names
2   4     B
1  12     A
4  14     D
3  34     C
```

(or)

In R, sorting data vectors can be efficiently achieved using various techniques depending on the specific criteria or requirements. Here, I'll explain efficient techniques for sorting data vectors based on different scenarios or criteria:

1. Sorting Numerical Vectors

- **Base R Function:** Use the `sort()` function for sorting numerical vectors. It sorts the vector in ascending order by default.

Example numeric vector

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

Sort in ascending order

```
sorted_x <- sort(x)
```

```
print(sorted_x) # Output: 1 2 3 4 5
```

- **Descending Order:** To sort in descending order, use the `decreasing = TRUE` argument.

Sort in descending order

```
sorted_x_desc <- sort(x, decreasing = TRUE)
```

```
print(sorted_x_desc) # Output: 5 4 3 2 1
```

2. Sorting Character Vectors

- **Base R Function:** Use `sort()` function for sorting character vectors alphabetically.

Example character vector

```
names <- c("Alice", "Bob", "Charlie", "David")
```

```
# Sort alphabetically
sorted_names <- sort(names)

print(sorted_names) # Output: "Alice" "Bob" "Charlie" "David"
```

- **Custom Sorting:** For custom sorting orders, use the `order()` function to get the index and then reorder the vector.

```
# Custom sorting order
custom_order <- c("Bob", "David", "Charlie", "Alice")
```

```
# Get the index based on custom order
idx <- match(names, custom_order)
```

```
# Sort based on custom order
sorted_names_custom <- names[order(idx)]

print(sorted_names_custom) # Output: "Bob" "David" "Charlie" "Alice"
```

3. Sorting Data Frames

- **Base R Function:** Use `order()` function to sort data frames based on one or more columns.

```
# Example data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie", "David"),
  Age = c(25, 30, 22, 28)
)
```

```
# Sort data frame by Age in ascending order
sorted_df <- df[order(df$Age), ]

print(sorted_df)
```

- **Descending Order:** Use `-` sign to sort in descending order.

```
# Sort data frame by Age in descending order
sorted_df_desc <- df[order(-df$Age), ]

print(sorted_df_desc)
```

6.a) Classify fundamental linear algebra operations on vectors and matrices in R, including addition, subtraction, and scalar multiplication.

Linear algebra operations form the backbone of many computational and data analysis tasks in R. Here's a classification of fundamental operations such as addition, subtraction, and scalar multiplication, including examples to illustrate how they are performed in R.

1. Vector Operations

a) Addition:

Adding two vectors of the same length involves element-wise addition of their corresponding elements.

```
# Define two vectors
```

```
v1 <- c(1, 2, 3)
```

```
v2 <- c(4, 5, 6)
```

```
# Vector addition
```

```
v_add <- v1 + v2
```

```
print(v_add)
```

Output:

```
5 7 9
```

b) Subtraction:

Subtracting one vector from another also involves element-wise operations.

```
# Vector subtraction
```

```
v_sub <- v1 - v2
```

```
print(v_sub)
```

Output:

```
-3 -3 -3
```

c) Scalar Multiplication:

Multiplying a vector by a scalar involves multiplying each element of the vector by the scalar.

```
# Scalar multiplication
```

```
scalar <- 2
```

```
v_scalar_mul <- scalar * v1
```

```
print(v_scalar_mul) # Output: 2 4 6
```

2. Matrix Operations

a) Addition:

Matrix addition requires both matrices to have the same dimensions. The operation is performed element-wise.

Define two matrices

```
m1 <- matrix(c(1, 2, 3, 4), nrow = 2)
```

```
m2 <- matrix(c(5, 6, 7, 8), nrow = 2)
```

Matrix addition

```
m_add <- m1 + m2
```

```
print(m_add)
```

Output:

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

```
[1,]  6  10
```

```
[2,]  8  12
```

b) Subtraction:

Matrix subtraction also requires matrices to be of the same dimensions and is performed element-wise.

Matrix subtraction

```
m_sub <- m1 - m2
```

```
print(m_sub)
```

Output:

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

```
[1,] -4 -4
```

```
[2,] -4 -4
```

c) Scalar Multiplication:

Each element of the matrix is multiplied by the scalar.

Scalar multiplication

```
m_scalar_mul <- scalar * m1
```

```
print(m_scalar_mul)
```

Output:

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

```
[1,]  2  6
```

```
[2,]  4  8
```

d) Matrix Multiplication:

Matrix multiplication (not element-wise) is performed using the `%*%` operator. This requires the number of columns of the first matrix to match the number of rows of the second matrix.

```
# Define another matrix for multiplication
```

```
m3 <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2)
```

```
# Matrix multiplication
```

```
m_mul <- m1 %*% m3
```

```
print(m_mul)
```

Output:

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

```
[1,] 19 22 25
```

```
[2,] 43 50 57
```

6.b) Prioritize an example (excluding vector cross product) of how vector operations are used in linear algebra applications within R.

One common application is solving a system of linear equations using vector and matrix operations. This example will demonstrate how these operations can be employed in practice.

Example: Solving a System of Linear Equations

Consider a system of linear equations:

Consider a system of linear equations:

$$\begin{cases} 2x+3y=5 \\ 4x-y=6 \end{cases}$$

This system can be represented in matrix form as:

$$\begin{bmatrix} 2 & 3 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

Where

- A is the coefficient matrix: $A = \begin{bmatrix} 2 & 3 \\ 4 & -1 \end{bmatrix}$
- \mathbf{b} is the vector of constants: $\mathbf{b} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$

We can solve for the vector \mathbf{x} which contains the variables x and y using R.

Steps in R:

1. Define the Coefficient Matrix and Constant Vector:

Coefficient matrix A

```
A <- matrix(c(2, 3, 4, -1), nrow = 2, byrow = TRUE)
```

Constant vector b

```
b <- c(5, 6)
```

2. Solve the System Using the solve() Function:

The solve() function in R can be used to solve the equation $A\mathbf{x}=\mathbf{b}$ for \mathbf{x} .

Solve for x

```
solution <- solve(A, b)
```

```
print(solution)
```

Explanation:

- The solve() function computes the solution to the system of linear equations.
- The function takes the coefficient matrix A and the constant vector \mathbf{b} as inputs and returns the vector \mathbf{x} , which contains the values of x and y .

Expected Output:

```
[1] 2.090909 0.272727
```

This output means that the solution to the system of equations is: $x=2.090909$ and $y=0.272727$.

Applications of Vector Operations in Linear Algebra:

- **Systems of Linear Equations:** Solving linear systems is fundamental in numerous fields such as engineering, physics, economics, and computer science.
- **Linear Transformations:** Representing and computing linear transformations using matrices and vectors.

- **Optimization Problems:** Many optimization problems, especially in operations research and machine learning, involve solving systems of linear equations or inequalities.
- **Eigenvalues and Eigenvectors:** These are crucial in understanding the properties of linear transformations, stability analysis, and principal component analysis (PCA) in data science.

7. Demonstrate the vector cross product and its applications in R.

Vector Cross Product:

Let's consider the issue of vector cross products. The definition is very simple: The cross product of vectors (x_1, x_2, x_3) and (y_1, y_2, y_3) in three dimensional space is a new three dimensional vector, as $(x_2y_3 - x_3y_2, -x_1y_3 + x_3y_1, x_1y_2 - x_2y_1)$.

This can be expressed compactly as the expansion along the top row of the determinant. Here, the elements in the top row are merely placeholders.

$$\begin{pmatrix} - & - & - \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}$$

The point is that the cross product vector can be computed as a sum of sub determinants. For instance, the first component in Equation is, $x_2y_3 - x_3y_2$, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column.

$$\begin{pmatrix} x_2 & x_3 \\ y_2 & y_3 \end{pmatrix}$$

Function to calculate cross product of vectors:

```
R 4.3.3 ~ /
> xprod <- function(x,y)
+ {
+   m <- rbind(rep(NA,3),x,y)
+   xp <- vector(length=3)
+   for (i in 1:3)
+     xp[i] <- -(-1)^i * det(m[2:3,-i])
+   return(xp)
+ }
> xprod(c(12,4,2),c(2,1,1))
[1] 2 -8 4
```

Applications of Cross Product:

1. Finding a Perpendicular Vector:

The cross product of two vectors in 3D space yields a vector that is perpendicular to both. This is useful in computer graphics, robotics, and physics.

Vectors representing directions

a <- c(1, 0, 0) # Along the x-axis

b <- c(0, 1, 0) # Along the y-axis

Cross product yields a vector along the z-axis

perpendicular_vector <- cross_product(a, b)

print(perpendicular_vector) # Output: 0 0 1

2. Calculating Torque:

Torque (τ $\mathbf{\tau}$) is the rotational force and is calculated as the cross product of the position vector (r \mathbf{r}) and the force vector (F \mathbf{F}).

Position vector

r <- c(3, 0, 0) # 3 meters along the x-axis

Force vector

F <- c(0, 10, 0) # 10 Newtons along the y-axis

Torque calculation

torque <- cross_product(r, F)

print(torque) # Output: 0 0 30 (30 Nm along the z-axis)

3. Area of a Parallelogram:

The magnitude of the cross product of two vectors gives the area of the parallelogram formed by the vectors.

Vectors defining the parallelogram

a <- c(2, 3, 4)

b <- c(5, 6, 7)

Cross product to find the area

cross_prod <- cross_product(a, b)

area <- sqrt(sum(cross_prod^2))

print(area) # Output: 3.464102 (Area of the parallelogram)

8.a) Distinguish Markov chains and their significance in modeling probabilistic systems.

A Markov chain is a random process in which we move among various states, in a “memoryless” fashion, whose definition need not concern us here. The state could be the number of jobs in a queue, the number of items stored in inventory, and so on. We will assume the number of states to be finite.

As a simple example, consider a game in which we toss a coin repeatedly and win a dollar whenever we accumulate three consecutive heads.

Our state at any time i will be the number of consecutive heads we have so far, so our state can be 0, 1, or 2. (When we get three heads in a row, our state reverts to 0.)

The central interest in Markov modeling is usually the long-run state distribution, meaning the long-run proportions of the time we are in each state. In our coin-toss game, we can use the code we’ll develop here to calculate that distribution, which turns out to have us at states 0, 1, and 2 in proportions 57.1%, 28.6%, and 14.3% of the time. Note that we win our dollar if we are in state 2 and toss a head, so $0.143 \times 0.5 = 0.071$ of our tosses will result in wins.

Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads. Let p_{ij} denote the transition probability of moving from state i to state j during a time step. In the game example, for instance, $p_{23} = 0.5$, reflecting the fact that with probability $1/2$, we will toss a head and thus move from having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus $p_{21} = 0.5$.

We are interested in calculating the vector $\pi = (\pi_1, \dots, \pi_s)$, where π_i is the long-run proportion of time spent at state i , over all states i . Let P denote the transition probability matrix whose i th row, j th column element is p_{ij} . Then it can be shown that π must satisfy the Equation.

Example:

```
1 findpi1 <- function(p) {  
2   n <- nrow(p)  
3   imp <- diag(n) - t(p)  
4   imp[n,] <- rep(1,n)  
5   rhs <- c(rep(0,n-1),1)  
6   pivec <- solve(imp,rhs)  
7   return(pivec)  
8 }
```

Here are the main steps:

1. Calculate $I - P^T$ in line 3. Note again that `diag()`, when called with a scalar argument, returns the identity matrix of the size given by that argument.
2. Replace the last row of P with 1 values in line 4.
3. Set up the right-hand side vector in line 5.
4. Solve for π in line 6.

8.b) Illustrate how to import data in R programming.

Importing data into R is a fundamental task that enables data analysis and manipulation. R provides a variety of functions and packages to read data from different file formats such as CSV, Excel, text files, and databases. Here are some common methods to import data in R with examples:

1. Importing CSV Files

CSV (Comma-Separated Values) is a common file format for storing tabular data.

Using read.csv Function:

```
# Import CSV file
```

```
data <- read.csv("path/to/your/file.csv")
```

```
# Display the first few rows of the dataset
```

```
head(data)
```

- read.csv(file, header = TRUE, sep = ",", stringsAsFactors = FALSE) is the function used.
- The header argument indicates whether the first row contains column names.
- The sep argument specifies the delimiter (comma by default).
- The stringsAsFactors argument determines whether strings are converted to factors.

2. Importing Excel Files

Excel files can be imported using the readxl package.

Using readxl Package:

```
# Install the package if not already installed
```

```
install.packages("readxl")
```

```
# Load the package
```

```
library(readxl)
```

```
# Import Excel file
```

```
data <- read_excel("path/to/your/file.xlsx", sheet = 1)
```

```
# Display the first few rows of the dataset
```

```
head(data)
```

- `read_excel(path, sheet = NULL, range = NULL, col_names = TRUE, col_types = NULL)` is the function used.
- The `sheet` argument specifies the sheet to read (can be the name or index).

3. Importing Text Files

Text files with different delimiters can be imported using the `read.table` function.

Using `read.table` Function:

Import text file

```
data <- read.table("path/to/your/file.txt", header = TRUE, sep = "\t")
```

Display the first few rows of the dataset

```
head(data)
```

- The `sep` argument can be adjusted for different delimiters, such as `"\t"` for tab-separated values.

4. Importing Data from URLs

You can also import data directly from the web.

Using `read.csv` from a URL:

Import CSV data from a URL

```
url <- "http://example.com/data.csv"
```

```
data <- read.csv(url)
```

Display the first few rows of the dataset

```
head(data)
```

5. Importing Data from Databases

R can connect to various databases using packages like `DBI` and `RSQLite` or `RODBC`.

Using `DBI` and `RSQLite` Packages:

Install the packages if not already installed

```
install.packages("DBI")
```

```
install.packages("RSQLite")
```

Load the packages

```
library(DBI)
```

```
library(RSQLite)
```

```
# Create a connection to the SQLite database
con <- dbConnect(RSQLite::SQLite(), dbname = "path/to/your/database.sqlite")

# Import data from a table
data <- dbGetQuery(con, "SELECT * FROM your_table")

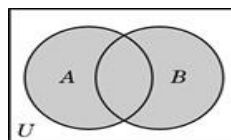
# Close the connection
dbDisconnect(con)

# Display the first few rows of the dataset
head(data)
```

9.a) Classify set operations (union, intersection, difference) and their implementation for data manipulation in R.

Set Operations: R includes some handy set operations, including these:

- **union(x,y):** The union of two sets is defined as the set of all the elements that are members of set A, set B or both and is denoted by $A \cup B$ read as A union B.



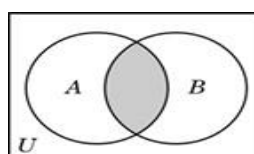
Eg: $A = \{1,2,3,4,5,a,b\}$

$B = \{a,b,c,d,e\}$

$A \cup B = \{1,2,3,4,5,a,b\} \cup \{a,b,c,d,e\}$

$= \{1,2,3,4,5,a,b,c,d,e\}$

- **intersect(x,y):** The intersection of any two sets A and B is the set containing of all the elements that belong to both A and B is denoted by $A \cap B$ read as A intersection B.

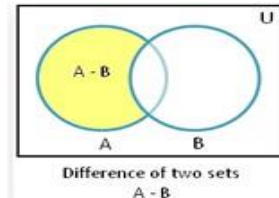


Eg: $A = \{1,2,3,4,5,a,b\}$

$B = \{a,b,c,d,e\}$

$A \ominus B = \{1,2,3,4,5,a,b\} \ominus \{a,b,c,d,e\} = \{a,b\}$

- **setdiff(x,y):** The set difference of any two sets A and B is the set of elements that belongs to A but not B. It is denoted by $A-B$ and read as 'A difference B'. $A-B$ is also denoted by $A \setminus B$ or $A \sim B$. It is also called the relative complement of B in A.



Eg: $A =$

$\{1,2,3,4,5,$

$6\}$ $B =$

$\{3,5,7,9\}$

$A-B = \{1,2,4,6\}$

$B-A = \{7,9\}$

- **setequal(x,y):** Test for equality between x and y. If both x and y are equal it returns TRUE otherwise returns FALSE
- **c %in% y:** Membership, testing whether c is an element of the set y. It checks every corresponding element of 'c with y', if both elements are equal it returns TRUE else return FALSE.
- **choose(n,r):** Number of possible subsets of size k chosen from a set of size . Ex:

```
R 4.0.3 ~ / > choose(2,1)
[1] 2
```

- ✓ choose() function computes the combination nCr .
- ✓ n: n elements
- ✓ r: r subset elements

$$nCr = \frac{n!}{(r! * (n-r)!)}$$

Example:


```

R 4.3.3 · ~/
> x <- c(1,5,3)
> y <- c(34,2,5)
> union(x,y)
[1] 1 5 3 34 2
> intersect(x,y)
[1] 5
> setdiff(x,y)
[1] 1 3
> setequal(x,y)
[1] FALSE
> choose(5,2)
[1] 10
> x %in% y
[1] FALSE TRUE FALSE
> 5 %in% y
[1] TRUE

```

combn() :-The function `combn()` generates combinations. Let's find the subsets of {1,2,3} of size 2.

```

> x <- combn(1:3,2)
> x
      [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3
> class(x)
[1] "matrix" "array"

```

The results are in the columns of the output. We see that the subsets of {1,2,3} of size 2 are (1,2), (1,3), and (2,3).

9.b) Explain reading and writing files in R.

Reading and writing files in R is essential for data manipulation, analysis, and storage. Here's a concise explanation of how to read and write files in R:

Reading Files

1. CSV Files:

- Use `read.csv()` function to read CSV files.
- **Example:**

```
data <- read.csv("file.csv")
```

2. Excel Files:

- Use `read_excel()` function from the `readxl` package to read Excel files.

- **Example:**

```
library(readxl)
data <- read_excel("file.xlsx")
```

3. Text Files:

- Use read.table() function to read text files with specified delimiters.

- **Example:**

```
data <- read.table("file.txt", header = TRUE, sep = "\t")
```

4. Other Formats:

- Use specialized packages like jsonlite for JSON files or database-specific packages (DBI, RSQLite) for database files.

Writing Files

1. CSV Files:

- Use write.csv() function to write data frames to CSV files.

- **Example:**

```
write.csv(data, "output.csv", row.names = FALSE)
```

2. Excel Files:

- Use write.xlsx() function from the openxlsx package to write data frames to Excel files.

- **Example:**

```
library(openxlsx)
write.xlsx(data, "output.xlsx")
```

3. Text Files:

- Use write.table() function to write data frames to text files with specified delimiters.

- **Example:**

```
write.table(data, "output.txt", sep = "\t", row.names = FALSE)
```

4. Other Formats:

- Use appropriate functions from packages like jsonlite::write_json() for JSON files or database-specific functions (dbWriteTable() from DBI package) for database files.

10.a) Illustrate the purpose of getwd() and setwd() functions in R.

In R, getwd() and setwd() are functions used to manage the current working directory. Understanding and using these functions is essential for efficiently navigating and accessing files on your computer within R sessions.

1. getwd() Function

The getwd() function stands for "get working directory". Its purpose is to retrieve and display the current working directory path that R is using.

Example:

```
# Get the current working directory
current_dir <- getwd()
print(current_dir)
```

Output:

```
csharp
[1] "/Users/username/Documents"
```

2. setwd() Function

The setwd() function stands for "set working directory". Its purpose is to change the current working directory to a specified directory path.

Example:

```
# Set a new working directory
setwd("/Users/username/Data")
```

In this example, setwd("/Users/username/Data") changes the current working directory to "/Users/username/Data". Now, any subsequent file operations or data imports/exports will be relative to this new directory unless specified otherwise.

Purpose and Usage:

- **Managing Files:** getwd() allows you to check which directory R is currently using, which is helpful when dealing with multiple projects or directories.
- **Navigating Directories:** setwd() helps in navigating to different directories where your data files or scripts are located, ensuring that R can find and access them easily.
- **Facilitating File Operations:** By setting the working directory, you simplify file operations (like reading or writing files) because you can specify relative paths rather than absolute paths each time.

10.b) Explain the purpose of the following functions in R:

i. read.csv()

ii. write.csv()

iii. read.xlsx()

In R, the functions `read.csv()`, `write.csv()`, and `read.xlsx()` serve specific purposes related to reading and writing data from/to different file formats. Let's delve into each function and their respective purposes:

1. read.csv()

Purpose:

The `read.csv()` function is used to read data from CSV (Comma-Separated Values) files into R as a data frame. CSV files are widely used for storing tabular data where each line represents a row and values are separated by commas (or other delimiters).

Syntax:

```
read.csv(file, header = TRUE, sep = ",", ...)
```

- `file`: Path to the CSV file.
- `header`: Logical value indicating if the file has a header line (default is TRUE).
- `sep`: Separator used in the file (default is ",").
- `...`: Additional arguments to be passed to `read.table()` (e.g., `stringsAsFactors`).

Example:

```
# Read data from a CSV file
```

```
data <- read.csv("data.csv")
```

2. write.csv()

Purpose:

The `write.csv()` function is used to write data frames or matrices as CSV files in R. It allows you to export data from R into a CSV file format that can be easily shared or imported into other software applications.

Syntax:

```
write.csv(x, file, row.names = FALSE, ...)
```

- `x`: Data frame or matrix to be written to the CSV file.
- `file`: Path where the CSV file will be written.
- `row.names`: Logical value indicating if row names should be included (default is FALSE).
- `...`: Additional arguments to be passed to `write.table()`.

Example:

```
# Example data frame
data <- data.frame(
  Name = c("John", "Jane", "Bob"),
  Age = c(25, 30, 28)
)
# Write data frame to a CSV file
write.csv(data, "output.csv", row.names = FALSE)
```

3. read.xlsx()

Purpose:

The read.xlsx() function is used to read data from Excel files (both .xls and .xlsx formats) into R as a data frame. Excel files are commonly used for storing data with complex formatting and multiple sheets.

Syntax:

```
read.xlsx(file, sheet = 1, range = NULL, ...)
```

- file: Path to the Excel file.
- sheet: Name or index of the sheet to read (default is 1).
- range: Range of cells to read (e.g., "A1:B10").
- ...: Additional arguments to be passed to read.xlsx() (e.g., colNames, skipEmptyRows).

Example:

```
# Install the 'readxl' package if not already installed
# install.packages("readxl")

# Load the 'readxl' package
library(readxl)

# Read data from an Excel file
data <- read_excel("data.xlsx", sheet = 1)
```

Unit – IV

GRAPHICS

1.a) Describe the role of the plot() function in R base graphics.

The Workhouse of R Base Graphics: The plot()Function:

The plot() function forms the foundation for much of R's base graphing operations, serving as the vehicle for producing many different kinds of graphs. plot() is a generic function, or a placeholder for a family of functions. The function that is actually called depends on the class of the object on which it is called. The basic syntax to create a line chart in R is

plot(v, type, col, xlab, ylab)

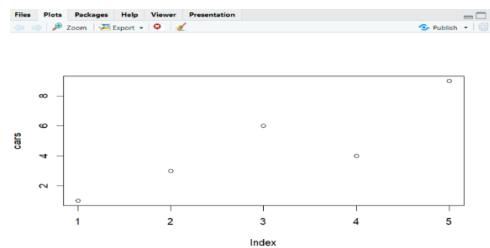
Following is the description of the parameters used –

- ✓ **v** is a vector containing the numeric values.
- ✓ **type** takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
- ✓ **xlab** is the label for x axis.
- ✓ **ylab** is the label for y axis.
- ✓ **main** is the Title of the chart.
- ✓ **col** is used to give colors to both the points and lines.

Examples of plot function:

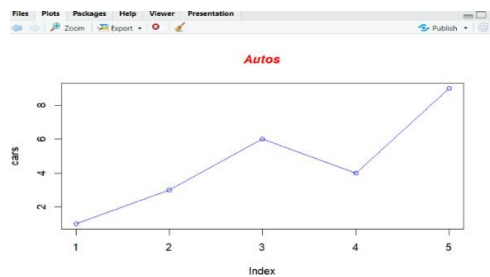
```
cars <- c(1, 3, 6, 4, 9)
```

```
plot(cars)
```

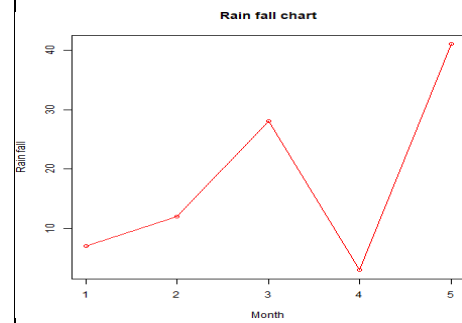


```
cars <- c(1, 3, 6, 4, 9)
```

```
plot(cars, type="o", col="blue")  
title(main="Autos", col.main="red", font.main=4)
```



```
v <- c(7,12,28,3,41)
png(file = "line_chart.jpg")
plot(v, type = "o", col = "red", xlab = "Month",
     ylab = "Rain fall", main = "Rain fall chart")
dev.off()
```



1.b) What are the essential components required to create a graph using the plot() function?

To create a graph using the plot() function in R, several essential components are required to customize and display the graph effectively. Here are the key components:

Essential Components for Creating a Graph with plot()

1. Data Input:

- The plot() function accepts various types of input data:
 - **Vectors:** For simple plots like scatter plots or line plots.
 - **Matrices and Data Frames:** For more complex plots where multiple variables are involved.

2. Basic Plotting:

- The basic syntax of plot() involves specifying the main data to be plotted on the x-axis (x) and the y-axis (y).

```
plot(x, y, ...)
```
- x and y can be numeric vectors representing coordinates or factors for categorical data.

3. Additional Components:

- Various arguments can be passed to plot() to enhance the appearance and information displayed on the plot:
 - **Type of Plot:** Specify the type of plot with type argument ("p" for points, "l" for lines, "b" for both, etc.).
 - **Labels and Titles:** Add labels to axes (xlab, ylab) and a title (main).
 - **Plotting Symbols and Colors:** Customize symbols (pch), line types (lty), and colors (col).
 - **Plotting Range:** Control the plotting range with xlim and ylim.

- **Grid Lines:** Add grid lines with `grid()`.
- **Legend:** Create a legend with `legend()`.

4. Customization:

- R provides extensive options for customizing plots to fit specific needs:

```
plot(x, y, type = "l", xlab = "X-axis", ylab = "Y-axis", main = "Title", col = "blue", pch = 16)
```

Example of Creating a Simple Plot

Let's create a simple scatter plot using the `plot()` function with some basic customization:

Example data

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

```
y <- c(1, 4, 9, 16, 25)
```

Basic scatter plot

```
plot(x, y,
     type = "b",      # 'b' for both points and lines
     pch = 16,        # Plotting character (solid circles)
     col = "blue",    # Color of points and lines
     xlab = "X-axis", # Label for x-axis
     ylab = "Y-axis", # Label for y-axis
     main = "Scatter Plot Example" # Title of the plot)
```

2.a) How do you customize the appearance of a graph using the `plot()` function in R?

Basic Customization Options

1. Plot Type (`type` argument):

- Specifies the type of plot. Common types include "p" for points, "l" for lines, "b" for both points and lines, "h" for high-density scatterplots, etc.

```
plot(x, y, type = "b") # Scatter plot with both points and lines
```

2. Plotting Symbols (`pch` argument):

- Specifies the plotting symbols for points.

```
plot(x, y, pch = 16) # Solid circles as plotting symbols
```


3. Colors (col argument):

- Specifies the color of points, lines, or both.

```
plot(x, y, col = "blue") # Blue color for points and lines
```

4. Line Types (lty argument):

- Specifies the line types for lines plotted (1 for solid, 2 for dashed, 3 for dotted, etc.).

```
plot(x, y, type = "l", lty = 2) # Dashed line plot
```

5. Labels and Titles:

- Customize axis labels (xlab, ylab) and the main title (main).

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

Additional Customization Options

6. Plotting Range (xlim, ylim):

- Set limits for the x-axis and y-axis.

```
plot(x, y, xlim = c(0, 10), ylim = c(0, 30))
```

7. Grid Lines (grid() function):

- Add grid lines to the plot.

```
plot(x, y)
```

```
grid()
```

8. Legend (legend() function):

- Add a legend to the plot.

```
plot(x, y, type = "b", col = c("blue", "red"), pch = c(16, 17))
```

```
legend("topright", legend = c("Series 1", "Series 2"), col = c("blue", "red"), pch = c(16, 17))
```

9. Text and Annotation (text() function):

- Add text annotations to specific points on the plot.

```
plot(x, y)
```

```
text(x, y, labels = c("Point 1", "Point 2", "Point 3", "Point 4", "Point 5"), pos = 3)
```

Example of Comprehensive Customization

```
# Example data
```

```
x <- 1:10
```

```
y <- x^2
```

```
# Customized plot
```

```

plot(x, y,
     type = "b",      # Points and lines
     pch = 16,        # Solid circles
     col = "blue",    # Blue color for points and lines
     lty = 2,         # Dashed lines
     xlim = c(0, 12), # X-axis limits
     ylim = c(0, 120), # Y-axis limits
     xlab = "X-axis",  # X-axis label
     ylab = "Y-axis",  # Y-axis label
     main = "Customized Plot Example" # Main title
)

# Add grid lines
grid()

# Add legend
legend("topright", legend = c("Data"), col = "blue", pch = 16)

# Add text annotation
text(5, 50, "Annotations", pos = 3)

```

2.b) Illustrate the concept of data visualization and its importance in data analysis.

Concept of Data Visualization and Its Importance in Data Analysis

1. What is Data Visualization?

Data visualization refers to the graphical representation of data and information using visual elements such as charts, graphs, and maps. It transforms raw data into visual formats that are easier to understand, interpret, and derive insights from. The goal of data visualization is to communicate complex data clearly and effectively.

2. Importance of Data Visualization in Data Analysis:

Data visualization plays a crucial role in the process of data analysis for several reasons:

- **Enhances Understanding:** Visual representations help to uncover patterns, trends, and relationships within data that may not be apparent in raw datasets. It provides a clearer understanding of the data's underlying structure and distribution.

- **Facilitates Communication:** Visualizations make it easier to communicate findings and insights to stakeholders, colleagues, and decision-makers who may not have technical expertise. Visuals can convey complex information succinctly and engage audiences more effectively.
- **Supports Decision Making:** Visualizations aid in decision-making processes by presenting data-driven insights in a format that is actionable. Whether it's identifying opportunities, understanding risks, or evaluating outcomes, visualizations provide valuable support for making informed decisions.
- **Detects Outliers and Anomalies:** Visual inspection of data through charts or plots can quickly reveal outliers, anomalies, or errors in data collection or processing. This aids in data cleaning and ensures the reliability of analysis results.
- **Enables Exploration and Discovery:** Interactive visualizations allow users to explore data from different angles, drill down into specific details, and dynamically adjust parameters. This fosters a deeper exploration of data and facilitates discovery of new insights.
- **Supports Storytelling:** Visualizations help to tell a compelling story about the data, making it more persuasive and memorable. They can illustrate the impact of trends, correlations, and predictions effectively, enhancing the narrative around data-driven insights.

3. Types of Data Visualizations:

Data can be visualized in various forms depending on the nature of the data and the insights being communicated:

- **Bar charts and Histograms:** Display categorical data or frequency distributions.
- **Line charts:** Show trends over time or relationships between variables.
- **Scatter plots:** Represent the relationship between two continuous variables.
- **Pie charts:** Display proportions or percentages of a whole.
- **Heatmaps and Geographic maps:** Visualize spatial distributions or patterns.
- **Box plots and Violin plots:** Illustrate distributions and variability in data.

4. Tools for Data Visualization:

- **R (ggplot2, plotly):** Widely used for creating static and interactive visualizations.
- **Python (Matplotlib, Seaborn):** Popular for generating a wide range of plots and statistical graphics.
- **Tableau, Power BI:** Tools for creating interactive dashboards and visual analytics.
- **Excel:** Basic charts and graphs for quick data exploration.

3.a) Discuss the advantages of using R base graphics for creating graphs.

Using R base graphics for creating graphs offers several advantages, making it a versatile choice for data visualization tasks. Here are the key advantages of using R base graphics:

Advantages of Using R Base Graphics

1. Ease to use:

R base graphics are straightforward and easy to learn, especially for beginners in R programming. The plotting functions (`plot()`, `hist()`, `boxplot()`, etc.) have simple syntax and intuitive parameters.

2. Integration with R:

Base graphics are part of the R core functionality, ensuring compatibility and reliability. They work seamlessly with other R functions and packages, facilitating smooth data analysis workflows.

3. Quick Prototyping:

Base graphics allow for rapid prototyping of plots. You can quickly generate standard plots such as scatter plots, histograms, and bar charts with minimal code, which is useful for exploratory data analysis.

4. Customization Control:

While base graphics may seem basic, they offer considerable customization options. You can control aspects such as colors, line types, plot symbols, axes labels, titles, and more using arguments within the plotting functions.

5. Low Overhead:

Base graphics have relatively low computational overhead compared to more complex plotting systems. They are efficient for rendering simple to moderately complex plots, making them suitable for handling medium-sized datasets.

6. Direct Interaction:

Base graphics allow direct interaction with plots. You can add elements interactively (e.g., text annotations, points, lines) or modify plot attributes directly within an active plotting window.

7. Availability and Stability:

Since base graphics are part of the R core, they are available by default with any R installation. They are stable and well-maintained, ensuring reliable performance across different operating systems and R versions.

Example of Using R Base Graphics

```
# Example data
x <- 1:10
y <- x^2
# Simple scatter plot
plot(x, y,
```

```

type = "b",      # Both points and lines
pch = 16,        # Solid circles as points
col = "blue",    # Blue color for points and lines
lty = 2,         # Dashed lines
xlim = c(0, 12), # X-axis limits
ylim = c(0, 120), # Y-axis limits
xlab = "X-axis", # X-axis label
ylab = "Y-axis", # Y-axis label
main = "Scatter Plot Example" # Main title
)

# Adding grid lines
grid()

# Adding text annotation
text(5, 50, "Annotations", pos = 3)

```

3.b) Categorize some common types of graphs that can be created using the plot() function.

The plot() function in R is versatile and can create a variety of common types of graphs to visualize different types of data. Here are some of the common types of graphs that can be created using the plot() function:

1. Scatter Plot:

- **Purpose:** Display the relationship between two continuous variables.
- **Function:** plot(x, y, type = "p") or plot(x, y, type = "b")
- **Example:**

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

```
y <- c(2, 4, 6, 8, 10)
```

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

2. Line Plot:

- **Purpose:** Show trends over time or continuous data points.

- **Function:** `plot(x, y, type = "l")`

- **Example:**

```
x <- 1:10
```

```
y <- x^2
```

```
plot(x, y, type = "l", main = "Line Plot Example", xlab = "X-axis", ylab = "Y-axis")
```

3. Bar Chart:

- **Purpose:** Display categorical data with rectangular bars.

- **Function:** `barplot(height, names.arg)`

- **Example:**

```
heights <- c(10, 20, 15, 25)
```

```
barplot(heights, names.arg = c("A", "B", "C", "D"), main = "Bar Chart Example",  
xlab = "Categories", ylab = "Values")
```

4. Histogram:

- **Purpose:** Visualize the distribution of continuous data.

- **Function:** `hist(x)`

- **Example:**

```
x <- rnorm(100, mean = 0, sd = 1)
```

```
hist(x, breaks = 10, main = "Histogram Example", xlab = "Values", ylab =  
"Frequency")
```

5. Boxplot:

- **Purpose:** Display the distribution of data across groups.

- **Function:** `boxplot(x ~ group)`

- **Example:**

```
x <- rnorm(100, mean = 0, sd = 1)
```

```
group <- rep(c("A", "B"), each = 50)
```

```
boxplot(x ~ group, main = "Boxplot Example", xlab = "Groups", ylab = "Values")
```

6. Pie Chart:

- **Purpose:** Show proportions or percentages of a whole.

- **Function:** `pie(x)`

- **Example:**

```
sizes <- c(20, 30, 10, 40)
```

```
pie(sizes, labels = c("Group 1", "Group 2", "Group 3", "Group 4"), main = "Pie Chart  
Example")
```

7. Density Plot:

- **Purpose:** Show the distribution of a continuous variable.
- **Function:** `plot(density(x))`
- **Example:**

```
x <- rnorm(100, mean = 0, sd = 1)
```

```
plot(density(x), main = "Density Plot Example", xlab = "Values", ylab = "Density")
```

8. Heatmap:

- **Purpose:** Visualize data in a matrix form using color gradients.
- **Function:** `heatmap(matrix)`
- **Example:**

```
mat <- matrix(rnorm(100), nrow = 10)
```

```
heatmap(mat, main = "Heatmap Example")
```

4.a) How do you add labels to the axes of a graph created with the `plot()` function?

Adding labels to the axes of a graph created with the `plot()` function in R is straightforward and involves using the `xlab` and `ylab` arguments within the `plot()` function. These arguments allow you to specify the labels for the x-axis and y-axis, respectively.

Syntax:

```
plot(x, y, xlab = "X-axis Label", ylab = "Y-axis Label")
```

- `x`: Data for the x-axis.
- `y`: Data for the y-axis.
- `xlab`: Label for the x-axis.
- `ylab`: Label for the y-axis.

Example:

Let's create a simple scatter plot and add labels to the axes:

```
# Example data
```

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

```
y <- c(2, 4, 6, 8, 10)
```

```
# Plot with axis labels
```

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

Output:

This code will generate a scatter plot with labeled axes:

- The x-axis will have the label "X-axis Label".
- The y-axis will have the label "Y-axis Label".
- The title of the plot will be "Scatter Plot Example".

4.b) Describe the process of changing the color and line type of a plot in R.

In R, you can change the color and line type of a plot using specific arguments within the `plot()` function or by modifying properties after the plot is created. Here's a step-by-step guide on how to change the color and line type of a plot:

1. Changing Color:

To change the color of elements in a plot, such as points, lines, or text, you can use the `col` argument within the `plot()` function or other plotting functions.

- **Using `col` Argument:**

The `col` argument can be set to a color name (e.g., "red", "blue"), a hexadecimal color code (e.g., "#FFA500" for orange), or an integer specifying a color palette index.

Example: Changing color of points in a scatter plot

```
x <- 1:10
y <- x^2
plot(x, y, col = "blue", main = "Scatter Plot with Blue Points")
```

- **Changing Color After Plot Creation:**

You can also change the color of specific elements after the plot is created using functions like `points()`, `lines()`, or by directly modifying graphical parameters.

Example: Changing color of points after plot creation

```
plot(x, y)
points(x, y, col = "red")
```

2. Changing Line Type:

To change the line type (e.g., solid, dashed, dotted) of lines in a plot, you can use the `lty` argument within the `plot()` function or other plotting functions.

- **Using `lty` Argument:**

- The `lty` argument accepts integer values representing different line types:
 - `lty = 1`: Solid line (default).

- lty = 2: Dashed line.
- lty = 3: Dotted line.
- lty = 4: Dot-dashed line, etc.

Example: Changing line type in a line plot

```
x <- 1:10
```

```
y <- x^2
```

```
plot(x, y, type = "l", lty = 2, col = "green", main = "Dashed Line Plot")
```

- **Changing Line Type After Plot Creation:**

Similar to changing color, you can modify the line type of specific lines using functions like lines() or by adjusting graphical parameters directly.

Example: Changing line type after plot creation

```
plot(x, y, type = "l", col = "blue")
```

```
lines(x, y, lty = 3, col = "red")
```

5.a) Explain the significance of customizing the title of a graph in data visualization.

Customizing the title of a graph in data visualization is significant for several reasons, primarily related to enhancing clarity, context, and communication of the information conveyed by the graph. Here are the key reasons why customizing the title of a graph is important:

1. Providing Context and Purpose:

- **Clarity of Message:** A well-crafted title succinctly summarizes the main message or purpose of the graph. It tells viewers what the graph is about and what insights they can expect to gain from it.
- **Focus and Orientation:** It helps viewers quickly orient themselves to the content of the graph, guiding their attention to the specific aspects of the data being presented.

2. Enhancing Interpretation:

- **Interpretation Guidance:** The title can provide interpretive cues that help viewers understand the data in a specific context or from a particular perspective.
- **Highlighting Key Insights:** A descriptive title can highlight the main findings or insights derived from the data, making it easier for viewers to grasp the significance of the information presented.

3. Communicating Results Effectively:

- **Communication Tool:** In data analysis and presentation, graphs serve as visual aids to convey complex information. A clear and informative title supports this process by framing the content and guiding the viewer's understanding.

- **Storytelling and Narrative:** The title contributes to the narrative of the data story being told. It sets expectations and prepares viewers for the analysis and conclusions drawn from the visual representation.

Best Practices for Customizing Graph Titles:

- **Be Clear and Specific:** Use concise language that accurately reflects the content and purpose of the graph. Avoid ambiguous or vague titles that could lead to misinterpretation.
- **Include Relevant Details:** Depending on the complexity of the graph, consider including additional information such as time periods, data sources, or specific variables being analyzed.
- **Use Proper Formatting:** Ensure the title is appropriately formatted with proper capitalization and punctuation for readability and professionalism.

Example of Customizing a Graph Title:

```
# Example data
x <- 1:10
y <- x^2

# Plot with customized title
plot(x, y, main = "Relationship between X and Y")

# Adding axis labels
xlabel <- "X-axis Label"
ylabel <- "Y-axis Label"
title <- "Scatter Plot Example"
plot(x, y, xlab = xlabel, ylab = ylabel, main = title)
```

5.b) Assess the use of the main parameter in the plot() function for adding a title to a graph.

The main parameter in the plot() function is used to add a title to a graph in R. This parameter is essential for providing context and a brief summary of the graph's content, making it easier for viewers to understand the purpose and focus of the visualization.

Key Points About the main Parameter

1. **Purpose:**
 - Adds a main title to the graph, which is displayed at the top of the plot area.

- Helps in quickly conveying the main message or context of the graph to the viewer.

2. Usage:

- The main parameter is included as an argument within the plot() function.
- The value of main should be a character string that describes the graph.

3. Syntax:

```
plot(x, y, main = "Title of the Graph")
```

Example:

Here's a simple example of using the main parameter to add a title to a scatter plot:

```
# Example data
```

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

```
y <- c(2, 4, 6, 8, 10)
```

```
# Plot with main title
```

```
plot(x, y, main = "Relationship between X and Y", xlab = "X-axis", ylab = "Y-axis")
```

In this example, the title "Relationship between X and Y" will appear at the top of the graph, providing immediate context to the viewer about what the graph represents.

Using the main parameter in the plot() function is a straightforward and effective way to enhance the clarity and impact of your graphs in R. It ensures that viewers can quickly grasp the key message and context of the data visualization.

6.a) Illustrate Data visualization with R and ggplot2.

Data visualization in R can be achieved through various methods, but one of the most powerful and flexible packages for this purpose is ggplot2. ggplot2 is part of the Tidyverse suite of packages and provides a robust framework for creating complex and aesthetically pleasing visualizations.

Key Features of ggplot2:

- **Grammar of Graphics:** ggplot2 is based on the "grammar of graphics," which allows you to build plots layer by layer, combining data, aesthetic mappings, and geometric objects.
- **Flexibility:** It offers extensive customization options for all plot elements, including scales, themes, and labels.
- **Consistency:** The consistent syntax and framework make it easy to create a wide variety of plots with minimal code changes.

Installing and Loading ggplot2

To use ggplot2, you need to install and load the package:

```
install.packages("ggplot2")  
library(ggplot2)
```

Basic Structure of ggplot2

A ggplot2 plot is constructed using the `ggplot()` function, which initializes the plotting object. This is followed by adding layers using the `+` operator. Common layers include `geom_point()`, `geom_line()`, `geom_bar()`, and many others.

Example: Scatter Plot with ggplot2

Let's create a scatter plot to illustrate the relationship between two continuous variables, `x` and `y`.

```
# Example data  
x <- 1:10  
y <- x^2  
data <- data.frame(x, y)  
  
# Basic scatter plot  
ggplot(data, aes(x = x, y = y)) +  
  geom_point() +  
  labs(title = "Scatter Plot Example",  
        x = "X-axis Label",  
        y = "Y-axis Label")
```

Adding Layers and Customization

1. Adding a Line of Best Fit:

```
ggplot(data, aes(x = x, y = y)) +  
  geom_point(color = "blue") +  
  geom_smooth(method = "lm", se = FALSE, color = "red") +  
  labs(title = "Scatter Plot with Line of Best Fit", x = "X-axis Label", y = "Y-axis Label")
```

2. Customizing Colors and Themes:

```
ggplot(data, aes(x = x, y = y)) +  
  geom_point(color = "blue", size = 3) +
```

```
theme_minimal() +  
labs(title = "Customized Scatter Plot",  
      x = "X-axis Label",  
      y = "Y-axis Label")
```

Example: Bar Plot with ggplot2

Let's create a bar plot to display the frequency of categorical data.

```
# Example data  
categories <- c("A", "B", "C", "D")  
values <- c(10, 20, 15, 25)  
bar_data <- data.frame(categories, values)  
  
# Basic bar plot  
ggplot(bar_data, aes(x = categories, y = values)) +  
  geom_bar(stat = "identity", fill = "skyblue") +  
  labs(title = "Bar Plot Example",  
        x = "Categories",  
        y = "Values")
```

Example: Histogram with ggplot2

Creating a histogram to visualize the distribution of a continuous variable.

```
# Example data  
values <- rnorm(100, mean = 0, sd = 1)  
hist_data <- data.frame(values)  
  
# Basic histogram  
ggplot(hist_data, aes(x = values)) +  
  geom_histogram(binwidth = 0.5, fill = "green", color = "black") +  
  labs(title = "Histogram Example",  
        x = "Values",  
        y = "Frequency")
```

Example: Box Plot with ggplot2

Creating a box plot to visualize the distribution of values across different categories.

```
data
```

```
categories <- rep(c("A", "B"), each = 50)
values <- c(rnorm(50, mean = 0, sd = 1), rnorm(50, mean = 1, sd = 1))
box_data <- data.frame(categories, values)
```

```
# Basic box plot
ggplot(box_data, aes(x = categories, y = values)) +
  geom_boxplot(fill = "purple", color = "black") +
  labs(title = "Box Plot Example",
        x = "Categories",
        y = "Values")
```

6.b) Identify the different options available for adjusting the size and aspect ratio of a graph in R.

In R, you have several options for adjusting the size and aspect ratio of a graph. Here are the main methods:

1. Using par() Function:

The par() function allows you to set graphical parameters, including size and aspect ratio, for base R graphics.

- **Aspect Ratio:**

```
par(pty = "s") # Square plot
```

```
par(pty = "m") # Default, maximized plot
```

- **Plot Margins:**

```
par(mar = c(bottom, left, top, right)) # Set margins in lines of text
```

2. Using dev.new() or windows() Function:

These functions open a new graphics device with specified width and height.

- **Size:**

```
dev.new(width = 7, height = 5) # Width and height in inches
```

3. Using ggplot2:

In ggplot2, you can adjust size and aspect ratio using the theme() function and the aspect.ratio argument.

- **Size and Aspect Ratio:**

```
library(ggplot2)
```

```
p <- ggplot(data, aes(x, y)) + geom_point()
p + theme(aspect.ratio = 1) # Square aspect ratio
```

- **Saving Plots with Specified Size:**

```
ggsave("plot.png", plot = p, width = 7, height = 5, units = "in") # Width and height in inches
```

4. Using R Markdown or RStudio:

In R Markdown or RStudio, you can adjust plot size within code chunks.

- **Chunk Options:**

```
markdown
```{r, fig.width = 7, fig.height = 5}
plot(x, y)
```

#### 7.a) Outline the process of saving a graph created with the plot() function to a file in R.

**Saving Graphs:** The R graphics display can consist of various graphics devices. The default device is the screen. In order to save a graph to a file, you must set up another device.

- ❖ The graph can be saved in a variety of formats from the menu File -> Save As.
- ❖ The graph can also be saved using one of the following functions.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpeg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Let's go through the basics of R graphics devices first to introduce R graphics device concepts, and then discuss a second approach that is much more direct and convenient.

```
pdf("d12.pdf")
```

This opens the file d12.pdf. We now have two devices open, as we can confirm:

```
dev.list()
X11 pdf
2 3
```

The screen is named X11 when R runs on Linux. (It's named windows on Windows systems.) It is device number 2 here. Our PDF file is device number 3. Our active device is the PDF file:

```
dev.cur() pdf
```

```
3
```

All graphics output will now go to this file instead of to the screen. But what if we wish to save what's already on the screen?

**Saving the Displayed Graph:** One way to save the graph currently displayed on the screen is to reestablish the screen as the current device and then copy it to the PDF device, which is 3 in our example, as follows:

```
dev.set(2)
X11
2
dev.copy(which=3)
pdf
3
```

But actually, it is best to set up a PDF device as shown earlier and then rerun whatever analyses led to the current screen. This is because the copy operation can result in distortions due to mismatches between screen devices and file devices.

## 7.b) Discover the purpose of the file parameter in the plot() function for saving graphs.

In R, the plot() function itself does not have a file parameter for directly saving graphs. Instead, saving graphs is typically handled through different methods or functions depending on the type of plot and the plotting package being used (base R graphics or ggplot2).



## **Saving Graphs in Base R Graphics:**

For base R graphics created using functions like `plot()`, `hist()`, `barplot()`, etc., you can save the current graphical device (plot) using functions like `dev.copy()`, `dev.print()`, or `pdf()`, `png()`, `jpeg()`.

### **Here's a basic approach to save a base R plot:**

# Example plot

```
plot(x, y)
```

# Save as PDF

```
pdf("plot.pdf")
```

```
dev.copy(pdf)
```

```
dev.off()
```

# Save as PNG

```
png("plot.png", width = 800, height = 600, units = "px")
```

```
dev.copy(png)
```

```
dev.off()
```

## **Saving Graphs in ggplot2:**

For graphs created using `ggplot2`, you can use the `ggsave()` function, which allows you to save the last created `ggplot` object to a file.

# Example ggplot2 plot

```
library(ggplot2)
```

```
p <- ggplot(data, aes(x, y)) + geom_point()
```

# Save as PDF

```
ggsave("plot.pdf", plot = p, width = 7, height = 5, units = "in")
```

# Save as PNG

```
ggsave("plot.png", plot = p, width = 800, height = 600, dpi = 300)
```

## **8.a) List and explain about different R – Charts and Graphs in R.**

### **R Charts and Graphs:**

R offers a powerful set of tools for creating various charts and graphs to visualize your data. Here's a breakdown of some common chart types with examples:

#### **1. Bar Charts:**

- **Purpose:** Compare values across categories.

- **Program:**

```
Sample data
```

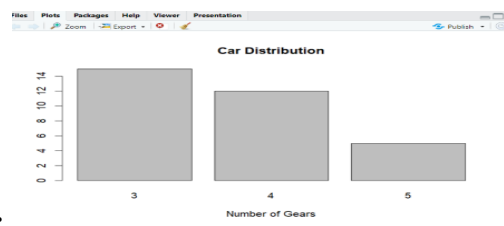
```
data <- data.frame(category = c("A", "B", "C"), value = c(20, 35, 15))
```

```
Basic bar chart
```

```
barplot(data$value, names.arg = data$category)
```

```
Customize with colors and title
```

```
barplot(data$value, names.arg = data$category, col = c("red", "green", "blue"), main = "Value by Category")
```



- **Example Diagram:**

## 2. Pie Charts:

- **Purpose:** Show proportions of a whole.

- **Program:**

```
Sample data
```

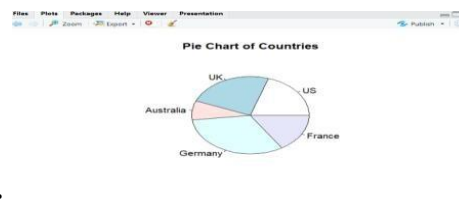
```
data <- c(20, 35, 15)
```

```
Basic pie chart
```

```
pie(data)
```

```
Customize with labels and colors
```

```
pie(data, labels = c("Category A", "Category B", "Category C"), col = c("red", "green", "blue"))
```



- **Example Diagram:**

## 3. Histograms:

- **Purpose:** Visualize the distribution of continuous data.

- **Program:**

```
Sample data
```

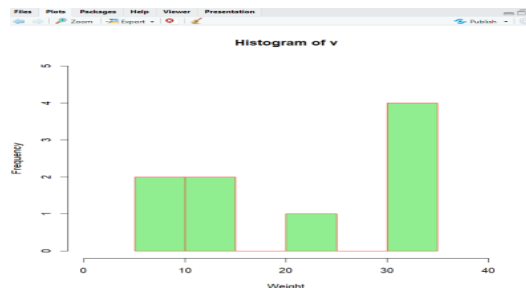
```
data <- rnorm(100) # Simulate normal distribution
```

```
Basic histogram
```

```
hist(data)
```

```
Customize with bins and title
```

```
hist(data, breaks = 20, main = "Distribution of Data")
```



- **Example Diagram:**

#### 4. Scatter Plots:

- **Purpose:** Explore relationships between two continuous variables.

- **Program:**

```
Sample data
```

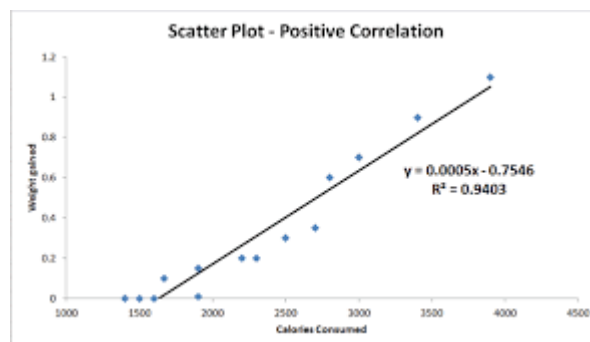
```
data <- data.frame(x = rnorm(100), y = rnorm(100))
```

```
Basic scatter plot
```

```
plot(data$x, data$y)
```

```
Customize with labels and title
```

```
plot(data$x, data$y, xlab = "Variable X", ylab = "Variable Y", main = "Relationship between X and Y")
```



- **Example Diagram:**

#### 5. Box Plots:

- **Purpose:** Summarize the distribution of a numeric variable, showing quartiles and outliers.

- **Program:**

```
Sample data
```

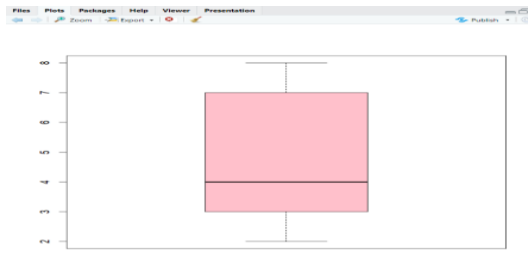
```
data <- rnorm(100)
```

```
Basic box plot
```

```
boxplot(data)
```

```
Customize with title
```

```
boxplot(data, main = "Distribution of Data (Box Plot)")
```



- **Example Diagram:**

### 8.b) Discriminate the importance of choosing an appropriate file format when saving a graph in R.

Choosing an appropriate file format when saving a graph in R is important for several reasons:

1. **Quality:** Different formats offer varying levels of quality and resolution. For example, vector formats like SVG or PDF maintain quality at any size, while raster formats like PNG or JPEG may lose clarity when scaled.
2. **Usability:** The intended use of the graph affects the choice. For web use, PNG or JPEG might be preferred, whereas for print, high-resolution PDF or TIFF files are better.
3. **File Size:** Some formats, like JPEG, compress the image and result in smaller file sizes, which is useful for web and email. Others, like TIFF, result in larger files, suitable for high-quality print purposes.
4. **Compatibility:** Ensuring the format is compatible with the software or platform where the graph will be used is crucial. For instance, certain journals or publishers might require specific formats.
5. **Editing:** If further editing is needed, vector formats (e.g., SVG) are more suitable as they allow for easier and more flexible modifications compared to raster formats.

### 9.a) How do you specify the dimensions and resolution of a saved graph in R?

In R, you can specify the dimensions and resolution of a saved graph by using the appropriate arguments in the graphics device functions such as `png()`, `jpeg()`, `tiff()`, `bmp()`, or `pdf()`. Here's how:

1. **Dimensions:** Use the width and height arguments to set the dimensions of the graph in inches.
2. **Resolution:** Use the `res` argument to set the resolution in dots per inch (DPI) for raster formats.

**Example for saving a PNG file:**

```
png("plot.png", width = 6, height = 4, units = "in", res = 300)
```

or

```
jpeg("filename.jpg", width = 800, height = 600, res = 300)
```

```
plot(x, y) # Your plotting code here
```

```
dev.off()
```

**Example for saving a PDF file** (resolution is inherently high for vector formats, so DPI is not needed):

```
pdf("plot.pdf", width = 6, height = 4)
```

```
plot(x, y) # Your plotting code here
```

```
dev.off()
```

**TIFF:**

```
tiff("filename.tiff", width = 800, height = 600, res = 300)
```

```
plot(y ~ x, data = df)
```

```
dev.off()
```

**9.b) Describe the process of exporting a graph to different file formats using the plot() function.**

To export a graph to different file formats using the plot() function in R, follow these steps:

1. **Open a Graphics Device:** Use a function corresponding to the desired file format to open a graphics device (e.g., png(), jpeg(), tiff(), pdf()). Specify the file name and other parameters such as dimensions and resolution.
2. **Create the Plot:** Generate the plot using the plot() function (or any other plotting function).
3. **Close the Graphics Device:** Use dev.off() to close the graphics device and finalize the file.

**Here's an example for each format:****PNG:**

```
png("plot.png", width = 800, height = 600, res = 300)
```

```
plot(y ~ x, data = df)
```

```
dev.off()
```

**JPEG:**

```
jpeg("plot.jpg", width = 800, height = 600, res = 300)
```

```
plot(y ~ x, data = df)
```

```
dev.off()
```

#### **TIFF:**

```
tiff("plot.tiff", width = 800, height = 600, res = 300)
```

```
plot(y ~ x, data = df)
```

```
dev.off()
```

#### **PDF:**

```
pdf("plot.pdf", width = 8, height = 6)
```

```
plot(y ~ x, data = df)
```

```
dev.off()
```

This process ensures your graph is saved in the desired format with specified dimensions and resolution.

### **10. Develop a code to demonstrate various charts using tree datasets for the following**

#### **a. Histogram**

#### **b. Scatter plot**

#### **c. Box plot**

#### **d. Line chart**

#### **Histogram:**

A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chart but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

R creates histogram using **hist()** function. This function takes a vector as an input and uses some more parameters to plot histograms.

#### **Syntax**

The basic syntax for creating a histogram using R is –

```
hist(v,main,xlab,xlim,ylim,breaks,col,border)
```

Following is the description of the parameters used –

- **v** is a vector containing numeric values used in histogram.
- **main** indicates title of the chart.
- **col** is used to set color of the bars.
- **border** is used to set border color of each bar.

- **xlab** is used to give description of x-axis.
- **xlim** is used to specify the range of values on the x-axis.
- **ylim** is used to specify the range of values on the y-axis.
- **breaks** is used to mention the width of each bar.

### Example:

A simple histogram is created using input vector, label, col and border parameters.

The script given below will create and save the histogram in the current R working directory.

Live Demo

```
Create data for the graph.
```

```
v <- c(9,13,21,8,36,22,12,41,31,33,19)
```

```
Give the chart file a name.
```

```
png(file = "histogram.png")
```

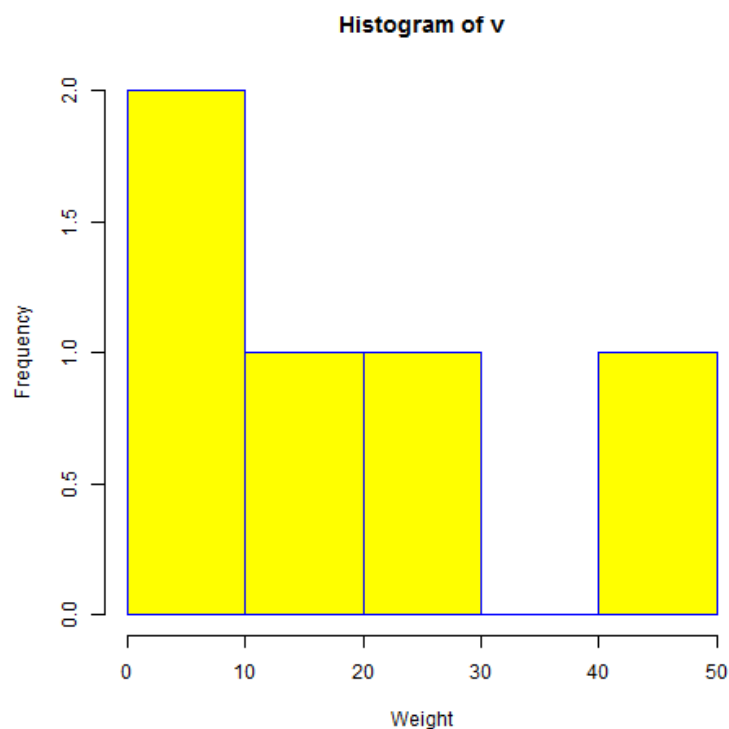
```
Create the histogram.
```

```
hist(v,xlab = "Weight",col = "yellow",border = "blue")
```

```
Save the file.
```

```
dev.off()
```

When we execute the above code, it produces the following result –



## Scatterplot:

Scatterplots show many points plotted in the Cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and another in the vertical axis.

The simple scatterplot is created using the **plot()** function.

## Syntax:

The basic syntax for creating scatterplot in R is –

```
plot(x, y, main, xlab, ylab, xlim, ylim, axes)
```

Following is the description of the parameters used –

- **x** is the data set whose values are the horizontal coordinates.
- **y** is the data set whose values are the vertical coordinates.
- **main** is the title of the graph.
- **xlab** is the label in the horizontal axis.
- **ylab** is the label in the vertical axis.
- **xlim** is the limits of the values of x used for plotting.
- **ylim** is the limits of the values of y used for plotting.
- **axes** indicates whether both axes should be drawn on the plot.

## Example:

We use the data set "**mtcars**" available in the R environment to create a basic scatterplot. Let's use the columns "wt" and "mpg" in mtcars.

Live Demo

```
input <- mtcars[,c('wt','mpg')]
print(head(input))
```

When we execute the above code, it produces the following result –

	wt	mpg
Mazda RX4	2.620	21.0
Mazda RX4 Wag	2.875	21.0
Datsun 710	2.320	22.8
Hornet 4 Drive	3.215	21.4
Hornet Sportabout	3.440	18.7
Valiant	3.460	18.1



### Creating the Scatterplot:

The below script will create a scatterplot graph for the relation between wt(weight) and mpg(miles per gallon).

Live Demo

# Get the input values.

```
input <- mtcars[,c('wt','mpg')]
```

# Give the chart file a name.

```
png(file = "scatterplot.png")
```

# Plot the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.

```
plot(x = input$wt,y = input$mpg,
```

```
 xlab = "Weight",
```

```
 ylab = "Milage",
```

```
 xlim = c(2.5,5),
```

```
 ylim = c(15,30),
```

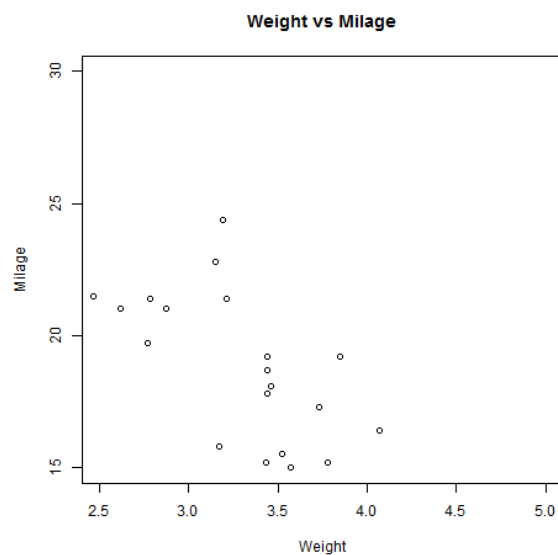
```
 main = "Weight vs Milage"
```

```
)
```

# Save the file.

```
dev.off()
```

When we execute the above code, it produces the following result –



## Box plot:

Boxplots are a measure of how well distributed is the data in a data set. It divides the data set into three quartiles. This graph represents the minimum, maximum, median, first quartile and third quartile in the data set. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.

Boxplots are created in R by using the **boxplot()** function.

## Syntax:

The basic syntax to create a boxplot in R is –

```
boxplot(x, data, notch, varwidth, names, main)
```

Following is the description of the parameters used –

- **x** is a vector or a formula.
- **data** is the data frame.
- **notch** is a logical value. Set as TRUE to draw a notch.
- **varwidth** is a logical value. Set as true to draw width of the box proportionate to the sample size.
- **names** are the group labels which will be printed under each boxplot.
- **main** is used to give a title to the graph.

## Example:

We use the data set "mtcars" available in the R environment to create a basic boxplot. Let's look at the columns "mpg" and "cyl" in mtcars.

Live Demo

```
input <- mtcars[,c('mpg','cyl')]
print(head(input))
```

When we execute above code, it produces following result –

	mpg	cyl
Mazda RX4	21.0	6
Mazda RX4 Wag	21.0	6
Datsun 710	22.8	4
Hornet 4 Drive	21.4	6
Hornet Sportabout	18.7	8
Valiant	18.1	6

### Creating the Boxplot:

The below script will create a boxplot graph for the relation between mpg (miles per gallon) and cyl (number of cylinders).

Live Demo

# Give the chart file a name.

```
png(file = "boxplot.png")
```

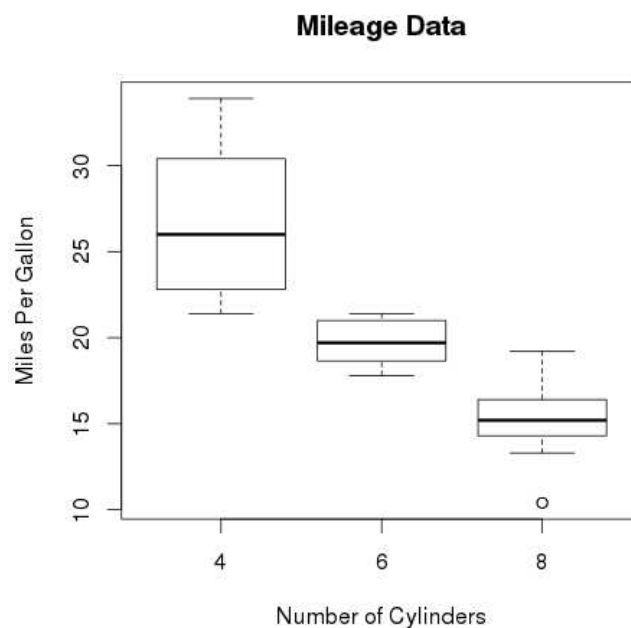
# Plot the chart.

```
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders",
 ylab = "Miles Per Gallon", main = "Mileage Data")
```

# Save the file.

```
dev.off()
```

When we execute the above code, it produces the following result –



### Line Graph:

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

#### Syntax:

The basic syntax to create a line chart in R is –

```
plot(v,type,col,xlab,ylab)
```

Following is the description of the parameters used –

- **v** is a vector containing the numeric values.
- **type** takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the Title of the chart.
- **col** is used to give colors to both the points and lines.

### Example:

A simple line chart is created using the input vector and the type parameter as "O". The below script will create and save a line chart in the current R working directory.

Live Demo

```
Create the data for the chart.
```

```
v <- c(7,12,28,3,41)
```

```
Give the chart file a name.
```

```
png(file = "line_chart.jpg")
```

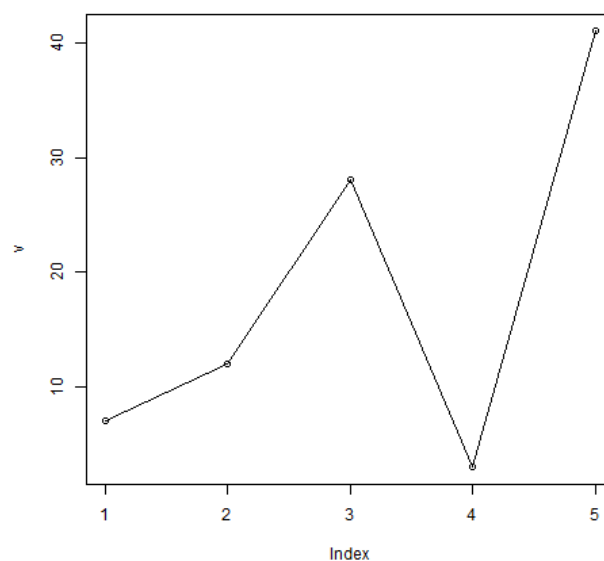
```
Plot the bar chart.
```

```
plot(v,type = "o")
```

```
Save the file.
```

```
dev.off()
```

When we execute the above code, it produces the following result –



## Unit – V

### PROBABILITY DISTRIBUTIONS AND LINEAR REGRESSION

#### **2.b) Discuss about standard deviation with example.**

**Standard deviation** is the square root of variance and is calculated using `sd()`. Like mean and var, `sd` has the `na.rm` argument to remove NAs before computation; otherwise, any NAs will cause the answer to be NA.

A low standard deviation indicates that the values tend to be close to the mean, while a high standard deviation indicates that the values are spread out over a wider range.

The formula used to calculate the standard deviation:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

where:

- N is the number of observations
- $x_i$  is each individual observation
- $\mu$  is the mean of the observations

#### **Syntax:**

`sd(vector)`

#### **Example:**

```
x <- c(95, 72, 87, 66)
```

```
y <- sqrt(var(x))
```

```
print(y)
```

#### **Output:**

```
[1] 13.34166
```

```
heights <- c(60, 62, 65, 67, 68, 70, 72, 74, 75, 77)
```

```
calculate the standard deviation of the dataset
```

```
std_dev_height <- sd(heights)
```

```
print the result
```

```
print(std_dev_height)
```

**Output:**

```
[1] 5.637178
```

**3.a) Determine basic statistics and explain their importance in data analysis?****Basic statistics:**

Some of the most common tools used in statistics are means, variances, correlations and t-tests. These are all well represented in R with easy-to-use functions such as `mean()`, `var()`, `cor()` and `t.test()`.

**Mean():**

The mean is the sum of all values in a dataset divided by the number of values.

**Example:**

```
> x <- c(60, 62, 65, 67, 68, 70, 72, 74)
```

```
> mean(x)
```

```
[1] 67.25
```

**Median():**

The median is the middle value of a dataset when the values are arranged in ascending order.

**Example:**

```
> x <- c(60, 62, 65, 67, 68, 70, 72, 74)
```

```
> median(x)
```

```
[1] 67.5
```

**Mode():**

The mode is the value that occurs most frequently in a dataset.

**Variance():**

The variance is the average of the squared deviations from the mean.

**Example:**

```
> x <- c(60, 62, 65, 67, 68, 60, 72, 74)
```

```
> var(x)
```

[1] 27.71429

### **Percentiles:**

Percentiles indicate the value below which a given percentage of observations fall.

### **Example:**

```
> x <- c(60, 62, 65, 67, 68, 60, 72, 74)
```

```
> quantile(x, probs = c(0.25, 0.5, 0.75))
```

25% 50% 75%

61.5 66.0 69.0

### **3.b) Illustrate the concepts of correlation and covariance and explain how they are calculated.**

Both correlation and covariance are measures used to assess the relationship between two variables in a dataset. While they provide similar information, there are key distinctions between them.

### **Correlation:**

Correlation is a normalized measure of the linear relationship between two variables. It is dimensionless and ranges from -1 to 1. `cor()` function is used to calculate the correlation in R.

- A correlation of **+1** indicates a perfect positive linear relationship.
- A correlation of **-1** indicates a perfect negative linear relationship.
- A correlation of **0** implies no linear relationship between the variables.

### **Covariance:**

Covariance (`cov`) captures the direction and strength of the linear relationship between two variables. `cov()` function is used to calculate covariance in R.

- A positive covariance indicates the variables tend to move in the same direction (i.e., both increase or decrease together).
- A negative covariance signifies the variables move in opposite directions (i.e., one increases while the other decreases).

### **Example:**

```
Sample dataset
```

```
x <- c(3, 5, 6, 9, 11)
```

```
y <- c(13, 24, 26, 29, 30)
```

```
Calculate covariance
```

```
covariance <- cov(x, y)
```

```
Calculate correlation
```

```
correlation <- cor(x, y)
```

```
Print the results
```

```
print(covariance)
```

```
print(correlation)
```

**Output:**

```
[1] 19.1
```

```
[1] 0.8789067
```

**4.a) Analyze the purpose of T-tests in statistical analysis and provide examples of when they are used.**

**T-Tests:**

T-tests are a fundamental tool in statistical analysis used for hypothesis testing. They specifically focus on comparing the means of two groups or populations. They are particularly useful when dealing with small sample sizes and are based on the t-distribution.

**Types of T-tests:**

**One-sample t-test:**

To compare the mean of a single sample to a known value or population mean.

**Two-sample (independent) t-test:**

Compares the means of two independent groups.

**Paired (Dependent) T-test:**

To compare the means of two related groups.

**Examples when T-tests are used:**

**Effectiveness of a new drug:** A researcher might use a t-test to compare the mean recovery time of patients who received a new drug versus a control group receiving a standard treatment.

**Impact of a training program:** A company might use a t-test to compare the average scores on a skills assessment before and after implementing a new training program for employees.

**Differences in student performance:** An educator might use a paired t-test to compare the average exam scores of students before and after a specific teaching method.



**Consumer preference studies:** Marketers might use a t-test to compare the average ratings of two different product prototypes by consumers.

**4.b) Distinguish the use of ANOVA (Analysis of Variance) in comparing means across multiple groups.**

Feature	T-test	ANOVA (Analysis of Variance)
Number of Groups Compared	Two	Three or more
Purpose	Assess if the means of two groups differ	Assess if the means of multiple groups differ (due to the factor being studied or random chance)
Hypothesis Testing	Null hypothesis: Means are equal	Null hypothesis: All group means are equal
Underlying Assumption	Data is normally distributed	Data is normally distributed (although some variations are more robust to violations)
Test Statistic	t-statistic	F-statistic
Output	p-value indicating significance of difference	p-value indicating significance of <b>overall</b> difference between groups
Post-hoc Analysis	Limited (e.g., Bonferroni correction)	Possible using post-hoc tests (e.g., Tukey's HSD) to identify specific groups with significant differences
Strengths	Simple to understand and implement	Efficient for multiple comparisons, provides a holistic view
Weaknesses	Limited to two groups	More complex than t-tests, assumes normality

**5.a) Examine the concept of linear models and give an example of simple linear regression.**

#### **Linear Models:**

Linear models are a fundamental concept in statistics and machine learning. They represent the relationship between a **dependent variable** (what you're trying to predict) and one or more **independent variables** (thought to influence the dependent variable) using a **linear equation**.

#### **Simple Linear Regression:**

**Simple linear regression** is the most basic form of linear models. It involves two variables: one independent (predictor) variable and one dependent (response) variable. The goal is to find the linear relationship between these two variables.

The equation for simple linear regression is:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where:

- Y is the dependent variable.
- X is the independent variable.
- $\beta_0$  is the y-intercept (the value of Y when X=0).
- $\beta_1$  is the slope of the line (the change in Y for a one-unit change in X).
- $\epsilon$  is the error term (the difference between the observed and predicted values).

**Example:**

```
Sample data
```

```
age <- c(5, 6, 7, 8, 9, 10)
```

```
height <- c(105, 108, 112, 115, 118, 120)
```

```
Create a data frame
```

```
data <- data.frame(age, height)
```

```
Plot the data
```

```
plot(data$age, data$height, main = "Height vs Age", xlab = "Age (years)", ylab = "Height (cm)")
```

```
Fit the simple linear regression model
```

```
model <- lm(height ~ age, data = data)
```

```
Summary of the model
```

```
summary(model)
```

```
Add the regression line to the plot
```

```
abline(model, col = "blue")
```

```
print(summary(model))
```

**Output:**

Call:

```
lm(formula = height ~ age, data = data)
```

Residuals:

```
 1 2 3 4 5 6
-0.2857 -0.3714 0.5429 0.4571 0.3714 -0.7143
```

Coefficients:

```
 Estimate Std. Error t value Pr(>|t|)
(Intercept) 89.857 1.077 83.46 1.24e-07 ***
age 3.086 0.140 22.05 2.51e-05 ***
```

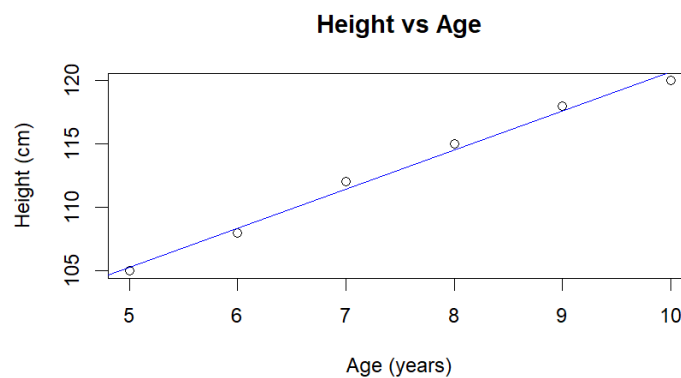
---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5855 on 4 degrees of freedom

Multiple R-squared: 0.9918, Adjusted R-squared: 0.9898

F-statistic: 486 on 1 and 4 DF, p-value: 2.506e-05



### 5.b) Describe the process of multiple regression and its applications.

#### Multiple regression:

**Multiple regression** is an extension of simple linear regression. It models the relationship between a dependent variable and two or more independent variables. This approach allows for more complex models that can account for multiple factors influencing the dependent variable.

### Equation of multiple regression:

The general form of a multiple regression model is:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_k X_k + \epsilon$$

where:

- Y is the dependent variable.
- $X_1, X_2, \dots, X_k$  are the independent variables.
- $\beta_0$  is the intercept.
- $\beta_1, \beta_2, \dots, \beta_k$  are the coefficients for the independent variables.
- $\epsilon$  is the error term.

### Steps in Multiple Regression:

**Data Collection:** Gather data for the dependent and independent variables.

**Data Preparation:** Clean the data, handle missing values, and ensure the data types are appropriate.

**Exploratory Data Analysis (EDA):** Visualize the data, understand relationships, and check assumptions.

**Fit the Model:** Use statistical software to fit the multiple regression model.

**Interpret Results:** Evaluate the coefficients, significance levels, and goodness-of-fit measures.

**Model Diagnostics:** Check for violations of assumptions (linearity, independence, homoscedasticity, and normality).

**Prediction:** Use the model for prediction and inference.

### Applications of Multiple regression:

- **Marketing:** Analyze the impact of advertising spending, product features, and pricing on sales.
- **Finance:** Model stock prices based on various economic indicators and company performance metrics.
- **Ecology:** Understand the relationship between environmental factors (e.g., temperature, rainfall) and plant growth or animal population.
- **Social Sciences:** Examine how factors like education level, income, and social background influence social mobility.

## 6.a) Discuss generalized linear models and their advantages over traditional linear models.

### Generalized linear models:

**Generalized Linear Models (GLMs)** extend traditional linear models to allow for response variables that have error distribution models other than a normal distribution. GLMs are particularly useful when the assumptions of traditional linear models (such as normally distributed errors, homoscedasticity, and linearity) do not hold.

### Common types of GLMs:

1. **Logistic Regression:** Used for binary response data. The link function is the logit function.

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

2. **Poisson Regression:** Used for count data. The link function is the log function.  
 $\log(\lambda) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_k X_k$

3. **Gamma Regression:** Used for continuous, positive data. The link function is the inverse function.  $g(\mu) = \frac{1}{\mu}$

### Key components of GLMs:

1. **Random Component:** Specifies the probability distribution of the response variable (e.g., normal, binomial, Poisson).
2. **Systematic Component:** A linear predictor, which is a linear combination of the independent variables (e.g.,  $\eta = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_k X_k$ ).
3. **Link Function:** Connects the linear predictor to the mean of the distribution function. It transforms the expected value of the response variable to the scale on which the linear predictor is measured.

### Advantages of GLMs over Traditional Linear models:

1. **Flexibility in Error Distribution:** GLMs accommodate different types of response variables (binary, count, continuous but positive) by allowing for non-normal error distributions (e.g., binomial, Poisson, gamma).
2. **Link Functions:** By using appropriate link functions, GLMs can model non-linear relationships between the predictors and the response variable on the original scale of the data.
3. **Handling Non-constant Variance:** GLMs can handle cases where the variance of the response variable is not constant (heteroscedasticity).
4. **Broad Applicability:** GLMs are suitable for a wide range of data types and applications, including binary outcomes, count data, and proportion data.

## 6.b) Explain the logistic regression model and provide an example of its use.

### Logistic Regression:

**Logistic regression** is a type of Generalized Linear Model (GLM) used when the dependent variable is binary (i.e., it has two possible outcomes such as yes/no, true/false, 0/1). It estimates the probability that a given input point belongs to a certain class.

**Logistic Function:** Unlike linear regression, logistic regression employs the logistic function (also called the sigmoid function). This S-shaped function transforms the linear combination of independent variables (like in linear models) into a probability value between 0 and 1.

**Odds vs. Probability:** Logistic regression works with the odds of an event occurring, which is the ratio of the probability of the event happening to the probability of it not happening. The logistic function then relates these odds to the linear combination of independent variables.

### Logistic Regression Applications:

**Fraud Detection:** Classifying financial transactions as fraudulent or legitimate based on spending patterns and account information.

**Risk Assessment:** Predicting the probability of loan default for a borrower based on financial history and credit score.

**Medical Diagnosis:** Estimating the likelihood of a patient having a specific disease based on symptoms and test results.

**Online Marketing:** Classifying website visitors as potential customers based on browsing behavior and demographics.

## 7. Apply the regression models:

Height	176	154	138	196	132	176	181	169	150	175
bodymass	82	49	53	112	47	69	77	71	62	78

For the above data:

- Perform linear regression and display the result.
- Create a Regression plot with the following specifications.
- Display the title of the graph as “Height Vs. Bodymass”
- Set the color of the plot as blue

# Data (convert to numeric if necessary)

```
height <- c(176, 154, 138, 196, 132, 176, 181, 169, 150, 175)
```

```
bodymass <- c(82, 49, 53, 112, 47, 69, 77, 71, 62, 78)
```

```
Linear Regression Model
```

```
model <- lm(bodymass ~ height, data = data.frame(height, bodymass))
```

```
Display Model Summary
```

```
print(summary(model))
```

```
Regression Plot
```

```
plot(height, bodymass, col = "blue", main = "Height Vs. Bodymass", xlab = "Height", ylab =
"Bodymass")
```

```
Add regression line
```

```
abline(lm(bodymass ~ height, data = data.frame(height, bodymass)), col = "red")
```

### **Output:**

Call:

```
lm(formula = bodymass ~ height, data = data.frame(height, bodymass))
```

Residuals:

Min	1Q	Median	3Q	Max
-11.8746	-5.8428	0.7893	4.8001	15.3061

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-70.4627	24.0148	-2.934	0.018878 *
height	0.8528	0.1448	5.889	0.000366 ***

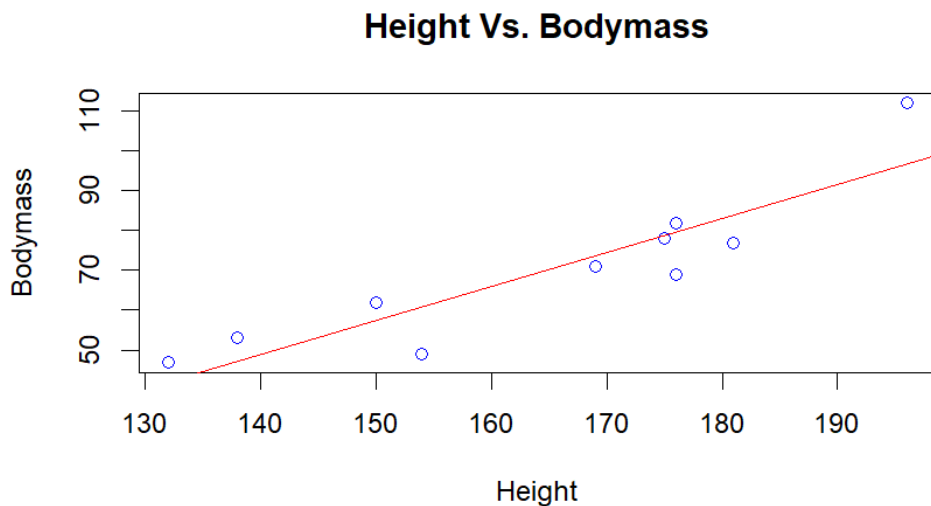
---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.854 on 8 degrees of freedom

Multiple R-squared: 0.8126, Adjusted R-squared: 0.7891

F-statistic: 34.68 on 1 and 8 DF, p-value: 0.0003662



#### 8.a) Assess the Poisson regression model and its application in analysing count data.

##### Poisson regression:

**Poisson regression** is a type of generalized linear model (GLM) used to model count data and contingency tables. It assumes the response variable  $Y$  follows a Poisson distribution and models the logarithm of its expected value as a linear combination of the predictor variables.

##### Applications of Poisson Regression:

**Public Health:** Modeling the number of disease cases in different regions.

**Transportation:** Analyzing traffic accidents based on traffic volume or other factors.

**Epidemiology:** Modeling the occurrence of rare events like infections or hospital admissions.

**Ecology:** Studying the count of species in different habitats.

##### Advantages of Poisson Regression:

**Handling Count Data:** Specifically designed to handle count data, which are non-negative integers.

**Modeling Rate Data:** Useful for modeling rates (e.g., accidents per mile, infections per person-year).

**Flexibility:** Can include offset terms to account for different exposure times or areas.

**Interpretable Results:** The model estimates the **expected rate** of events, which is easier to interpret in the context of count data compared to raw count predictions.



## **8.b) Summarize advantages of using Random Forest.**

### **Advantages of using Random Forest:**

#### **High Accuracy and Performance:**

Random Forest ensembles multiple decision trees, leading to generally higher accuracy and better performance on unseen data compared to a single decision tree. This is because it reduces the variance of the model and avoids overfitting.

#### **Robust to Noise and Outliers:**

By averaging the predictions from multiple trees, Random Forest is less sensitive to noise and outliers in the data compared to some other models.

#### **Handles Diverse Data Types:**

Random Forest can handle both categorical and continuous features without extensive data preprocessing or feature scaling.

#### **Automatic Feature Selection:**

During tree building, Random Forest considers a random subset of features at each split point. This helps identify important features for prediction without the need for manual selection.

#### **Provides Feature Importance:**

Random Forest calculates the importance of each feature based on how much it contributes to the final prediction. This helps you understand which features are most influential in the model.

#### **Less Prone to Overfitting:**

The random selection of features at each split point in tree building helps prevent the model from overfitting to the training data.

#### **Handles Missing Data:**

Random Forest can inherently handle missing data by considering alternative splits during tree construction, reducing the need for imputation techniques.

#### **Easy to Use and Interpret:**

Random Forest is relatively easy to use in R with packages like randomForest. While the inner workings of individual trees might be complex, the overall model predictions are interpretable.

#### **Scales Well with Large Datasets:**

Random Forest can efficiently handle large datasets due to its parallelization capabilities.

## 9.a) Explain the structure of decision tree.

### Decision Trees:

A relatively modern technique for fitting nonlinear models is the decision tree. Decision trees work for both regression and classification by performing binary splits on the recursive predictors.

In R, decision trees are typically created and manipulated using packages like rpart.

#### 1. Root Node

**Definition:** The topmost node in a decision tree, representing the entire dataset.

**Function:** It splits the data into two or more homogeneous sets.

#### 2. Decision Nodes (Internal Nodes)

**Definition:** Nodes that represent decisions or tests on attributes.

**Function:** These nodes split the data based on certain conditions. Each decision node has two or more branches.

#### 3. Branches

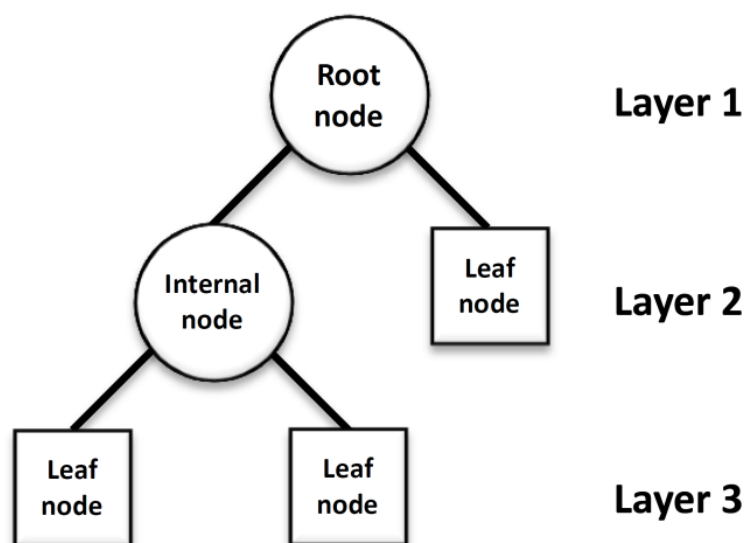
**Definition:** The outcome of a decision node leading to another node (either another decision node or a leaf node).

**Function:** They represent the flow from one decision to the next based on the outcomes of the tests.

#### 4. Leaf Nodes (Terminal Nodes)

**Definition:** The nodes that do not split further; they represent the final outcome or decision.

**Function:** Each leaf node represents a class label (in classification) or a value (in regression).



## 9.b) Categorize the principles behind decision trees and their application in random forests.

### Principles behind decision trees:

#### Recursive Partitioning:

- **Splitting:** The process of dividing a node into two or more sub-nodes based on a feature that maximizes the separation of classes (for classification) or reduces variance (for regression).
- **Best Split Selection:** Criteria like Gini impurity, information gain (entropy), or variance reduction are used to determine the best split.

#### Tree Structure:

- **Root Node:** The topmost node representing the entire dataset.
- **Decision Nodes:** Internal nodes where the data is split based on a feature.
- **Leaf Nodes:** Terminal nodes that represent the final outcome (class label for classification or value for regression).

#### Pruning:

- **Purpose:** To reduce the complexity of the model and prevent overfitting.
- **Methods:** Pre-pruning (stop growing the tree early) and post-pruning (remove branches from a fully grown tree).

#### Handling Different Data Types:

- **Categorical Data:** Splits are made based on distinct categories.
- **Numerical Data:** Splits are made based on threshold values.

#### Applications of Decision Trees in Random Forests:

**Ensemble Learning:** Random forests leverage the power of multiple decision trees. They create a collection of weak learners (individual decision trees) and combine their predictions to form a stronger, more robust model (ensemble).

#### Building a Random Forest:

1. **Bootstrap Aggregation (Bagging):** Randomly sample subsets (with replacement) of the data (typically around two-thirds) to create training sets for each tree. The remaining data points (out-of-bag data) are used for error estimation.
2. **Feature Randomness:** At each split point in a tree, a random subset of features (typically the square root of the total number of features) is considered as candidates for splitting. This helps prevent overfitting by reducing the focus on any single feature.
3. **Growing Each Tree:** Each tree is grown independently using its assigned training set and the random feature selection process.

## **Prediction in Random Forests:**

For classification problems, the most frequent class predicted by the individual trees becomes the ensemble's prediction.

For regression problems, the average predicted value from all trees is considered the ensemble's prediction.

## **10.a) Discuss the advantages of random forests over traditional statistical models.**

### **1. Enhanced Accuracy and Performance:**

- **Ensemble Learning:** Random forests create a "forest" of decision trees, each making predictions. By averaging these predictions, they often outperform single models in terms of accuracy, especially on unseen data.
- **Reduced Variance:** Combining multiple trees reduces the overall variance of the model, leading to more stable and reliable predictions.

### **2. Robustness and Overfitting Resistance:**

- **Bagging:** Random forests use bagging (bootstrap aggregating) where each tree trains on a random subset of the data. This reduces the model's dependence on any specific data points and helps prevent overfitting to the training data.
- **Feature Randomness:** At each split in a tree, only a random subset of features is considered. This prevents any single feature from dominating the model and makes it less sensitive to noise and outliers.

### **3. Handling Diverse Data Types:**

- Unlike some traditional models that require specific data formats, random forests can handle both categorical and continuous features without extensive preprocessing or feature scaling.

### **4. Automatic Feature Selection and Importance:**

- During tree building, random forests consider various features at each split. This inherent process helps identify which features are most influential for prediction, providing valuable insights into feature importance.

### **5. Scalability with Large Datasets:**

- Random forests can efficiently handle large datasets due to their parallelization capabilities. Each tree can be built independently, making them suitable for big data applications.

## 10.b) Give examples of real-world scenarios where random forests are used for predictive modeling.

### 1. Finance and Banking

- **Credit Scoring:** Predicting credit risk of applicants based on various financial and personal attributes.
- **Fraud Detection:** Identifying fraudulent transactions based on historical patterns and customer behavior.
- **Customer Segmentation:** Segmenting customers based on their financial behaviors and demographics for targeted marketing.

### 2. Healthcare

- **Disease Diagnosis:** Predicting diseases based on patient symptoms, medical history, and demographic data.
- **Drug Discovery:** Identifying potential drug candidates based on molecular structures and biological properties.
- **Patient Outcome Prediction:** Predicting patient outcomes after treatments or surgeries based on medical records and test results.

### 3. Marketing and Customer Analytics

- **Customer Churn Prediction:** Predicting which customers are likely to leave a service or product based on past behavior and interactions.
- **Cross-Selling and Upselling:** Identifying opportunities for cross-selling or upselling based on customer purchase history and preferences.
- **Market Basket Analysis:** Analyzing customer purchasing patterns to recommend related products or services.

### 4. Environmental Science

- **Ecological Modeling:** Predicting species distribution based on environmental factors such as temperature, precipitation, and habitat characteristics.
- **Climate Change Impact:** Predicting the impact of climate change on ecosystems or predicting weather patterns.

### 5. Retail and E-commerce

- **Demand Forecasting:** Predicting future demand for products based on historical sales data, seasonality, and external factors.
- **Inventory Management:** Optimizing inventory levels based on predicted demand and supply chain dynamics.
- **Price Optimization:** Predicting optimal pricing strategies based on competitor pricing, customer behavior, and market conditions.

### 6. Manufacturing and Engineering

- **Quality Control:** Predicting product defects or quality issues based on manufacturing process data and sensor readings.
- **Predictive Maintenance:** Predicting equipment failure or maintenance needs based on sensor data and historical maintenance records.

### 7. Social Media and Internet

- **Sentiment Analysis:** Analyzing and predicting public sentiment or opinion on social media platforms.
- **Click-Through Rate Prediction:** Predicting the likelihood of users clicking on online advertisements based on user behavior and ad content.

## 8. Education

- **Student Performance Prediction:** Predicting student performance based on academic records, attendance, and socio-economic factors.
- **Course Recommendation:** Recommending courses or educational programs based on student interests and past performance.