EXPERIMENT NO 10

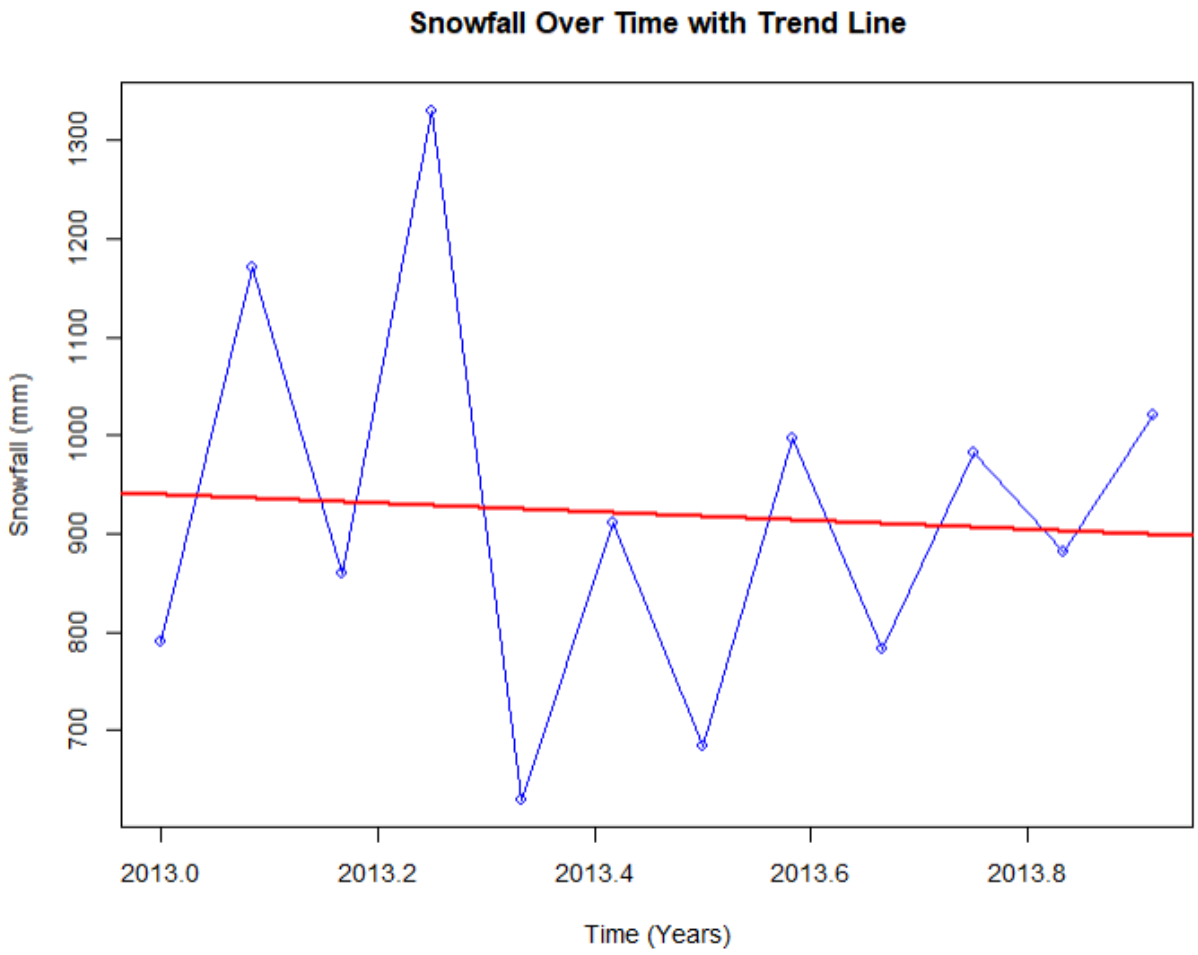
Write R program to find Time Series Analysis with the sample data and visualize the regression graphically.

Program:

```
# Getting the data points in form of a R vector.
snowfall <- c(790,1170.8,860.1,1330.6,630.4,911.5,
              683.5,996.6,783.2,982,881.8,1021)

# Converting it into a time series object.
snowfall_timeseries <- ts(snowfall, start = c(2013, 1), frequency = 12)

# Printing the time series data.
print(snowfall_timeseries)

# Plotting a graph of the time series.
plot(snowfall_timeseries,
    main = "Monthly Snowfall Time Series",
    xlab = "Time (Months)",
    ylab = "Snowfall (mm)",
    col = "blue",
    type = "o")

# Fitting linear regression model
model <- lm(snowfall_timeseries ~ time(snowfall_timeseries))

# Plotting with trend line
plot(snowfall_timeseries,
    main = "Snowfall Over Time with Trend Line",
    xlab = "Time (Years)",
    ylab = "Snowfall (mm)",
    col = "blue",
    type = "o")
abline(model, col = "red", lwd = 2)
```

Output:

```
> print(snowfall_timeseries)
      Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov
2013 790.0 1170.8 860.1 1330.6 630.4 911.5 683.5 996.6 783.2 982.0 881.8
      Dec
2013 1021.0
```



Explanation:

```
# Getting the data points in form of a R vector.
snowfall <- c(790,1170.8,860.1,1330.6,630.4,911.5,
              683.5,996.6,783.2,982,881.8,1021)
```

- `snowfall <- c(...)`
  Creates a **numeric vector** named `snowfall` that contains 12 values (one for each month).
  `c()` combines the numbers into a single object. Think of this as your raw monthly data.

```
# Converting it into a time series object.
snowfall_timeseries <- ts(snowfall, start = c(2013, 1), frequency = 12)
```

- `ts(...)` converts the numeric vector into a **time series object**.
  - `sales` becomes a time-aware object (`snowfall_timeseries`).
  - `start = c(2013, 1)` means the first observation is **January 2013**.
  - `frequency = 12` means **12 observations per year** (monthly data).
    Using `ts()` lets plotting and time functions treat the data as ordered time data (with years and months).

```
# Printing the time series data.
print(snowfall_timeseries)
```

- `print(...)` simply **displays the time series** in the console, showing each month's value with the corresponding year/month labels (e.g., `2013 Jan`, `2013 Feb`, ...).

```
# Plotting a graph of the time series.
plot(snowfall_timeseries,
    main = "Monthly Snowfall Time Series",
    xlab = "Time (Months)",
    ylab = "Snowfall (mm)",
    col = "blue",
    type = "o")
```

- `plot(snowfall_timeseries, ...)` draws a graph of the time series. The arguments:
  - `main = "..."` sets the **plot title**.

- `xlab = "..."` labels the **x-axis**.
- `ylab = "..."` labels the **y-axis**.
- `col = "blue"` sets the **color** of the plotted line/points to blue.
- `type = "o"` means **both lines and points** will appear (`o` stands for overplotted points and lines).
  This produces a simple visual of how snowfall changes month-to-month.

```
# Fitting linear regression model
model <- lm(snowfall_timeseries ~ time(snowfall_timeseries))
```

- `lm(y ~ x)` fits a **linear regression** (straight-line) model predicting `y` from `x`.
  - Left side `snowfall_timeseries` is the dependent variable (what we model).
  - Right side `time(snowfall_timeseries)` gives a numeric **time index** for each observation (e.g., 2013.000, 2013.0833, ...).
- The result is stored in `model`. This `model` summarizes the intercept and slope of the best-fit straight line through the data — i.e., the **trend** over time.

```
# Plotting with trend line
plot(snowfall_timeseries,
    main = "Snowfall Over Time with Trend Line",
    xlab = "Time (Years)",
    ylab = "Snowfall (mm)",
    col = "blue",
    type = "o")
```

- This is another plot call (same as before, but with a different title/axis wording). It redraws the time series so we can add the trend line on top of it.

```
abline(model, col = "red", lwd = 2)
```

- `abline(model, ...)` draws the **regression (trend) line** from the `lm` model onto the current plot.
    - `col = "red"` makes the line red so it stands out.
    - `lwd = 2` increases the **line width** (thicker line).
- Visually, the red line shows the long-term linear trend in snowfall across the months.

Short notes / tips

- `time(snowfall_timeseries)` returns numeric time values (year + fraction of year) used by `lm` as the predictor.
- If you want to **save** the plots to disk, wrap the plotting calls between `png("file.png")` and `dev.off()`. If you want to **see** plots interactively in RStudio, remove the `png()`/`dev.off()` calls so plots show in the Plots pane.
- To check how well the trend fits, run `summary(model)` — it shows slope, intercept, R-squared, and p-values.

Write R program to find Non Linear Least Square with the sample data and visualize the regression graphically.

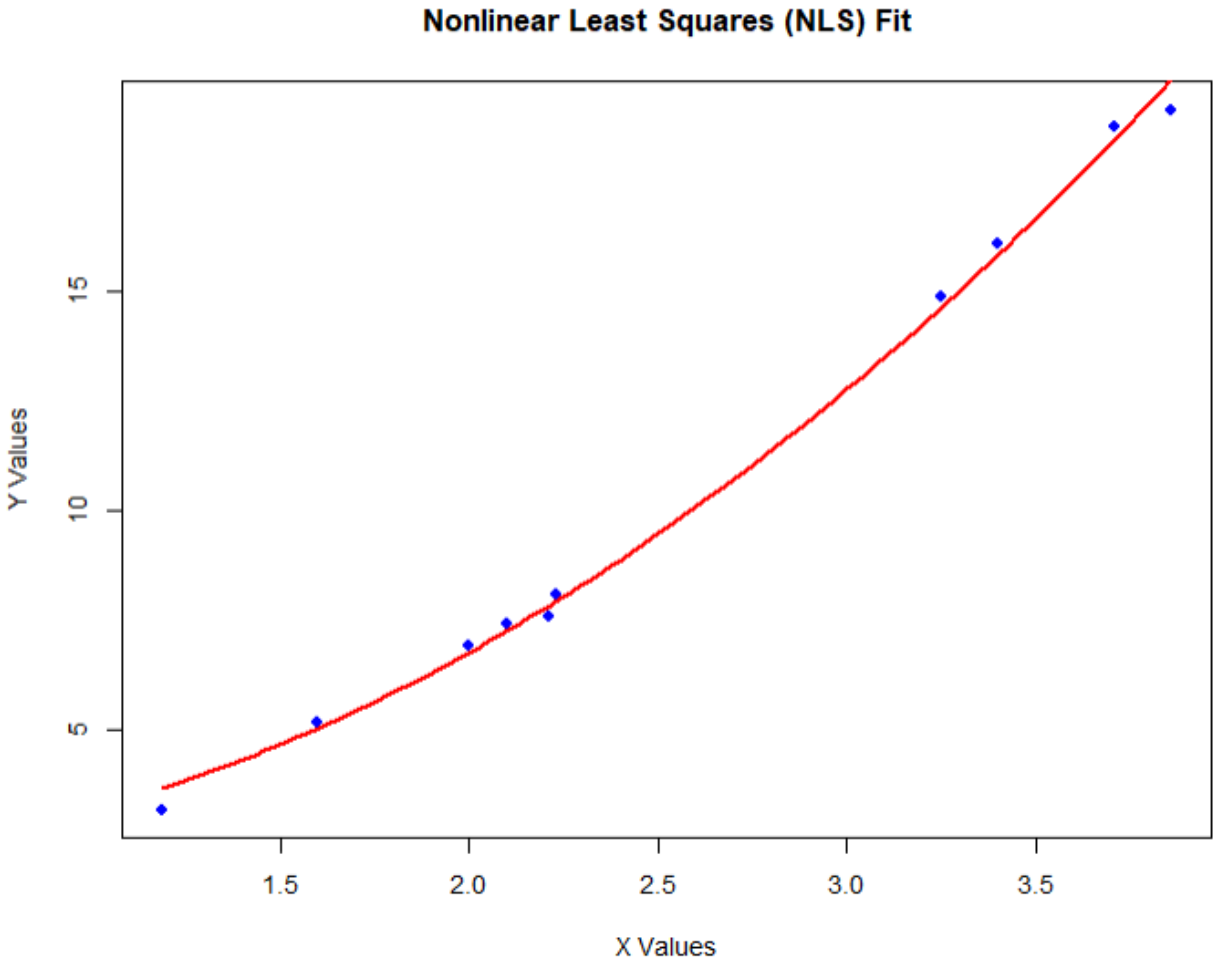
Program:

```
# Input data
xvalues <- c(1.6,2.1,2,2.23,3.71,3.25,3.4,3.86,1.19,2.21)
yvalues <- c(5.19,7.43,6.94,8.11,18.75,14.88,16.06,19.12,3.21,7.58)
```

```
# Plot the given data points
plot(xvalues, yvalues,
     main = "Nonlinear Least Squares (NLS) Fit",
     xlab = "X Values",
     ylab = "Y Values",
     col = "blue",
     pch = 16)

# Fit nonlinear model: y = b1*x^2 + b2
model <- nls(yvalues ~ b1*xvalues^2 + b2,
             start = list(b1 = 1, b2 = 3))

# Create 100 evenly spaced x values for smooth curve
new.data <- data.frame(xvalues = seq(min(xvalues), max(xvalues), length.out = 100))

# Add the fitted curve to the plot
lines(new.data$xvalues, predict(model, newdata = new.data),
      col = "red", lwd = 2)

# Print model diagnostics
cat("Sum of squared residuals:\n")
print(sum(resid(model)^2))

cat("Confidence intervals for coefficients:\n")
print(confint(model))
```

Output:

```
 >cat("Sum of squared residuals:\n")
Sum of squared residuals:
> print(sum(resid(model)^2))
[1] 1.081935
>
> cat("Confidence intervals for coefficients:\n")
Confidence intervals for coefficients:
> print(confint(model))
Waiting for profiling to be done...
       2.5%    97.5%
b1 1.137708 1.253135
b2 1.497364 2.496484
>
```



Explanation:

xvalues <- c(1.6,2.1,2,2.23,3.71,3.25,3.4,3.86,1.19,2.21)
yvalues <- c(5.19,7.43,6.94,8.11,18.75,14.88,16.06,19.12,3.21,7.58)

- `xvalues <- c(...)`
  Creates a numeric vector named `xvalues` containing 10 x-data points. `c()` combines the numbers into one object.

- `yvalues <- c(...)`
  Creates a numeric vector named `yvalues` containing the corresponding 10 y-data points. These pairs (x,y) are the data you will fit.

```
# Plot the given data points
plot(xvalues, yvalues,
    main = "Nonlinear Least Squares (NLS) Fit",
    xlab = "X Values",
    ylab = "Y Values",
    col = "blue",
    pch = 16)
```

- `plot(xvalues, yvalues, ...)`
  Draws a scatter plot with `xvalues` on the x-axis and `yvalues` on the y-axis.
- `main = "..."` sets the plot title.
- `xlab = "X Values"` labels the x-axis.
- `ylab = "Y Values"` labels the y-axis.
- `col = "blue"` sets the color of the points to blue.
- `pch = 16` chooses a solid circle symbol for the points.
  This line visually shows the raw data so you can see its shape before fitting a model.

```
# Fit nonlinear model: y = b1*x^2 + b2
model <- nls(yvalues ~ b1*xvalues^2 + b2,
             start = list(b1 = 1, b2 = 3))
```

- `nls(...)` performs **nonlinear least squares** fitting.
- `yvalues ~ b1*xvalues^2 + b2` is the model formula: it says you expect `y ≈ b1 * x^2 + b2`.
  - `b1` and `b2` are parameters the function will estimate.
- `start = list(b1 = 1, b2 = 3)` supplies **initial guesses** for the parameters. Nonlinear solvers need starting values to begin iteration.
- The fitted model object is saved into `model`. You can later inspect it (coefficients, residuals, etc.).

In short: `nls` tries to find values of `b1` and `b2` that minimize the sum of squared differences between observed `yvalues` and `b1*x^2 + b2`.

```
# Create 100 evenly spaced x values for smooth curve
new.data <- data.frame(xvalues = seq(min(xvalues), max(xvalues), length.out = 100))
```

- `seq(min(xvalues), max(xvalues), length.out = 100)` creates a sequence of 100 numbers evenly spaced from the smallest to the largest `xvalues`. This gives a smooth set of x points for plotting the fitted curve.
- `data.frame(xvalues = ...)` wraps that sequence into a data frame with a column named `xvalues`. `predict()` expects a data frame for `newdata`.
- `new.data` now holds 100 x values where you will evaluate the fitted model to draw a smooth red curve.

```
# Add the fitted curve to the plot
lines(new.data$xvalues, predict(model, newdata = new.data),
    col = "red", lwd = 2)
```

- `predict(model, newdata = new.data)` computes the model's predicted y values at each of the 100 `new.data$xvalues`, using the estimated `b1` and `b2`.
- `lines(x, y, ...)` draws a line connecting the predicted points on the existing plot (it does not create a new plot).
- `col = "red"` colors the fitted curve red so it stands out against the blue points.
- `lwd = 2` makes the line thicker (line width = 2).
  Result: a smooth red curve `y = b1*x^2 + b2` is overlaid on the blue scatter, showing how the model fits the data.

```
# Print model diagnostics
cat("Sum of squared residuals:\n")
print(sum(resid(model)^2))
```

```
cat("Confidence intervals for coefficients:\n")
print(confint(model))
```

- `cat("...")` prints a simple label text to the console to explain the following number.
- `resid(model)` returns the residuals (observed y − fitted y) for each observation.
- `sum(resid(model)^2)` computes the **sum of squared residuals (SSR)**, a single number showing the total squared error the model has — lower SSR generally means better fit.
- `print(...)` prints the SSR value.
- `confint(model)` computes approximate **confidence intervals** for the fitted parameters (`b1` and `b2`) — typically a 95% interval. It tells you a plausible range for each parameter given the data and model assumptions.
- `print(confint(model))` displays those intervals in the console.
  These diagnostics let you quantify fit quality (SSR) and uncertainty in parameter estimates (confidence intervals).

Quick tips / reminders

- If you want to **see** the plot in RStudio, do **not** wrap plotting commands between `png()` and `dev.off()` — those send the plot straight to a file instead of the RStudio Plots pane.
- To inspect the fitted parameter values directly, run `coef(model)` or `summary(model)`.
- If `nls()` fails to converge, try different `start` values — good starting guesses often help the algorithm find the best solution.

Write R program to find Decision Tree with the sample data and visualize the regression graphically.

Program:

```
# Load the built-in iris dataset

data(iris)


# Install and load required libraries

library(C50)

library(caTools)


# Make results reproducible

set.seed(7)


# Split data into 70% training and 30% testing

split <- sample.split(iris$Species, SplitRatio = 0.7)

training <- subset(iris, split == TRUE)

testing <- subset(iris, split == FALSE)


# Build the decision tree model

model <- C5.0(Species ~ ., data = training)


# View model summary
```

```
summary(model)

# Predict on the test data
pred <- predict(model, testing[,-5])

# Calculate accuracy
a <- table(testing$Species, pred)
accuracy <- sum(diag(a)) / sum(a)
print(paste("Accuracy:", round(accuracy, 3)))

# Visualize the decision tree
plot(model)
```

Output:

```
summary(model)

Call:
C5.0.formula(formula = Species ~ ., data = training)


C5.0 [Release 2.07 GPL Edition]       Wed Oct 22 12:00:48 2025
-------------------------------

Class specified by attribute `outcome'
```

Read 105 cases (5 attributes) from undefined.data

Decision tree:

Petal.Length <= 1.7: setosa (35)

Petal.Length > 1.7:

:...Petal.Length <= 4.8: versicolor (34)

    Petal.Length > 4.8: virginica (36/1)

Evaluation on training data (105 cases):

Decision Tree

Size Errors

3 1(1.0%) <<

(a) (b) (c) <-classified as

---- ---- ----

35 (a): class setosa

34 1 (b): class versicolor

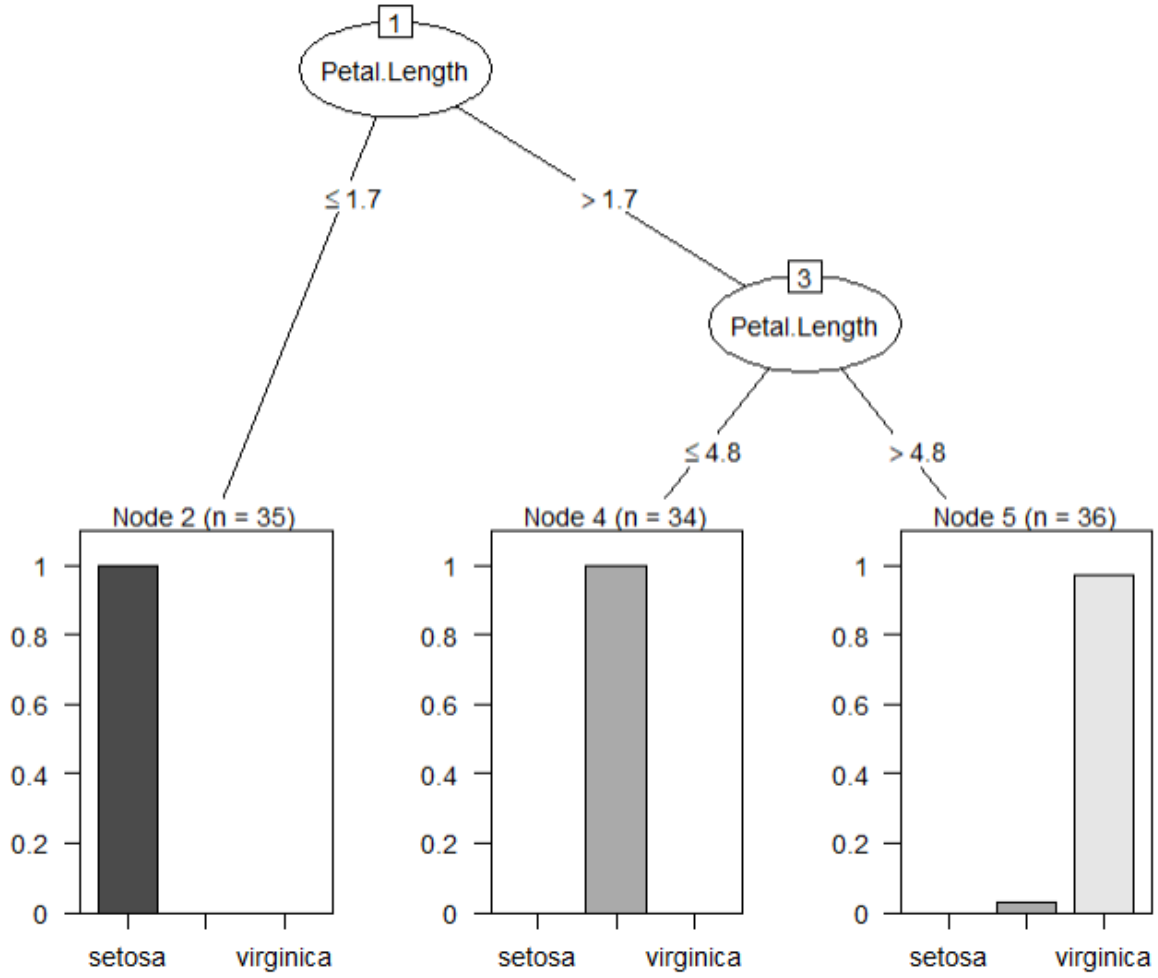
35 (c): class virginica

Attribute usage:

100.00% Petal.Length

Time: 0.0 secs

```
print(paste("Accuracy:", round(accuracy, 3)))
[1] "Accuracy: 0.822"
>
```



Explanation:

Load the built-in iris dataset

data(iris)

- `data(iris)` loads the **iris** dataset into your R session.
- `iris` is a built-in data frame that contains 150 rows and 5 columns: four numeric features (`Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`) and one factor `Species` (three classes of iris).

- After this line, you can use `iris` like any other data frame.

Install and load required libraries

library(C50)

library(caTools)

- `library(C50)` loads the **C5.0** package, which provides functions to build C5.0 decision tree models. If the package is not installed you must install it first using `install.packages("C50")`.
- `library(caTools)` loads the **caTools** package, which supplies `sample.split()` (a convenient function for randomly splitting data into training and testing sets).
- `library()` does not install packages — it only makes installed packages available.

Make results reproducible

set.seed(7)

- `set.seed(7)` sets the random number generator seed to **7**.
- This ensures that any random operations (like splitting the data) produce the **same result each time** you run the code. Use any number you like; using the same number gives identical results across runs.

Split data into 70% training and 30% testing

split <- sample.split(iris$Species, SplitRatio = 0.7)

- `sample.split(iris$Species, SplitRatio = 0.7)` creates a logical vector (`TRUE`/`FALSE`) of length 150 that indicates which rows go into the training set.
- It splits **stratified by `Species`**, meaning it tries to keep the same class proportion in training and testing sets.
- `SplitRatio = 0.7` means **~70%** of rows are marked `TRUE` (training) and the rest `FALSE` (testing).
- The result is stored in the variable `split`.

```
training <- subset(iris, split == TRUE)

testing <- subset(iris, split == FALSE)
```

- `subset(iris, split == TRUE)` selects the rows of `iris` where `split` is `TRUE` and saves them as the **training** data frame.
- `subset(iris, split == FALSE)` selects the rows where `split` is `FALSE` and saves them as the **testing** data frame.
- After these lines, `training` contains about 70% of the data (used to build the model) and `testing` contains about 30% (used to evaluate performance).

```
# Build the decision tree model

model <- C5.0(Species ~ ., data = training)
```

- `C5.0(Species ~ ., data = training)` fits a **C5.0 decision tree** to predict `Species`.
- `Species ~ .` is a formula: it means “predict `Species` using all other columns in `training` as predictors.” The `.` stands for “all other variables.”

- The fitted model object (which contains the learned tree structure, rules, and parameters) is stored in `model`.

View model summary

summary(model)

- `summary(model)` prints a human-readable summary of the fitted C5.0 model to the console. Typical contents:
    - The tree structure or rules the model learned.
    - Information on how many cases fall into each leaf/node.
    - Training accuracy and possibly pruning information.
- This helps you understand what the tree is doing and which features are being used for splits.

Predict on the test data

pred <- predict(model, testing[,-5])

- `testing[,-5]` takes the `testing` data frame and removes the 5th column (which is `Species`) so that only predictor columns are passed to `predict()`.
- `predict(model, testing[,-5])` asks the trained `model` to predict the species labels for each row in the test set using only the predictor features.
- The predicted class labels are stored in `pred` (a factor vector of predicted species).

Calculate accuracy

```
a <- table(testing$Species, pred)
```

- `table(testing$Species, pred)` builds a **confusion matrix** (`a`) where:
    - rows are the true species (from `testing$Species`),
    - columns are the predicted species (`pred`).
- Each cell counts how many test cases of a given true class were predicted as a given class. This is how you see where the model makes correct and incorrect predictions.

```
accuracy <- sum(diag(a)) / sum(a)
```

- `diag(a)` extracts the diagonal of the confusion matrix (the counts of correct predictions for each class).
- `sum(diag(a))` gives the total number of correct predictions.
- `sum(a)` gives the total number of test cases.
- Dividing gives the **overall accuracy** (proportion of correct predictions). This is a single number between 0 and 1.

```
print(paste("Accuracy:", round(accuracy, 3)))
```

- `round(accuracy, 3)` rounds the accuracy to 3 decimal places for tidy display.
- `paste("Accuracy:", ...)` concatenates the text `"Accuracy:"` with the rounded number into a single string.
- `print(...)` prints that string to the console, e.g. `"Accuracy: 0.967"`.

- This gives you a readable summary of how well the model did on the test set.

```
# Visualize the decision tree

plot(model)
```

- `plot(model)` draws a **visual representation** of the C5.0 decision tree in R's plotting window.
- The visual typically shows:
    - Root node and splits (which feature and threshold is used).
    - Child nodes and leaf nodes (with predicted class and counts).
    - Sometimes bar plots inside leaves showing class proportions.
- This plot helps you interpret the model — you can see the sequence of rules the tree uses to make predictions.

Final notes / tips

- If `C5.0` or `caTools` are not installed, first run `install.packages("C50")` and `install.packages("caTools")`.
- `predict(model, testing)` (without removing `Species`) also works for many models; some implementations require you to exclude the response — removing `testing[,-5]` is safe.
- If you prefer a different split method, `createDataPartition()` from `caret` can do stratified splits too, but it requires installing `caret`.

- To **save** the plotted tree to a file, wrap the `plot()` call between `png("tree.png")` and `dev.off()` (or use `pdf()`).

EXPERIMENT NO 11

Write R program to find the following Distribution with the sample data and visualize the linear regression graphically.

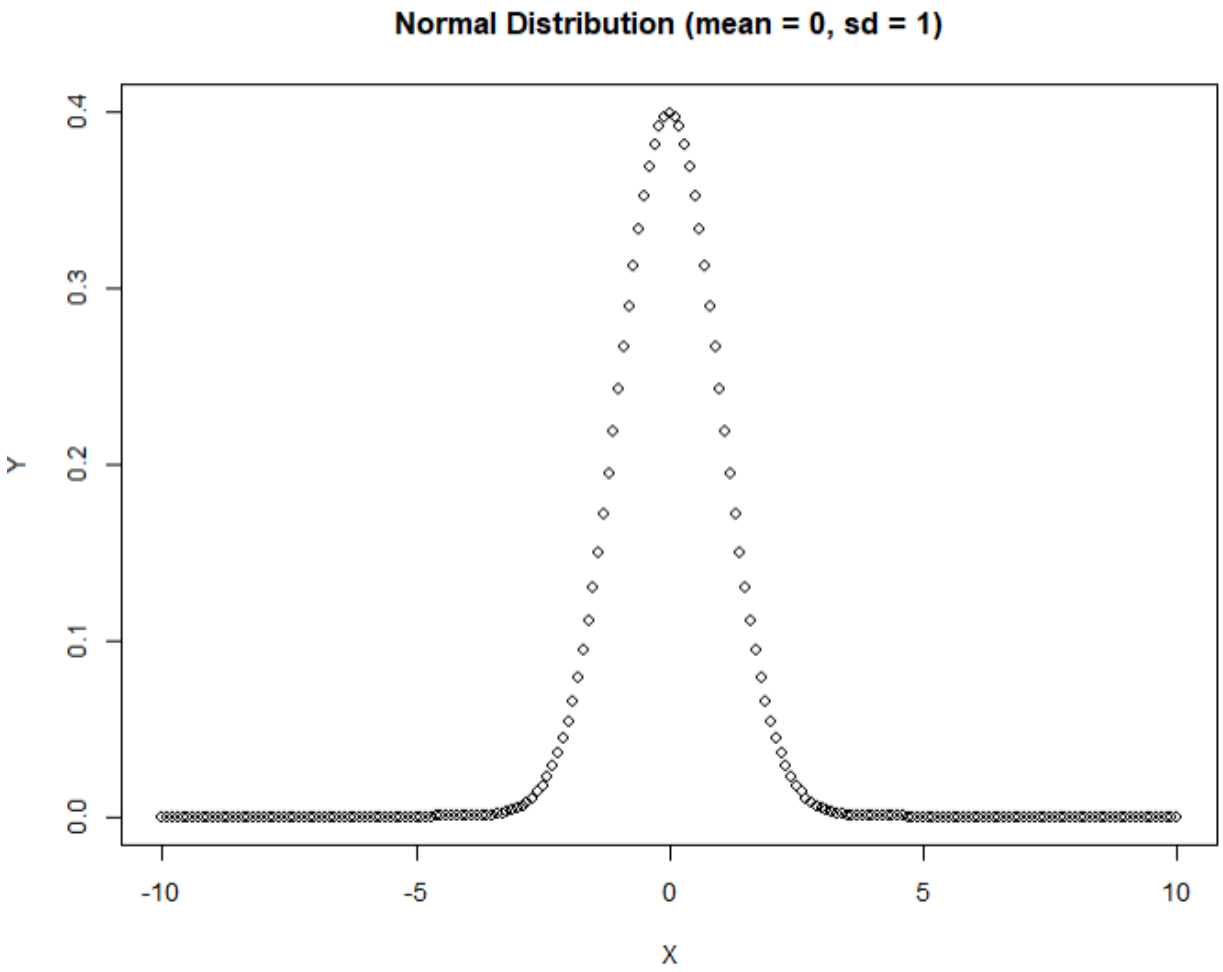
a) Normal Distribution- dnorm

Program

```
# Create a sequence of numbers between -10 and 10 incrementing by 0.1

x <- seq(-10, 10, by = 0.1)


# Choose mean = 0 and standard deviation = 1

y <- dnorm(x, mean = 0, sd = 1)


# Plot the normal distribution

plot(x, y,

    main = "Normal Distribution (mean = 0, sd = 1)",

    xlab = "X",

    ylab = "Y",
```

pch = 1) # open circles like in your image

Output:



Explanation:

Create a sequence of numbers between -10 and 10 incrementing by 0.1

x <- seq(-10, 10, by = 0.1)

- `seq()` generates a **sequence of numbers**.
- Here, `x` will contain values from `-10` to `10` in **steps of 0.1**.

- This sequence will be the **x-axis values** for plotting the normal distribution.
- Example: `x = -10, -9.9, -9.8, ..., 9.9, 10`.

```
# Choose mean = 0 and standard deviation = 1
y <- dnorm(x, mean = 0, sd = 1)
```

- `dnorm()` computes the **probability density function (PDF)** of a **normal distribution**.
- `mean = 0` → the distribution is centered at 0.
- `sd = 1` → standard deviation of 1, controlling the spread.
- `y` contains the **height of the normal curve** corresponding to each `x`.
- These values form the **y-axis** of the plot.

```
# Plot the normal distribution
plot(x, y,
     main = "Normal Distribution (mean = 0, sd = 1)",
     xlab = "X",
     ylab = "Y",
     pch = 1)   # open circles like in your image
```

- `plot(x, y, ...)` creates a **scatter plot** of `y` versus `x`.
- `main` → sets the **title** of the plot.

- `xlab` → label for the x-axis.
- `ylab` → label for the y-axis.
- `pch = 1` → specifies **plotting symbol** (here, open circles).
- This does **not connect the points with lines**; it only plots the points at `(x, y)` positions.

What This Code Produces

- A **set of points** representing the normal distribution with mean = 0 and sd = 1.
- The points form the **bell-shaped curve**, but as discrete dots instead of a continuous line.

Pnorm

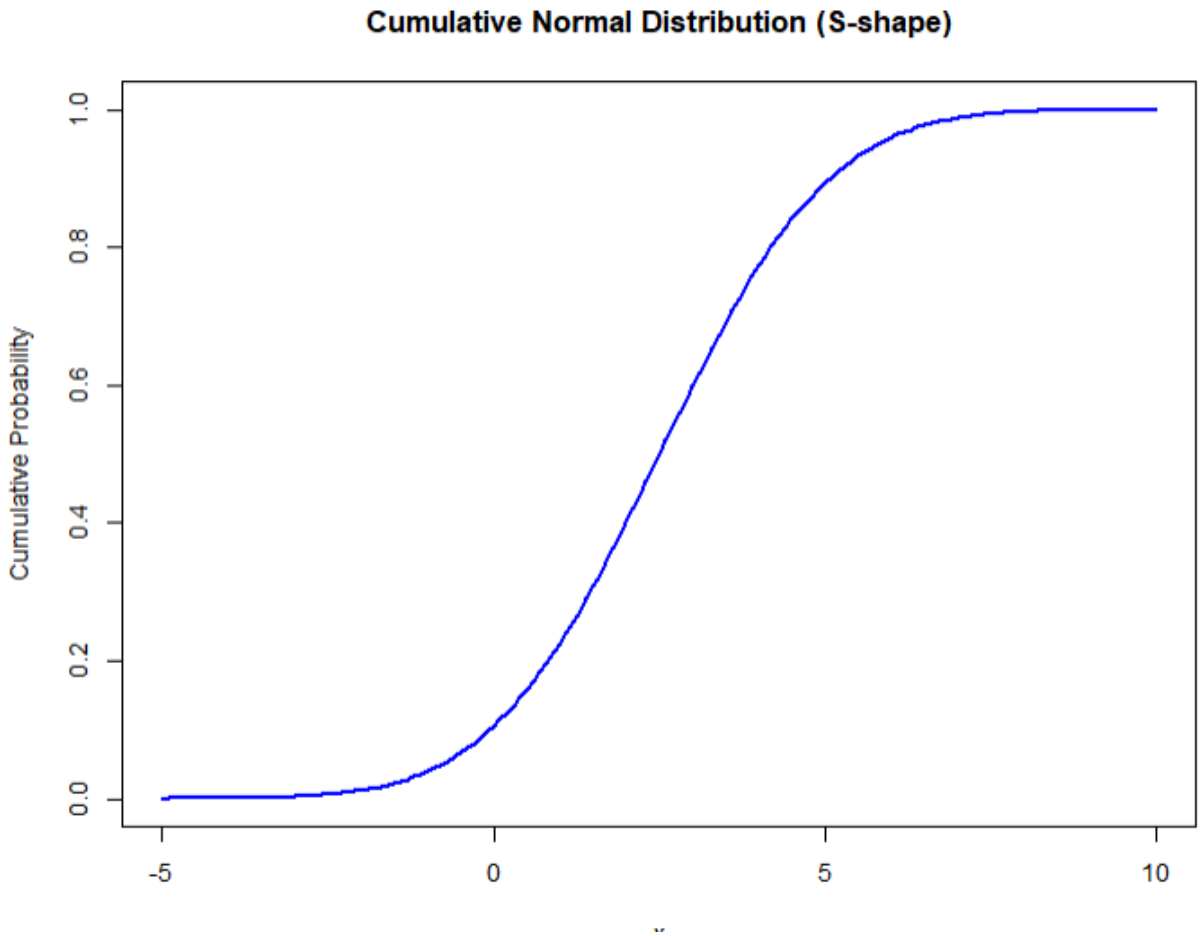
Program:

```
# Create a sequence of numbers between -5 and 10 (better range for S-shape)

x <- seq(-5, 10, by = 0.1)


# Cumulative normal distribution with mean 2.5 and sd 2

y <- pnorm(x, mean = 2.5, sd = 2)


# Plot on screen

plot(x, y, type = "l", col = "blue", lwd = 2,

    main = "Cumulative Normal Distribution (S-shape)",

    xlab = "x", ylab = "Cumulative Probability")
```

```
# Save the plot to a file
png(file = "pnorm_s_shape.png", width = 800, height = 600)
plot(x, y, type = "l", col = "blue", lwd = 2,
     main = "Cumulative Normal Distribution (S-shape)",
     xlab = "x", ylab = "Cumulative Probability")
dev.off()
```

Output:

```
dev.off()
RStudioGD
        2
>
```



Explanation:

```
# Create a sequence of numbers between -5 and 10 (better range for S-shape)
x <- seq(-5, 10, by = 0.1)
```

- `seq()` generates a **sequence of numbers**.
- Here, `x` will contain values from **-5 to 10** in increments of 0.1.
- These numbers will serve as the **x-axis values** for plotting the cumulative normal distribution.
- Choosing -5 to 10 ensures the S-shape of the cumulative distribution is **fully visible**.

```
# Cumulative normal distribution with mean 2.5 and sd 2
y <- pnorm(x, mean = 2.5, sd = 2)
```

- `pnorm()` calculates the **cumulative probability** for a normal distribution (the **CDF**).
- `mean = 2.5` → the center of the distribution.
- `sd = 2` → controls how spread out the distribution is.
- `y` now contains values between 0 and 1, representing the **probability that a random variable ≤ x**.
- Plotting `y` versus `x` will produce the **S-shaped curve** characteristic of a cumulative normal distribution.

```
# Plot on screen
plot(x, y, type = "l", col = "blue", lwd = 2,
    main = "Cumulative Normal Distribution (S-shape)",
    xlab = "x", ylab = "Cumulative Probability")
```

- `plot(x, y, ...)` creates a plot of the cumulative probabilities.
- `type = "l"` → connects the points with a **line**, producing a smooth curve.
- `col = "blue"` → sets the **line color** to blue.
- `lwd = 2` → sets the **line width** to 2 for better visibility.
- `main` → title of the plot.
- `xlab` and `ylab` → labels for the x-axis and y-axis.
- This produces a **visible S-shaped curve on the screen**.

```
# Save the plot to a file
png(file = "pnorm_s_shape.png", width = 800, height = 600)
```

- `png()` opens a **graphics device** to save the plot as a **PNG file**.
- `file = "pnorm_s_shape.png"` → specifies the **output filename**.
- `width` and `height` → dimensions of the saved image in pixels.

```
plot(x, y, type = "l", col = "blue", lwd = 2,
    main = "Cumulative Normal Distribution (S-shape)",
    xlab = "x", ylab = "Cumulative Probability")
```

- This **recreates the same plot**, but now it is sent to the **PNG file** instead of the screen.

```
dev.off()
```

- `dev.off()` **closes the PNG device**, saving the plot to the file.
- Without this, the image file would not be properly created.

Summary

- This code creates a **sequence of x values**, calculates their **cumulative normal probabilities**, plots an **S-shaped cumulative distribution**, and saves it as a **PNG file**.
- `type="l"` ensures the S-shape is smooth, and `col`/`lwd` make it visually clear.

---

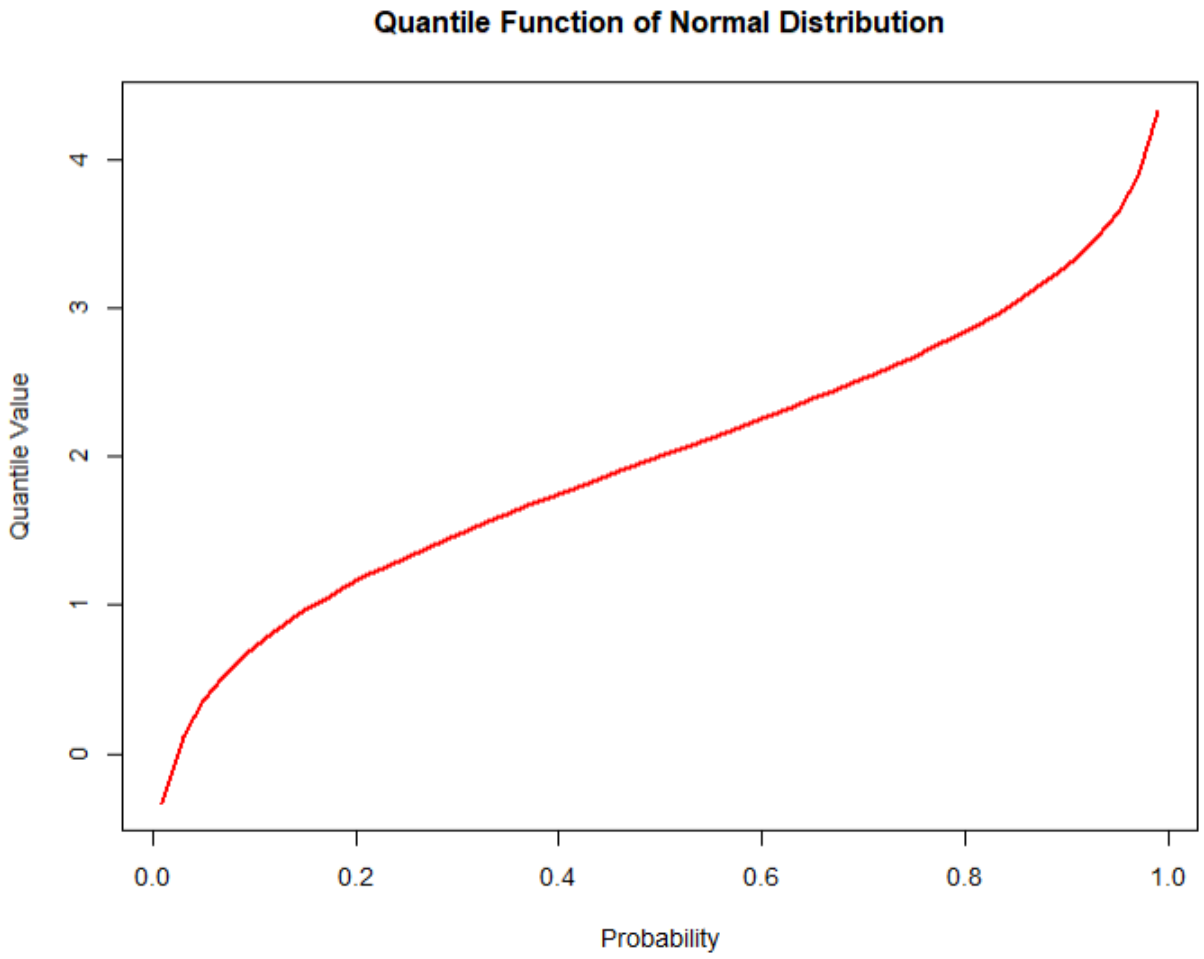
qnorm

Program:

```
# Create a sequence of probability values between 0.01 and 0.99
x <- seq(0.01, 0.99, by = 0.02)   # avoid 0 and 1, qnorm goes to -Inf/+Inf

# Quantile function of normal distribution (inverse CDF)
y <- qnorm(x, mean = 2, sd = 1)

# Plot on screen
plot(x, y, type = "l", col = "red", lwd = 2,
     main = "Quantile Function of Normal Distribution",
     xlab = "Probability", ylab = "Quantile Value")

# Save the plot as PNG
png(file = "qnorm_curve.png", width = 800, height = 600)
plot(x, y, type = "l", col = "red", lwd = 2,
     main = "Quantile Function of Normal Distribution",
     xlab = "Probability", ylab = "Quantile Value")
dev.off()
```

Output:

```
dev.off()
RStudioGD
        2
>
```



Explanation:

```
# Create a sequence of probability values between 0.01 and 0.99
x <- seq(0.01, 0.99, by = 0.02)   # avoid 0 and 1, qnorm goes to -Inf/+Inf
```

- `seq()` generates a **sequence of numbers**.
- Here, `x` contains probabilities from **0.01 to 0.99** in increments of 0.02.
- **Avoid 0 and 1** because `qnorm(0)` is `-Inf` and `qnorm(1)` is `+Inf`.

- These probabilities will be used as **input values** for the quantile function.

```
# Quantile function of normal distribution (inverse CDF)
y <- qnorm(x, mean = 2, sd = 1)
```

- `qnorm()` computes the **quantile function** (inverse CDF) of a normal distribution.
- `mean = 2` → the distribution is centered at 2.
- `sd = 1` → standard deviation controls the spread.
- `y` contains the **quantile values** corresponding to the probabilities `x`.
- Essentially, `y[i]` is the value of the normal variable such that **P(X ≤ y[i]) = x[i]**.

```
# Plot on screen
plot(x, y, type = "l", col = "red", lwd = 2,
    main = "Quantile Function of Normal Distribution",
    xlab = "Probability", ylab = "Quantile Value")
```

- `plot(x, y, ...)` creates a **line plot** of `y` versus `x`.
- `type = "l"` → draws a **line** connecting all points.
- `col = "red"` → sets the line color to red.
- `lwd = 2` → makes the line thicker for better visibility.
- `main` → title of the plot.

- `xlab` and `ylab` → labels for the x-axis (probability) and y-axis (quantile).
- This produces a **smooth, diagonal S-shaped curve** showing the quantiles across probabilities.

```
# Save the plot as PNG

png(file = "qnorm_curve.png", width = 800, height = 600)
```

- Opens a **graphics device** to save the plot as a **PNG file**.
- `file = "qnorm_curve.png"` → the output filename.
- `width` and `height` → dimensions of the image in pixels.
- Any plot commands after this will be sent to the PNG file instead of the screen.

```
plot(x, y, type = "l", col = "red", lwd = 2,
     main = "Quantile Function of Normal Distribution",
     xlab = "Probability", ylab = "Quantile Value")
```

- Recreates the **same plot**, but now it is sent to the **PNG file**.
- Ensures the saved file matches the on-screen plot.

```
dev.off()
```

- Closes the **graphics device**, finalizing the PNG file.

- Without this, the file may be incomplete or blank.

Summary

- Creates a **probability sequence** for the CDF.
- Calculates **quantiles using `qnorm`** (inverse of the normal CDF).
- Plots a **smooth S-shaped quantile curve**.
- Saves the plot to a **PNG file**.

---

rnorm

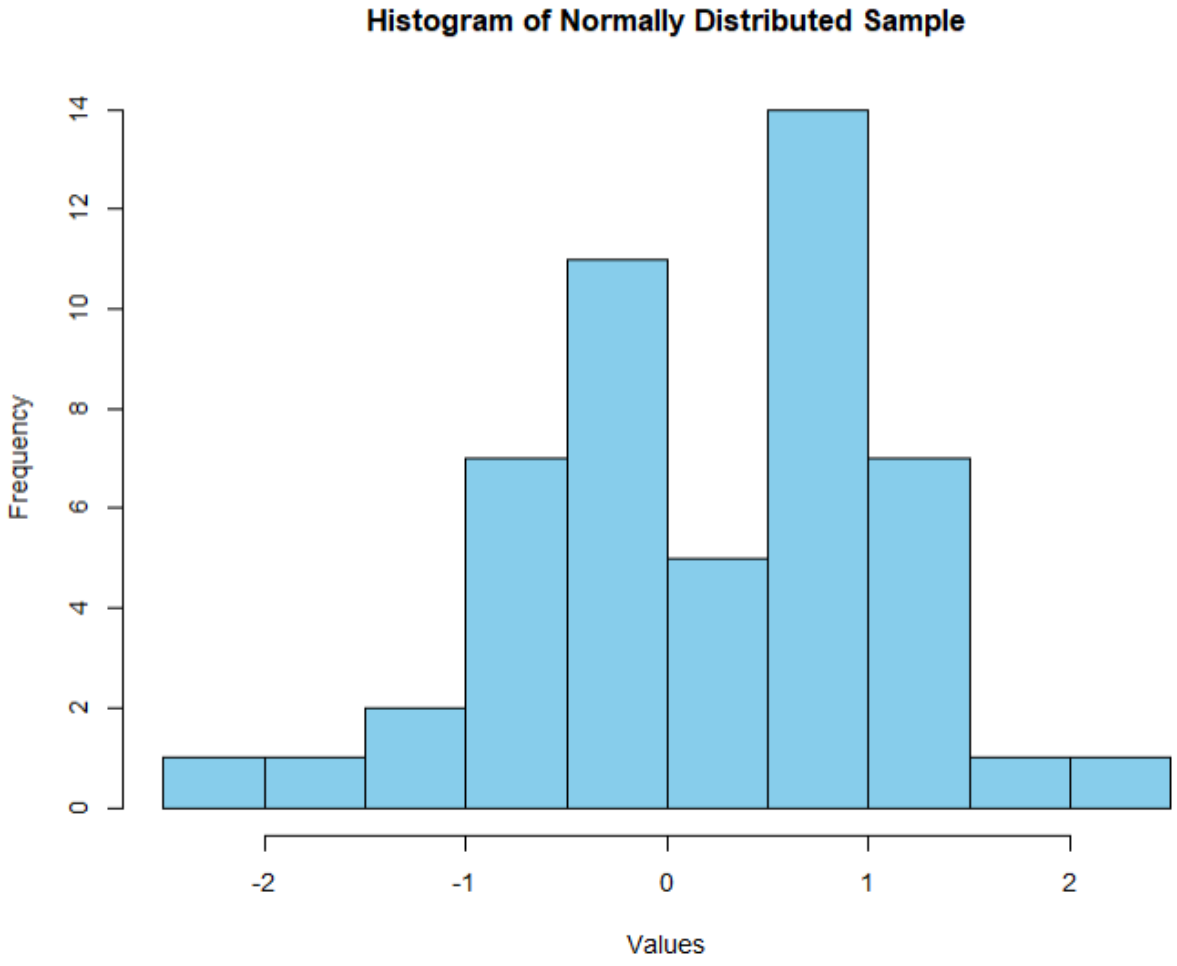
Program:

```
# Create a sample of 50 normally distributed numbers
y <- rnorm(50)


# Plot histogram on screen
hist(y,
     breaks = 10,            # number of bins
     col = "skyblue",        # bar color
     border = "black",       # bar border
     main = "Histogram of Normally Distributed Sample",
     xlab = "Values",
```

```
    ylab = "Frequency")


# Save the histogram as PNG
png(file = "rnorm_histogram.png", width = 800, height = 600)
hist(y,
    breaks = 10,
    col = "skyblue",
    border = "black",
    main = "Histogram of Normally Distributed Sample",
    xlab = "Values",
    ylab = "Frequency")
dev.off()
```

Output:

```
dev.off()
RStudioGD
        2
```



Explanation:

```
# Create a sample of 50 normally distributed numbers

y <- rnorm(50)
```

- `rnorm(50)` generates **50 random numbers** from a **standard normal distribution** (mean = 0, sd = 1 by default).
- These numbers are stored in `y`.
- This will be the data used to create a histogram.

```
# Plot histogram on screen
hist(y,
     breaks = 10,            # number of bins
     col = "skyblue",        # bar color
     border = "black",       # bar border
     main = "Histogram of Normally Distributed Sample",
     xlab = "Values",
     ylab = "Frequency")
```

- `hist(y, ...)` creates a **histogram** of the sample `y`.
- `breaks = 10` → divides the data range into **10 bins**, determining bar width.
- `col = "skyblue"` → sets the **fill color** of the bars.
- `border = "black"` → sets the **color of bar edges**.
- `main` → adds a **title** to the plot.
- `xlab` → label for the x-axis (**Values of the sample**).
- `ylab` → label for the y-axis (**Frequency/count of observations**).
- This produces a **histogram on the screen** showing the distribution of the 50 numbers.

```
# Save the histogram as PNG
png(file = "rnorm_histogram.png", width = 800, height = 600)
```

- Opens a **PNG graphics device** to save the next plot as a file.
- `file = "rnorm_histogram.png"` → name of the output file.
- `width` and `height` → dimensions of the saved image in pixels.
- Any subsequent plotting commands are sent to this file instead of the screen.

```
hist(y,
    breaks = 10,
    col = "skyblue",
    border = "black",
    main = "Histogram of Normally Distributed Sample",
    xlab = "Values",
    ylab = "Frequency")
```

- Recreates the **same histogram**, but this time it is sent to the **PNG file**.

```
dev.off()
```

- Closes the **graphics device**, finalizing the PNG file.
- Without `dev.off()`, the file might be incomplete or blank.

Summary

- Generates **50 random normal numbers**.
- Plots a **histogram on screen** to show the distribution.
- Saves the histogram as a **PNG file** with custom color, bins, and labels.

Binomial Distribution

dbinom

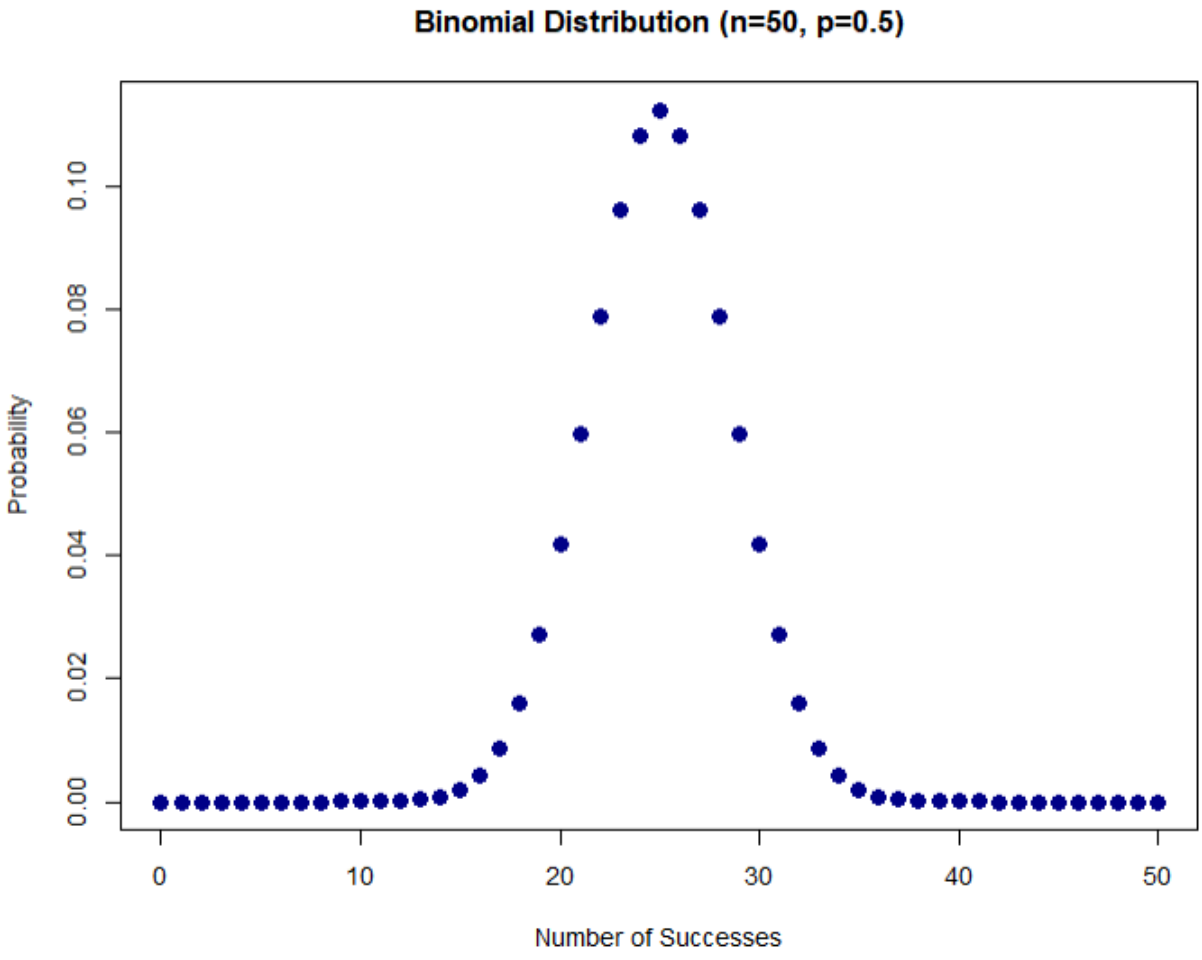
Program:

```
# Create a sequence of numbers from 0 to 50
x <- seq(0, 50, by = 1)

# Binomial distribution with n=50, p=0.5
y <- dbinom(x, size = 50, prob = 0.5)

# Plot on screen with dots only
plot(x, y,
     type = "p",           # points only
     pch = 19,             # solid dots
     col = "darkblue",     # dot color
     cex = 1.5,            # dot size
     main = "Binomial Distribution (n=50, p=0.5)",
     xlab = "Number of Successes",
     ylab = "Probability")

# Save the plot as PNG
png(file = "dbinom_dots.png", width = 800, height = 600)
plot(x, y,
     type = "p",
```

```
    pch = 19,
    col = "darkblue",
    cex = 1.5,
    main = "Binomial Distribution (n=50, p=0.5)",
    xlab = "Number of Successes",
    ylab = "Probability")
dev.off()
```

Output:

```
 dev.off()
RStudioGD
        2
>
```



Explanation:

```
# Create a sequence of numbers from 0 to 50
x <- seq(0, 50, by = 1)
```

- `seq()` generates a **sequence of numbers**.
- Here, `x` contains all integers from 0 to 50.
- These values represent the **possible number of successes** in a binomial experiment with 50 trials.

```
# Binomial distribution with n=50, p=0.5
y <- dbinom(x, size = 50, prob = 0.5)
```

- `dbinom()` computes the **probability mass function (PMF)** of the binomial distribution.
- `size = 50` → number of trials.
- `prob = 0.5` → probability of success in each trial.
- `y` contains the **probabilities of each number of successes** from 0 to 50.

```
# Plot on screen with dots only
plot(x, y,
    type = "p",           # points only
    pch = 19,             # solid dots
    col = "darkblue",     # dot color
    cex = 1.5,            # dot size
    main = "Binomial Distribution (n=50, p=0.5)",
    xlab = "Number of Successes",
    ylab = "Probability")
```

- `plot(x, y, ...)` creates a **scatter plot** of `y` versus `x`.

- `type = "p"` → plots **only points**, no lines connecting them.
- `pch = 19` → solid circles for the points.
- `col = "darkblue"` → sets the dot color to dark blue.
- `cex = 1.5` → increases the **size of the dots** for visibility.
- `main` → title of the plot.
- `xlab` and `ylab` → labels for x-axis (**Number of Successes**) and y-axis (**Probability**).
- This produces a **mountain-like shape** formed by dots representing the binomial PMF.

```
# Save the plot as PNG
png(file = "dbinom_dots.png", width = 800, height = 600)
```

- Opens a **PNG graphics device** to save the next plot as an image file.
- `file = "dbinom_dots.png"` → name of the output file.
- `width` and `height` → size of the saved image in pixels.
- Any plot commands after this are sent to the **PNG file**, not the screen.

```
plot(x, y,
    type = "p",
    pch = 19,
    col = "darkblue",
    cex = 1.5,
    main = "Binomial Distribution (n=50, p=0.5)",
    xlab = "Number of Successes",
    ylab = "Probability")
```

- Recreates the **same scatter plot**, but now it is drawn in the **PNG file** instead of the screen.

---

dev.off()

- Closes the **graphics device**, finalizing and saving the PNG file.
- Without this, the file might not be complete or may remain blank.

Summary

- Generates **all possible number of successes** for 50 trials.
- Calculates **probabilities using the binomial distribution**.
- Plots a **mountain-like distribution using dots only** on screen.
- Saves the **same plot as a PNG file**.

Pbinom

Program:

```
# Probability of getting 26 or less heads from a 51 tosses of a coin.
x <- pbinom(26,51,0.5)

print(x)
```

Output:

```
> print(x)
[1] 0.610116
>
```

Explanation:

# Probability of getting 26 or less heads from 51 tosses of a coin.
x <- pbinom(26, 51, 0.5)

- `pbinom()` computes the **cumulative probability** of a binomial distribution.
- `26` → the number of successes (here, heads) you are interested in.
- `51` → the total number of trials (coin tosses).
- `0.5` → probability of success (head) in each trial.
- This function calculates **P(X ≤ 26)**, i.e., the probability of getting **26 or fewer heads** in 51 tosses.
- The result is stored in `x`.

---

print(x)

- Prints the **cumulative probability** calculated above.
- This will be a number between 0 and 1, representing the likelihood of getting **26 or fewer heads**.

Example Interpretation

- If `x = 0.55`, it means there is a **55% chance** of getting **26 or fewer heads** in 51 coin tosses.

---

qbinom

Program:

```
x <- qbinom(0.25,51,1/2)

print(x)
```

Output:

```
> print(x)
[1] 23
>
```

Explanation:

```
x <- qbinom(0.25, 51, 1/2)
```

- `qbinom()` computes the **quantile function** (inverse CDF) of the **binomial distribution**.
- `0.25` → the cumulative probability you are interested in (25%).
- `51` → number of trials (e.g., 51 coin tosses).
- `1/2` → probability of success in each trial (here, probability of heads = 0.5).
- The function calculates the **smallest number of successes k** such that **P(X ≤ k) ≥ 0.25**.
- In other words, it answers the question: "**How many heads correspond to the 25th percentile of 51 tosses?**"
- The result is stored in `x`.

```
print(x)
```

- Prints the value of `x`, i.e., the **quantile or cutoff number of successes**.
- For example, if `x = 24`, it means that **25% of the time, you get 24 or fewer heads** in 51 tosses.

Summary

- `pbinom()` → gives probability for a **given number of successes** (CDF).
- `qbinom()` → gives the **number of successes corresponding to a given probability** (inverse CDF).
- Here, you are finding the **25th percentile** of heads in 51 coin tosses.

---

rbinom

Program:

```
x <- rbinom(8,150,.4)

print(x)
```

Output:

```
print(x)
[1] 54 67 66 56 58 60 51 60
>
```

Explanation:

x <- rbinom(8, 150, 0.4)

- `rbinom()` generates **random numbers from a binomial distribution**.
- `8` → the **number of random observations** you want to generate.
- `150` → the **number of trials** in each observation (e.g., 150 coin tosses per trial).
- `0.4` → the **probability of success** in each trial (e.g., probability of getting "heads" = 0.4).
- The function returns **8 random numbers**, each representing the **number of successes in 150 trials**.
- The result is stored in `x`.

---

print(x)

- Prints the **8 randomly generated numbers**.

Example output:

[1] 58 61 60 55 62 59 57 63

- Each number is the **count of successes** (e.g., heads) out of 150 trials for that observation.

Summary

- `rbinom(n, size, prob)` → **simulate binomial experiments**.
- Here, you simulated **8 experiments**, each with **150 trials and 0.4 probability of success**.
- The output shows **how many successes occurred in each experiment**.

EXPERIMENT NO 12

Write R program to do the following tests with the sample data and visualize the results graphically.

a) χ2-test

Program:

```
# Load the MASS library to access the Cars93 dataset
library(MASS)

# Display the structure of Cars93
print(str(Cars93))

# Create a contingency table of AirBags vs Type
car_data <- table(Cars93$AirBags, Cars93$Type)
print(car_data)

# Perform the Chi-Square test on the table
chi_result <- chisq.test(car_data)
print(chi_result)
```

Output:

```
> print(str(Cars93))
'data.frame':    93 obs. of  27 variables:
 $ Manufacturer      : Factor w/ 32 levels "Acura","Audi",..: 1 1 2 2 3 4 4 4 4 5 ...
 $ Model             : Factor w/ 93 levels "100","190E","240",..: 49 56 9 1 6 24 54 74 73 35 ...
 $ Type              : Factor w/ 6 levels "Compact","Large",..: 4 3 1 3 3 3 2 2 3 2 ...
 $ Min.Price         : num  12.9 29.2 25.9 30.8 23.7 14.2 19.9 22.6 26.3 33 ...
 $ Price             : num  15.9 33.9 29.1 37.7 30 15.7 20.8 23.7 26.3 34.7 ...
 $ Max.Price         : num  18.8 38.7 32.3 44.6 36.2 17.3 21.7 24.9 26.3 36.3 ...
```

```
 $ MPG.city          : int  25 18 20 19 22 22 19 16 19 16 ...
 $ MPG.highway       : int  31 25 26 26 30 31 28 25 27 25 ...
 $ AirBags           : Factor w/ 3 levels "Driver & Passenger",..: 3 1 2 1 2 2 2 2 2 2 ...
 $ DriveTrain        : Factor w/ 3 levels "4WD","Front",..: 2 2 2 2 3 2 2 3 2 2 ...
 $ Cylinders         : Factor w/ 6 levels "3","4","5","6",..: 2 4 4 4 2 2 4 4 4 5 ...
 $ EngineSize        : num  1.8 3.2 2.8 2.8 3.5 2.2 3.8 5.7 3.8 4.9 ...
 $ Horsepower        : int  140 200 172 172 208 110 170 180 170 200 ...
 $ RPM               : int  6300 5500 5500 5500 5700 5200 4800 4000 4800 4100 ...
 $ Rev.per.mile      : int  2890 2335 2280 2535 2545 2565 1570 1320 1690 1510 ...
 $ Man.trans.avail   : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 1 1 1 1 ...
 $ Fuel.tank.capacity: num  13.2 18 16.9 21.1 21.1 16.4 18 23 18.8 18 ...
 $ Passengers        : int  5 5 5 6 4 6 6 6 5 6 ...
 $ Length            : int  177 195 180 193 186 189 200 216 198 206 ...
 $ Wheelbase         : int  102 115 102 106 109 105 111 116 108 114 ...
 $ Width             : int  68 71 67 70 69 69 74 78 73 73 ...
 $ Turn.circle       : int  37 38 37 37 39 41 42 45 41 43 ...
 $ Rear.seat.room    : num  26.5 30 28 31 27 28 30.5 30.5 26.5 35 ...
 $ Luggage.room      : int  11 15 14 17 13 16 17 21 14 18 ...
 $ Weight            : int  2705 3560 3375 3405 3640 2880 3470 4105 3495 3620 ...
 $ Origin            : Factor w/ 2 levels "USA","non-USA": 2 2 2 2 2 1 1 1 1 1 ...
 $ Make              : Factor w/ 93 levels "Acura Integra",..: 1 2 4 3 5 6 7 9 8 10 ...
NULL
>
> # Create a contingency table of AirBags vs Type
> car_data <- table(Cars93$AirBags, Cars93$Type)
> print(car_data)

                   Compact Large Midsize Small Sporty Van
  Driver & Passenger       2     4       7     0      3   0
  Driver only              9     7      11     5      8   3
  None                     5     0       4    16      3   6
>
> # Perform the Chi-Square test on the table
> chi_result <- chisq.test(car_data)
Warning message:
In chisq.test(car_data) : Chi-squared approximation may be incorrect
> print(chi_result)

        Pearson's Chi-squared test

data:  car_data
X-squared = 33.001, df = 10, p-value = 0.0002723
```

Explanation:

```
# Load the MASS library to access the Cars93 dataset
library(MASS)
```

- `library(MASS)` loads the **MASS package**, which contains the dataset **Cars93**.
- Without this, you cannot access `Cars93`.

```
# Display the structure of Cars93
print(str(Cars93))
```

- `str(Cars93)` shows the **structure of the dataset**.
- It tells you:
    - How many **observations** and **variables** there are.
    - The **type of each variable** (numeric, factor, etc.).
    - A **preview of the data**.
- `print()` just ensures the output is displayed on the console.

```
# Create a contingency table of AirBags vs Type
car_data <- table(Cars93$AirBags, Cars93$Type)
```

- `table()` creates a **cross-tabulation** (contingency table).
- Rows = different **AirBags categories** (e.g., "Driver & Passenger", "Driver only", "None").
- Columns = different **Car Types** (e.g., Compact, Large, Midsize, etc.).
- Each cell = **count of cars** for that AirBags type and Car Type combination.

---

print(car_data)

- Displays the **contingency table** on the screen.
- You can see **how many cars** have a particular AirBags type for each Car Type.

# Perform the Chi-Square test on the table
chi_result <- chisq.test(car_data)

- `chisq.test()` performs a **Chi-Square test of independence**.
- It tests if **AirBags type is independent of Car Type**.
- The result contains:
    - `X-squared` value → test statistic.
    - `df` → degrees of freedom.
    - `p-value` → tells if the relationship is **statistically significant**.

---

print(chi_result)

- Displays the **results of the Chi-Square test** on the screen.
- You can see whether **AirBags and Car Type are related**.

Summary in Simple Terms

1. Load dataset (`MASS` library).
2. Look at dataset structure (`str(Cars93)`).

3. Make a table showing **AirBags vs Car Type**.
4. Print the table.
5. Perform **Chi-Square test** to see if AirBags and Car Type are related.
6. Print the test result.

---

t-test

Program:

x <- c(0.593, 0.142, 0.329, 0.691, 0.231, 0.793, 0.519, 0.392, 0.418) t.test(x, alternative="greater", mu=0.3)

Output:

```
> t.test(x, alternative="greater", mu=0.3)

        One Sample t-test

data:  x
t = 2.2051, df = 8, p-value = 0.02927
alternative hypothesis: true mean is greater than 0.3
95 percent confidence interval:
 0.3245133       Inf
sample estimates:
mean of x
0.4564444
```

Explanation:

x <- c(0.593, 0.142, 0.329, 0.691, 0.231, 0.793, 0.519, 0.392, 0.418)

- `c()` creates a **vector of numbers**.

- Here, `x` is a dataset of **9 observations**.
- These could represent measurements, scores, or any sample data.

---

t.test(x, alternative="greater", mu=0.3)

- `t.test()` performs a **one-sample t-test**.
- **Purpose:** To check if the **mean of the sample `x` is significantly greater than a hypothesized value `mu`**.

Arguments:

1. `x` → the **sample data**.
2. `alternative = "greater"` → we are testing if **mean(x) > mu**.
3. `mu = 0.3` → the **hypothesized population mean**.

What the Test Does

- Null hypothesis (H_0): `mean(x) = 0.3`
- Alternative hypothesis (H_1): `mean(x) > 0.3`
- The test calculates:
    - `t-value` → measures how far the sample mean is from 0.3 in terms of standard error.
    - `df` → degrees of freedom = n-1 (here, 9-1 = 8).
    - `p-value` → probability of observing such a t-value if H_0 is true.

- If `p-value < 0.05` (common threshold), we **reject H_0** and conclude the sample mean is significantly **greater than 0.3**.

Example of interpretation
Suppose output gives:
t = 2.7, df = 8, p-value = 0.013
mean of x = 0.441

- The **sample mean 0.441 > 0.3**,
- `p-value = 0.013 < 0.05` → significant, so we **reject H_0**.
- Conclusion: The mean of the sample is **significantly greater than 0.3**.

In short: This code **tests if your 9 numbers have a mean larger than 0.3** and gives a **statistical measure of significance**.

---

f-test

Program

```
x<-c(18,19,22,25,27,28,41,45,51,55)
y<-c(14,15,15,17,18,22,25,25,27,34)
print(var.test(x,y))
```

Output:

```
F test to compare two variances

data:  x and y
F = 4.3871, num df = 9, denom df = 9, p-value = 0.03825
alternative hypothesis: true ratio of variances is not equal to 1
```

95 percent confidence interval:
 1.089699 17.662528
sample estimates:
ratio of variances
        4.387122

Explanation:

x <- c(18, 19, 22, 25, 27, 28, 41, 45, 51, 55)

- `c()` creates a **vector** of numbers.
- Here, `x` is a sample of **10 observations**.
- These could represent measurements or data points from one group.

---

y <- c(14, 15, 15, 17, 18, 22, 25, 25, 27, 34)

- Another **vector of numbers**, representing a second group of **10 observations**.
- We want to **compare the variability** (variance) of `x` and `y`.

---

print(var.test(x, y))

- `var.test()` performs an **F-test to compare the variances** of two samples.

What it does:

- Null hypothesis (H_0): **the variances are equal** (`var(x) = var(y)`)
- Alternative hypothesis (H_1): **the variances are different** (depends on default or specified `alternative`)

- Computes:
    - `F` value → ratio of sample variances (`var(x)/var(y)` if `var(x) > var(y)`)
    - `df1` and `df2` → degrees of freedom for x and y (`n1-1` and `n2-1`)
    - `p-value` → probability of observing this F-statistic if H_0 is true
- `print()` shows the result on the screen.

Example Output Interpretation

Suppose the output is:

```
        F test to compare two variances

data:  x and y
F = 2.15, num df = 9, denom df = 9, p-value = 0.1645
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.73 6.32
sample estimates:
ratio of variances
             2.15
```

- `F = 2.15` → variance of `x` is **2.15 times** the variance of `y`.
- `p-value = 0.1645 > 0.05` → **fail to reject H_0**, variances are **not significantly different**.
- Confidence interval gives the range for the true ratio of variances.

Summary

1. `x` and `y` → two samples.

2. `var.test(x, y)` → F-test to check if the **variances of the two samples are equal**.

3. Output includes **F-value, degrees of freedom, p-value, confidence interval, and ratio of variances**.
