# **FSAI Lab Experiments**

## **Experiment 1**

**1. R - Matrix Transformations and Linear Algebra**

* **Aim:** To demonstrate the use of R for creating matrices and performing common linear algebra operations like addition, multiplication, transpose, inverse, determinant calculation, solving linear equations, and finding eigenvalues/eigenvectors.
* **Algorithm/Approach:**
  1. **Matrix Creation:** Use matrix() to create matrices A and B, and a vector x.
  2. **Basic Operations:** Perform addition (+), subtraction (-), element-wise multiplication (\*), and matrix multiplication (%\*%).
  3. **Transpose:** Calculate the transpose using t().
  4. **Determinant:** Calculate the determinant using det(). Check if it's non-zero.
  5. **Inverse:** If the determinant is non-zero, calculate the inverse using solve(A).
  6. **Solve Linear System:** If the determinant is non-zero, solve the system Ay = x for y using solve(A, x).
  7. **Eigen Decomposition:** Calculate eigenvalues and eigenvectors using eigen().
  8. **Print Results:** Print all matrices and results of operations with descriptive labels.
* **Code/Program:**  
  # Create matrices  
  A <- matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE)  
  B <- matrix(c(5, 6, 7, 8), nrow = 2, byrow = TRUE)  
  x <- c(1, 2) # A vector  
    
  # Print matrices  
  print("Matrix A:")  
  print(A)  
  print("Matrix B:")  
  print(B)  
  print("Vector x:")  
  print(x)  
    
  # Matrix Addition  
  print("A + B:")  
  print(A + B)  
    
  # Matrix Subtraction  
  print("A - B:")  
  print(A - B)  
    
  # Matrix Multiplication (Element-wise)  
  print("A \* B (Element-wise):")  
  print(A \* B)  
    
  # Matrix Multiplication (Linear Algebra)  
  print("A %\*% B (Matrix Multiplication):")  
  print(A %\*% B)  
    
  # Matrix Transpose  
  print("Transpose of A:")  
  print(t(A))  
    
  # Matrix Inverse  
  print("Inverse of A:")  
  # Check if determinant is non-zero before inverting  
  if(det(A) != 0) {  
   print(solve(A))  
  } else {  
   print("Matrix A is singular, cannot compute inverse.")  
  }  
    
    
  # Determinant  
  print("Determinant of A:")  
  print(det(A))  
    
  # Solving Linear Equations (Ax = b)  
  # Example: Solve for y in Ay = x  
  if(det(A) != 0) {  
   y <- solve(A, x)  
   print("Solution y for Ay = x:")  
   print(y)  
   # Verification  
   print("Verification A %\*% y:")  
   print(A %\*% y)  
  } else {  
   print("Matrix A is singular, cannot solve Ay = x uniquely.")  
  }  
    
  # Eigenvalues and Eigenvectors  
  print("Eigen decomposition of A:")  
  eigen\_decomp <- eigen(A)  
  print("Eigenvalues:")  
  print(eigen\_decomp$values)  
  print("Eigenvectors:")  
  print(eigen\_decomp$vectors)
* **Output (Sample):**  
  [1] "Matrix A:"  
   [,1] [,2]  
  [1,] 1 2  
  [2,] 3 4  
  [1] "Matrix B:"  
   [,1] [,2]  
  [1,] 5 6  
  [2,] 7 8  
  [1] "Vector x:"  
  [1] 1 2  
  [1] "A + B:"  
   [,1] [,2]  
  [1,] 6 8  
  [2,] 10 12  
  [1] "A - B:"  
   [,1] [,2]  
  [1,] -4 -4  
  [2,] -4 -4  
  [1] "A \* B (Element-wise):"  
   [,1] [,2]  
  [1,] 5 12  
  [2,] 21 32  
  [1] "A %\*% B (Matrix Multiplication):"  
   [,1] [,2]  
  [1,] 19 22  
  [2,] 43 50  
  [1] "Transpose of A:"  
   [,1] [,2]  
  [1,] 1 3  
  [2,] 2 4  
  [1] "Inverse of A:"  
   [,1] [,2]  
  [1,] -2 1.0  
  [2,] 1.5 -0.5  
  [1] "Determinant of A:"  
  [1] -2  
  [1] "Solution y for Ay = x:"  
  [1] 0.0 0.5  
  [1] "Verification A %\*% y:"  
   [,1]  
  [1,] 1  
  [2,] 2  
  [1] "Eigen decomposition of A:"  
  [1] "Eigenvalues:"  
  [1] 5.372281 -0.3722813  
  [1] "Eigenvectors:"  
   [,1] [,2]  
  [1,] -0.4159736 -0.8068982  
  [2,] -0.9093767 0.5906905

**2. Python - 3Sum**

* **Aim:** To find all unique triplets in a given array of integers nums that sum up to zero.
* **Algorithm/Approach:**
  1. **Sort:** Sort the input array nums in ascending order.
  2. **Initialize:** Create an empty list result to store the triplets. Get the length n of nums.
  3. **Outer Loop:** Iterate through the sorted array with an index i from 0 up to n-3.
     + **Skip Duplicates (Outer):** If i > 0 and nums[i] is the same as nums[i-1], continue to the next iteration to avoid duplicate triplets starting with the same number.
  4. **Two Pointers:** Initialize left = i + 1 and right = n - 1.
  5. **Inner Loop:** While left < right:
     + Calculate current\_sum = nums[i] + nums[left] + nums[right].
     + **Found Triplet:** If current\_sum == 0:
       - Append the triplet [nums[i], nums[left], nums[right]] to result.
       - **Skip Duplicates (Inner):** Increment left while left < right and nums[left] equals nums[left+1]. Decrement right while left < right and nums[right] equals nums[right-1].
       - Move pointers: Increment left and decrement right to look for the next potential triplet.
     + **Adjust Pointers:**
       - If current\_sum < 0, increment left (need a larger sum).
       - If current\_sum > 0, decrement right (need a smaller sum).
  6. **Return:** Return the result list containing all unique triplets.
* **Code/Program:**  
  from typing import List  
    
  def threeSum(nums: List[int]) -> List[List[int]]:  
   nums.sort()  
   result = []  
   n = len(nums)  
   for i in range(n - 2):  
   if i > 0 and nums[i] == nums[i - 1]:  
   continue  
   left, right = i + 1, n - 1  
   while left < right:  
   current\_sum = nums[i] + nums[left] + nums[right]  
   if current\_sum == 0:  
   result.append([nums[i], nums[left], nums[right]])  
   while left < right and nums[left] == nums[left + 1]:  
   left += 1  
   while left < right and nums[right] == nums[right - 1]:  
   right -= 1  
   left += 1  
   right -= 1  
   elif current\_sum < 0:  
   left += 1  
   else:  
   right -= 1  
   return result  
    
  # Example usage:  
  nums1 = [-1, 0, 1, 2, -1, -4]  
  output1 = threeSum(nums1)  
  print(f"Input: {nums1}")  
  print(f"Output: {output1}")  
    
  nums2 = [0, 1, 1]  
  output2 = threeSum(nums2)  
  print(f"Input: {nums2}")  
  print(f"Output: {output2}")  
    
  nums3 = [0, 0, 0]  
  output3 = threeSum(nums3)  
  print(f"Input: {nums3}")  
  print(f"Output: {output3}")
* **Output (Sample):**  
  Input: [-1, 0, 1, 2, -1, -4]  
  Output: [[-1, -1, 2], [-1, 0, 1]]  
  Input: [0, 1, 1]  
  Output: []  
  Input: [0, 0, 0]  
  Output: [[0, 0, 0]]

## **Experiment 2**

**1. Python - Basic Text Similarity for Deduplication (Conceptual Example)**

* **Aim:** To demonstrate a basic approach for detecting potential duplicate text content using TF-IDF vectorization and cosine similarity. (Note: This is a conceptual stand-in for the vague "Fake news deductor" prompt).
* **Algorithm/Approach:**
  1. **Import Libraries:** Import TfidfVectorizer and cosine\_similarity from sklearn.
  2. **Define Function:** Create a function check\_similarity(text1, text2, threshold) accepting two strings and a similarity threshold.
  3. **Vectorize:** Instantiate TfidfVectorizer. Use fit\_transform() on a list containing text1 and text2 to get the TF-IDF matrix.
  4. **Calculate Similarity:** Compute cosine\_similarity between the first vector (tfidf\_matrix[0:1]) and the second vector (tfidf\_matrix[1:2]).
  5. **Compare & Print:** Print the calculated similarity score. Compare the score with the threshold. Print whether the texts are considered similar or different based on the comparison. Return True if similar, False otherwise.
* **Code/Program:**  
  # Question 1: Python - Basic Text Similarity for Deduplication (Conceptual Example)  
  # Note: "Fake news deductor" is vague. This shows basic text similarity.  
  # A real system would be much more complex.  
    
  from sklearn.feature\_extraction.text import TfidfVectorizer  
  from sklearn.metrics.pairwise import cosine\_similarity  
    
  def check\_similarity(text1, text2, threshold=0.8):  
   vectorizer = TfidfVectorizer()  
   tfidf\_matrix = vectorizer.fit\_transform([text1, text2])  
   similarity = cosine\_similarity(tfidf\_matrix[0:1], tfidf\_matrix[1:2])  
   print(f"Similarity score: {similarity[0][0]:.4f}")  
   if similarity[0][0] > threshold:  
   print("Texts are considered similar (potential duplicate).")  
   return True  
   else:  
   print("Texts are considered different.")  
   return False  
    
  # Example Usage:  
  news1 = "The prime minister announced new economic policies today."  
  news2 = "New economic policies were announced by the prime minister."  
  news3 = "Local sports team wins championship game."  
    
  print("Comparing news1 and news2:")  
  check\_similarity(news1, news2)  
    
  print("\nComparing news1 and news3:")  
  check\_similarity(news1, news3)
* **Output (Sample):**  
  Comparing news1 and news2:  
  Similarity score: 0.7554  
  Texts are considered different.  
    
  Comparing news1 and news3:  
  Similarity score: 0.0000  
  Texts are considered different.  
    
  *(Note: The threshold might need adjustment based on the specific texts and desired sensitivity. A higher score would indicate greater similarity)*

**2. Python - Add Two Numbers (Linked List)**

* **Aim:** To add two non-negative integers represented by linked lists, where digits are stored in reverse order, and return the sum as a new linked list.
* **Algorithm/Approach:**
  1. **Define Node:** Define a ListNode class with val and next attributes.
  2. **Initialize:** Create a dummy\_head node for the result list and a current pointer pointing to it. Initialize carry = 0.
  3. **Loop:** Iterate using a while loop that continues as long as l1 is not null, l2 is not null, or carry is non-zero.
  4. **Get Values:** Inside the loop, get the value of the current l1 node (val1) or 0 if l1 is null. Do the same for l2 (val2).
  5. **Calculate Sum:** Compute total\_sum = val1 + val2 + carry.
  6. **Update Carry:** Calculate the new carry = total\_sum // 10.
  7. **Calculate Digit:** Determine the digit for the new node: digit = total\_sum % 10.
  8. **Create Node:** Create a new ListNode with the calculated digit.
  9. **Link Node:** Set current.next to the new node and advance current to current.next.
  10. **Advance Lists:** Move l1 to l1.next if l1 is not null. Move l2 to l2.next if l2 is not null.
  11. **Return Result:** After the loop, return dummy\_head.next, which points to the head of the actual sum list.
* **Code/Program:**  
  # Question 2: Python - Add Two Numbers (Linked List)  
    
  from typing import Optional  
    
  class ListNode:  
   def \_\_init\_\_(self, val=0, next=None):  
   self.val = val  
   self.next = next  
    
  def addTwoNumbers(l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:  
   dummy\_head = ListNode(0)  
   current = dummy\_head  
   carry = 0  
   while l1 or l2 or carry:  
   val1 = l1.val if l1 else 0  
   val2 = l2.val if l2 else 0  
    
   total\_sum = val1 + val2 + carry  
   carry = total\_sum // 10  
   digit = total\_sum % 10  
    
   current.next = ListNode(digit)  
   current = current.next  
    
   if l1:  
   l1 = l1.next  
   if l2:  
   l2 = l2.next  
    
   return dummy\_head.next  
    
  # Helper function to create linked list from list  
  def create\_linked\_list(nums: list) -> Optional[ListNode]:  
   if not nums:  
   return None  
   head = ListNode(nums[0])  
   current = head  
   for i in range(1, len(nums)):  
   current.next = ListNode(nums[i])  
   current = current.next  
   return head  
    
  # Helper function to print linked list  
  def print\_linked\_list(head: Optional[ListNode]):  
   nums = []  
   current = head  
   while current:  
   nums.append(current.val)  
   current = current.next  
   print(nums)  
    
  # Example usage:  
  l1\_list = [2, 4, 3]  
  l2\_list = [5, 6, 4]  
  l1 = create\_linked\_list(l1\_list)  
  l2 = create\_linked\_list(l2\_list)  
    
  print("Input l1:", l1\_list)  
  print("Input l2:", l2\_list)  
  result\_list = addTwoNumbers(l1, l2)  
  print("Output:", end=" ")  
  print\_linked\_list(result\_list) # Output: [7, 0, 8]  
    
  l1\_list = [0]  
  l2\_list = [0]  
  l1 = create\_linked\_list(l1\_list)  
  l2 = create\_linked\_list(l2\_list)  
  print("\nInput l1:", l1\_list)  
  print("Input l2:", l2\_list)  
  result\_list = addTwoNumbers(l1, l2)  
  print("Output:", end=" ")  
  print\_linked\_list(result\_list) # Output: [0]  
    
  l1\_list = [9,9,9,9,9,9,9]  
  l2\_list = [9,9,9,9]  
  l1 = create\_linked\_list(l1\_list)  
  l2 = create\_linked\_list(l2\_list)  
  print("\nInput l1:", l1\_list)  
  print("Input l2:", l2\_list)  
  result\_list = addTwoNumbers(l1, l2)  
  print("Output:", end=" ")  
  print\_linked\_list(result\_list) # Output: [8, 9, 9, 9, 0, 0, 0, 1]
* **Output (Sample):**  
  Input l1: [2, 4, 3]  
  Input l2: [5, 6, 4]  
  Output: [7, 0, 8]  
    
  Input l1: [0]  
  Input l2: [0]  
  Output: [0]  
    
  Input l1: [9, 9, 9, 9, 9, 9, 9]  
  Input l2: [9, 9, 9, 9]  
  Output: [8, 9, 9, 9, 0, 0, 0, 1]

## **Experiment 3**

**1. R - Statistical and Machine Learning Functions**

* **Aim:** To demonstrate the use of R for various common statistical calculations (mean, median, sd, correlation, t-test) and basic machine learning tasks (linear regression, k-means clustering).
* **Algorithm/Approach:**
  1. **Prepare Data:** Create a sample numeric vector (data\_vector) and a sample data frame (data\_frame) with correlated columns x and y.
  2. **Basic Stats:** Calculate and print mean (mean()), median (median()), standard deviation (sd()), variance (var()) for the vector. Print a summary() of the data frame. Calculate and print the correlation (cor()) between x and y.
  3. **Statistical Test:** Perform a one-sample t-test (t.test()) on data\_frame$x to test if its mean is different from a hypothetical value (e.g., 10). Print the result.
  4. **Linear Regression:** Fit a linear model (lm()) predicting y from x using the data frame. Print the summary() of the model, which includes coefficients, R-squared, etc.
  5. **K-Means Clustering:**
     + Select the relevant columns (x, y) for clustering.
     + Set a random seed (set.seed()) for reproducibility.
     + Perform k-means clustering using kmeans() with a specified number of centers (e.g., 3).
     + Print the clustering results (cluster sizes, centers).
     + Optionally, add the cluster assignments back to the data frame and print the head.
* **Code/Program:**  
  # Question 1: R - Statistical and Machine Learning Functions  
    
  # Sample Data  
  data\_vector <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)  
  data\_frame <- data.frame(  
   x = rnorm(50, mean = 10, sd = 2),  
   y = rnorm(50, mean = 5, sd = 1)  
  )  
  data\_frame$y <- data\_frame$y + 0.5 \* data\_frame$x + rnorm(50, 0, 0.5) # Add some correlation  
    
  print("Sample Vector:")  
  print(data\_vector)  
  print("Sample Data Frame (head):")  
  print(head(data\_frame))  
    
  # --- Basic Statistics ---  
  print("--- Basic Statistics ---")  
  print(paste("Mean of vector:", mean(data\_vector)))  
  print(paste("Median of vector:", median(data\_vector)))  
  print(paste("Standard Deviation of vector:", sd(data\_vector)))  
  print(paste("Variance of vector:", var(data\_vector)))  
  print("Summary of data frame:")  
  print(summary(data\_frame))  
  print("Correlation between x and y:")  
  print(cor(data\_frame$x, data\_frame$y))  
    
  # --- Statistical Tests (Example: T-test) ---  
  print("--- Statistical Tests ---")  
  # Test if mean of x is different from 10  
  t\_test\_result <- t.test(data\_frame$x, mu = 10)  
  print("T-test result (mean of x vs 10):")  
  print(t\_test\_result)  
    
  # --- Machine Learning (Example: Linear Regression) ---  
  print("--- Machine Learning ---")  
  # Fit a linear model: y ~ x  
  lm\_model <- lm(y ~ x, data = data\_frame)  
  print("Linear Regression Model (y ~ x):")  
  print(summary(lm\_model))  
    
  # --- Machine Learning (Example: K-Means Clustering) ---  
  # Use only x and y for clustering  
  clustering\_data <- data\_frame[, c("x", "y")]  
  # Perform k-means with k=3  
  set.seed(123) # for reproducibility  
  kmeans\_result <- kmeans(clustering\_data, centers = 3)  
  print("K-Means Clustering Result (k=3):")  
  print(paste("Cluster sizes:", paste(kmeans\_result$size, collapse=", ")))  
  print("Cluster centers:")  
  print(kmeans\_result$centers)  
  # Add cluster assignment to data frame  
  data\_frame$cluster <- kmeans\_result$cluster  
  print("Data Frame with Cluster Assignment (head):")  
  print(head(data\_frame))
* **Output (Sample - structure and typical values shown):**  
  [1] "Sample Vector:"  
   [1] 1 2 3 4 5 6 7 8 9 10 10 9 8 7 6 5 4 3 2 1  
  [1] "Sample Data Frame (head):"  
  # (Head of data\_frame)  
  [1] "--- Basic Statistics ---"  
  [1] "Mean of vector: 5.5"  
  [1] "Median of vector: 5.5"  
  [1] "Standard Deviation of vector: 3.02765035409749"  
  [1] "Variance of vector: 9.16666666666667"  
  [1] "Summary of data frame:"  
  # (Summary output for data\_frame$x and data\_frame$y)  
  [1] "Correlation between x and y:"  
  # (Correlation coefficient)  
  [1] "--- Statistical Tests ---"  
  [1] "T-test result (mean of x vs 10):"  
  # (Output of t.test function)  
  [1] "--- Machine Learning ---"  
  [1] "Linear Regression Model (y ~ x):"  
  # (Output of summary.lm function including coefficients, R-squared etc.)  
  [1] "K-Means Clustering Result (k=3):"  
  [1] "Cluster sizes: 18, 16, 16"  
  [1] "Cluster centers:"  
  # (Cluster centers matrix)  
  [1] "Data Frame with Cluster Assignment (head):"  
  # (Head of data\_frame with cluster column)

**2. Python - Longest Substring Without Repeating Characters**

* **Aim:** To find the length of the longest substring within a given string s that does not contain any repeating characters.
* **Algorithm/Approach:** Sliding Window Technique:
  1. **Initialize:** Create an empty set char\_set to track characters in the current window. Set left = 0 (start of the window) and max\_length = 0.
  2. **Iterate:** Loop through the string with a right pointer from 0 to len(s) - 1.
  3. **Check for Duplicates:** Inside the loop, use a while loop: while s[right] is already in char\_set:
     + Remove the character at the left pointer (s[left]) from char\_set.
     + Increment left (shrink the window from the left).
  4. **Add Character:** Add the current character s[right] to char\_set.
  5. **Update Max Length:** Calculate the current window length (right - left + 1) and update max\_length = max(max\_length, current\_window\_length).
  6. **Return:** After the main loop finishes, return max\_length.
* **Code/Program:**  
  # Question 2: Python - Longest Substring Without Repeating Characters  
    
  def lengthOfLongestSubstring(s: str) -> int:  
   char\_set = set()  
   left = 0  
   max\_length = 0  
   for right in range(len(s)):  
   while s[right] in char\_set:  
   char\_set.remove(s[left])  
   left += 1  
   char\_set.add(s[right])  
   max\_length = max(max\_length, right - left + 1)  
   return max\_length  
    
  # Example usage:  
  s1 = "abcabcbb"  
  output1 = lengthOfLongestSubstring(s1)  
  print(f"Input: s = \"{s1}\"")  
  print(f"Output: {output1}")  
    
  s2 = "bbbbb"  
  output2 = lengthOfLongestSubstring(s2)  
  print(f"Input: s = \"{s2}\"")  
  print(f"Output: {output2}")  
    
  s3 = "pwwkew"  
  output3 = lengthOfLongestSubstring(s3)  
  print(f"Input: s = \"{s3}\"")  
  print(f"Output: {output3}")  
    
  s4 = ""  
  output4 = lengthOfLongestSubstring(s4)  
  print(f"Input: s = \"{s4}\"")  
  print(f"Output: {output4}")
* **Output (Sample):**  
  Input: s = "abcabcbb"  
  Output: 3  
  Input: s = "bbbbb"  
  Output: 1  
  Input: s = "pwwkew"  
  Output: 3  
  Input: s = ""  
  Output: 0

## **Experiment 4**

**1. R - Reliability and Goodness of Fit Analysis**

* **Aim:** To demonstrate how to perform reliability analysis (using Cronbach's Alpha) for measurement scales and goodness-of-fit analysis for statistical models (using R-squared for linear regression and Chi-squared test for categorical data) in R.
* **Algorithm/Approach:**
  1. **Load Package:** Load the psych package (library(psych)). Install if necessary (install.packages("psych")).
  2. **Reliability Analysis:**
     + Prepare sample data (scale\_data) representing responses to scale items (e.g., a data frame with columns as items).
     + Calculate Cronbach's Alpha using alpha(scale\_data).
     + Print the results. The raw\_alpha value indicates the scale reliability.
  3. **Goodness of Fit (Linear Regression):**
     + Prepare sample data (regression\_data) with predictor (x) and response (y\_actual) variables.
     + Fit a linear model using lm(y\_actual ~ x, data = regression\_data).
     + Get the model summary using summary(lm\_model).
     + Print the summary.
     + Extract and print specific metrics like R-squared (model\_summary$r.squared), Adjusted R-squared (model\_summary$adj.r.squared), F-statistic, and the model's p-value.
  4. **Goodness of Fit (Chi-Squared):**
     + Prepare a contingency table (observed) using matrix(). Assign dimension names if needed.
     + Perform the Chi-squared test using chisq.test(observed).
     + Print the results, which include the chi-squared statistic, degrees of freedom, and p-value.
* **Code/Program:**  
  # Question 1: R - Reliability and Goodness of Fit Analysis  
    
  # Note: Requires 'psych' package for Cronbach's Alpha. Install if needed.  
  # install.packages("psych")  
  library(psych)  
    
  # --- Reliability Analysis (Cronbach's Alpha) ---  
  # Example: Assume we have a scale with 3 items (e.g., from a survey)  
  # Create sample data representing responses to these items  
  set.seed(123)  
  item1 <- sample(1:5, 100, replace = TRUE)  
  item2 <- item1 + sample(-1:1, 100, replace = TRUE) # Correlated item  
  item3 <- sample(1:5, 100, replace = TRUE) # Less correlated item  
  item2[item2 < 1] <- 1  
  item2[item2 > 5] <- 5  
  scale\_data <- data.frame(item1, item2, item3)  
    
  print("Sample Scale Data (head):")  
  print(head(scale\_data))  
    
  # Calculate Cronbach's Alpha  
  alpha\_result <- alpha(scale\_data)  
  print("Reliability Analysis (Cronbach's Alpha):")  
  print(alpha\_result)  
  # Look for 'raw\_alpha' in the output summary  
    
  # --- Goodness of Fit (Example: Linear Regression R-squared) ---  
  # Create sample data for regression  
  set.seed(456)  
  x <- rnorm(100, 10, 2)  
  y\_actual <- 5 + 2 \* x + rnorm(100, 0, 3) # y = 5 + 2x + error  
  regression\_data <- data.frame(x, y\_actual)  
    
  print("Sample Regression Data (head):")  
  print(head(regression\_data))  
    
  # Fit a linear model  
  lm\_model <- lm(y\_actual ~ x, data = regression\_data)  
    
  print("Goodness of Fit Analysis (Linear Model):")  
  model\_summary <- summary(lm\_model)  
  print(model\_summary)  
    
  # Extract specific goodness-of-fit statistics  
  r\_squared <- model\_summary$r.squared  
  adj\_r\_squared <- model\_summary$adj.r.squared  
  f\_statistic <- model\_summary$fstatistic  
  # Correct way to calculate p-value from F-statistic summary object  
  p\_value <- pf(f\_statistic[1], f\_statistic[2], f\_statistic[3], lower.tail = FALSE)  
    
    
  print(paste("R-squared:", round(r\_squared, 4)))  
  print(paste("Adjusted R-squared:", round(adj\_r\_squared, 4)))  
  print(paste("F-statistic value:", round(f\_statistic[1], 2)))  
  print(paste("Model p-value:", format.pval(p\_value, digits = 4)))  
    
    
  # --- Goodness of Fit (Example: Chi-Squared for Categorical Data) ---  
  # Create sample contingency table  
  observed <- matrix(c(50, 30, 20, 40), nrow = 2, byrow = TRUE)  
  dimnames(observed) <- list(Group = c("A", "B"), Outcome = c("Success", "Failure"))  
  print("Sample Contingency Table:")  
  print(observed)  
    
  # Perform Chi-Squared Test  
  chisq\_result <- chisq.test(observed)  
  print("Goodness of Fit (Chi-Squared Test):")  
  print(chisq\_result)
* **Output (Sample - structure and typical values shown):**  
  [1] "Sample Scale Data (head):"  
  # (Head of scale\_data)  
  [1] "Reliability Analysis (Cronbach's Alpha):"  
  # (Output of alpha function, including raw\_alpha)  
  [1] "Sample Regression Data (head):"  
  # (Head of regression\_data)  
  [1] "Goodness of Fit Analysis (Linear Model):"  
  # (Output of summary.lm function)  
  [1] "R-squared: ..."  
  [1] "Adjusted R-squared: ..."  
  [1] "F-statistic value: ..."  
  [1] "Model p-value: ..."  
  [1] "Sample Contingency Table:"  
   Outcome  
  Group Success Failure  
   A 50 30  
   B 20 40  
  [1] "Goodness of Fit (Chi-Squared Test):"  
  # (Output of chisq.test function, including chi-squared value, df, p-value)

**2. Python - Group Anagrams**

* **Aim:** To group an array of strings strs such that all anagrams are together in sub-lists.
* **Algorithm/Approach:**
  1. **Import:** Import defaultdict from collections.
  2. **Initialize Map:** Create a defaultdict(list) called anagram\_map. This dictionary will store the sorted string as the key and a list of its anagrams as the value.
  3. **Iterate Strings:** Loop through each string s in the input list strs.
  4. **Sort String:** Create a canonical representation (key) by sorting the characters of s alphabetically and joining them back into a string (sorted\_s = "".join(sorted(s))).
  5. **Append to Map:** Append the original string s to the list associated with sorted\_s in the anagram\_map.
  6. **Return Groups:** After the loop, the values of anagram\_map contain the lists of grouped anagrams. Return list(anagram\_map.values()).
* **Code/Program:**  
  # Question 2: Python - Group Anagrams  
    
  from typing import List  
  from collections import defaultdict  
    
  def groupAnagrams(strs: List[str]) -> List[List[str]]:  
   anagram\_map = defaultdict(list)  
   for s in strs:  
   sorted\_s = "".join(sorted(s))  
   anagram\_map[sorted\_s].append(s)  
   return list(anagram\_map.values())  
    
  # Example usage:  
  strs1 = ["eat", "tea", "tan", "ate", "nat", "bat"]  
  output1 = groupAnagrams(strs1)  
  print(f"Input: {strs1}")  
  print(f"Output: {output1}")  
    
  strs2 = [""]  
  output2 = groupAnagrams(strs2)  
  print(f"Input: {strs2}")  
  print(f"Output: {output2}")  
    
  strs3 = ["a"]  
  output3 = groupAnagrams(strs3)  
  print(f"Input: {strs3}")  
  print(f"Output: {output3}")
* **Output (Sample):**  
  Input: ['eat', 'tea', 'tan', 'ate', 'nat', 'bat']  
  Output: [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']] # Order of groups/elements within groups may vary  
  Input: ['']  
  Output: [['']]  
  Input: ['a']  
  Output: [['a']]

## **Experiment 5**

**1. Python - Primitive Datatypes**

* **Aim:** To demonstrate the declaration, initialization, and basic operations of Python's primitive data types: integer (int), float (float), string (str), boolean (bool), and NoneType.
* **Algorithm/Approach:**
  1. **Integer (int):** Declare an integer variable. Print its value and type (type()). Perform and print addition/multiplication.
  2. **Float (float):** Declare a float variable. Print its value and type. Perform and print division.
  3. **String (str):** Declare a string variable. Print its value and type. Perform and print concatenation (+), conversion to uppercase (.upper()), length calculation (len()), and slicing ([start:end]).
  4. **Boolean (bool):** Declare boolean variables (True, False). Print their values and types. Perform and print logical AND (and), OR (or), and NOT (not) operations.
  5. **NoneType (None):** Declare a variable assigned to None. Print its value and type.
* **Code/Program:**  
  # Question 1: Python - Primitive Datatypes  
    
  # Integer  
  my\_int = 10  
  print(f"Integer: {my\_int}, Type: {type(my\_int)}")  
  int\_sum = my\_int + 5  
  print(f"Integer Sum (10 + 5): {int\_sum}")  
  int\_prod = my\_int \* 2  
  print(f"Integer Product (10 \* 2): {int\_prod}")  
    
  # Float  
  my\_float = 3.14  
  print(f"\nFloat: {my\_float}, Type: {type(my\_float)}")  
  float\_div = my\_float / 2  
  print(f"Float Division (3.14 / 2): {float\_div}")  
    
  # String  
  my\_string = "Hello Python"  
  print(f"\nString: {my\_string}, Type: {type(my\_string)}")  
  string\_concat = my\_string + " World"  
  print(f"String Concatenation: {string\_concat}")  
  print(f"String Uppercase: {my\_string.upper()}")  
  print(f"String Length: {len(my\_string)}")  
  print(f"String Slice [0:5]: {my\_string[0:5]}")  
    
  # Boolean  
  my\_bool\_true = True  
  my\_bool\_false = False  
  print(f"\nBoolean True: {my\_bool\_true}, Type: {type(my\_bool\_true)}")  
  print(f"Boolean False: {my\_bool\_false}, Type: {type(my\_bool\_false)}")  
  print(f"Logical AND (True and False): {my\_bool\_true and my\_bool\_false}")  
  print(f"Logical OR (True or False): {my\_bool\_true or my\_bool\_false}")  
  print(f"Logical NOT (not True): {not my\_bool\_true}")  
    
  # NoneType  
  my\_none = None  
  print(f"\nNoneType: {my\_none}, Type: {type(my\_none)}")
* **Output (Sample):**  
  Integer: 10, Type: <class 'int'>  
  Integer Sum (10 + 5): 15  
  Integer Product (10 \* 2): 20  
    
  Float: 3.14, Type: <class 'float'>  
  Float Division (3.14 / 2): 1.57  
    
  String: Hello Python, Type: <class 'str'>  
  String Concatenation: Hello Python World  
  String Uppercase: HELLO PYTHON  
  String Length: 12  
  String Slice [0:5]: Hello  
    
  Boolean True: True, Type: <class 'bool'>  
  Boolean False: False, Type: <class 'bool'>  
  Logical AND (True and False): False  
  Logical OR (True or False): True  
  Logical NOT (not True): False  
    
  NoneType: None, Type: <class 'NoneType'>

**2. Python - Top K Frequent Elements**

* **Aim:** To find the k most frequently occurring elements in a given integer array nums.
* **Algorithm/Approach:** Using Hash Map and Min-Heap:
  1. **Import:** Import Counter from collections and heapq.
  2. **Handle Empty Input:** If nums is empty, return an empty list.
  3. **Count Frequencies:** Use Counter(nums) to create a frequency map (count) of elements.
  4. **Initialize Heap:** Create an empty list min\_heap which will function as a min-heap.
  5. **Populate Heap:** Iterate through the (num, freq) items in the count map:
     + Push the tuple (freq, num) onto min\_heap using heapq.heappush(). Frequency comes first for heap ordering based on frequency.
     + **Maintain Size:** If len(min\_heap) is greater than k, remove the element with the smallest frequency using heapq.heappop(min\_heap).
  6. **Extract Result:** After iterating through all items, min\_heap contains the k elements with the highest frequencies. Create a list comprehension [num for freq, num in min\_heap] to extract just the numbers.
  7. **Return:** Return the resulting list.
* **Code/Program:**  
  # Question 2: Python - Top K Frequent Elements  
    
  from typing import List  
  from collections import Counter  
  import heapq  
    
  def topKFrequent(nums: List[int], k: int) -> List[int]:  
   if not nums:  
   return []  
    
   count = Counter(nums)  
   # Use a min-heap of size k  
   # Store tuples as (frequency, number)  
   min\_heap = []  
   for num, freq in count.items():  
   heapq.heappush(min\_heap, (freq, num))  
   if len(min\_heap) > k:  
   heapq.heappop(min\_heap) # Remove the element with the smallest frequency  
    
   # The heap now contains the k elements with the highest frequencies  
   result = [num for freq, num in min\_heap]  
   return result  
    
  # Example usage:  
  nums1 = [1, 1, 1, 2, 2, 3]  
  k1 = 2  
  output1 = topKFrequent(nums1, k1)  
  print(f"Input: nums = {nums1}, k = {k1}")  
  print(f"Output: {output1}")  
    
  nums2 = [1]  
  k2 = 1  
  output2 = topKFrequent(nums2, k2)  
  print(f"Input: nums = {nums2}, k = {k2}")  
  print(f"Output: {output2}")  
    
  nums3 = [4, 1, -1, 2, -1, 2, 3]  
  k3 = 2  
  output3 = topKFrequent(nums3, k3)  
  print(f"Input: nums = {nums3}, k = {k3}")  
  print(f"Output: {output3}") # Output: [2, -1] or [-1, 2]
* **Output (Sample):**  
  Input: nums = [1, 1, 1, 2, 2, 3], k = 2  
  Output: [1, 2] # Order may vary  
  Input: nums = [1], k = 1  
  Output: [1]  
  Input: nums = [4, 1, -1, 2, -1, 2, 3], k = 2  
  Output: [-1, 2] # Order may vary

## **Experiment 6**

**1. Python - Control Statements**

* **Aim:** To demonstrate the usage of Python's control flow statements: conditional statements (if, elif, else), loops (for, while), and loop control statements (break, continue, pass).
* **Algorithm/Approach:**
  1. **Conditional (if-elif-else):** Define a variable (e.g., score). Use if, elif, and else blocks to assign a value to another variable (e.g., grade) based on the score. Print the result. Repeat with a different score to show different branches.
  2. **for Loop:**
     + Iterate over a sequence (e.g., a list of strings) using for item in sequence:. Print each item.
     + Iterate over a range of numbers using for i in range(n):. Print each number.
  3. **while Loop:** Initialize a counter variable. Use while condition: to loop as long as the condition is true. Print the counter and increment it within the loop to eventually terminate.
  4. **Loop Control (break, continue):**
     + Use a for loop. Inside the loop, use an if condition to break out of the loop prematurely.
     + Use another for loop. Inside, use an if condition to continue to the next iteration, skipping the rest of the current iteration's code.
  5. **pass Statement:** Define an empty function using def function\_name(): pass. Call the function to show it executes without error. pass acts as a placeholder where syntax requires a statement but no action is needed.
* **Code/Program:**  
  # Question 1: Python - Control Statements  
    
  # --- If-Elif-Else ---  
  print("--- If-Elif-Else ---")  
  score = 75  
  if score >= 90:  
   grade = "A"  
  elif score >= 80:  
   grade = "B"  
  elif score >= 70:  
   grade = "C"  
  else:  
   grade = "D"  
  print(f"Score: {score}, Grade: {grade}")  
    
  score = 50  
  if score >= 90:  
   grade = "A"  
  elif score >= 80:  
   grade = "B"  
  elif score >= 70:  
   grade = "C"  
  else:  
   grade = "D"  
  print(f"Score: {score}, Grade: {grade}")  
    
  # --- For Loop ---  
  print("\n--- For Loop ---")  
  my\_list = ["apple", "banana", "cherry"]  
  print("Iterating through list:")  
  for fruit in my\_list:  
   print(fruit)  
    
  print("\nIterating through range:")  
  for i in range(5): # 0 to 4  
   print(i, end=" ")  
  print()  
    
  # --- While Loop ---  
  print("\n--- While Loop ---")  
  count = 0  
  print("Counting up to 3:")  
  while count < 3:  
   print(count, end=" ")  
   count += 1  
  print()  
    
  # --- Break and Continue ---  
  print("\n--- Break and Continue ---")  
  print("Loop with break at 5:")  
  for i in range(10):  
   if i == 5:  
   break  
   print(i, end=" ")  
  print()  
    
  print("\nLoop with continue at 3:")  
  for i in range(6):  
   if i == 3:  
   continue  
   print(i, end=" ")  
  print()  
    
  # --- Pass Statement ---  
  print("\n--- Pass Statement ---")  
  def my\_empty\_function():  
   pass # Placeholder, does nothing  
    
  my\_empty\_function()  
  print("Empty function called (using pass).")
* **Output (Sample):**  
  --- If-Elif-Else ---  
  Score: 75, Grade: C  
  Score: 50, Grade: D  
    
  --- For Loop ---  
  Iterating through list:  
  apple  
  banana  
  cherry  
    
  Iterating through range:  
  0 1 2 3 4  
    
  --- While Loop ---  
  Counting up to 3:  
  0 1 2  
    
  --- Break and Continue ---  
  Loop with break at 5:  
  0 1 2 3 4  
    
  Loop with continue at 3:  
  0 1 2 4 5  
    
  --- Pass Statement ---  
  Empty function called (using pass).

**2. Python - Binary Tree Inorder and Postorder Traversal**

* **Aim:** To perform inorder (Left-Node-Right) and postorder (Left-Right-Node) traversals of a given binary tree and return the node values in the order they are visited.
* **Algorithm/Approach:** Recursive Traversal:
  1. **Define Node:** Define a TreeNode class with val, left, and right attributes.
  2. **Inorder Function (inorderTraversal):**
     + Initialize an empty list result.
     + Define a nested helper function traverse(node):
       - **Base Case:** If node is null, return.
       - **Recurse Left:** Call traverse(node.left).
       - **Visit Node:** Append node.val to result.
       - **Recurse Right:** Call traverse(node.right).
     + Call the helper function starting with the root: traverse(root).
     + Return result.
  3. **Postorder Function (postorderTraversal):**
     + Initialize an empty list result.
     + Define a nested helper function traverse(node):
       - **Base Case:** If node is null, return.
       - **Recurse Left:** Call traverse(node.left).
       - **Recurse Right:** Call traverse(node.right).
       - **Visit Node:** Append node.val to result.
     + Call the helper function starting with the root: traverse(root).
     + Return result.
* **Code/Program:**  
  # Question 2: Python - Binary Tree Inorder and Postorder Traversal  
    
  from typing import List, Optional  
    
  class TreeNode:  
   def \_\_init\_\_(self, val=0, left=None, right=None):  
   self.val = val  
   self.left = left  
   self.right = right  
    
  def inorderTraversal(root: Optional[TreeNode]) -> List[int]:  
   result = []  
   def traverse(node):  
   if node:  
   traverse(node.left)  
   result.append(node.val)  
   traverse(node.right)  
   traverse(root)  
   return result  
    
  def postorderTraversal(root: Optional[TreeNode]) -> List[int]:  
   result = []  
   def traverse(node):  
   if node:  
   traverse(node.left)  
   traverse(node.right)  
   result.append(node.val)  
   traverse(root)  
   return result  
    
  # Helper function to build tree from list (simplified level order)  
  # None represents null nodes, only works for fairly complete trees easily  
  def build\_tree(nodes: List[Optional[int]]) -> Optional[TreeNode]:  
   if not nodes or nodes[0] is None:  
   return None  
   root = TreeNode(nodes[0])  
   queue = [(root, 0)]  
   head = 0  
   while head < len(queue):  
   curr\_node, index = queue[head]  
   head += 1  
    
   left\_child\_index = 2 \* index + 1  
   if left\_child\_index < len(nodes) and nodes[left\_child\_index] is not None:  
   curr\_node.left = TreeNode(nodes[left\_child\_index])  
   queue.append((curr\_node.left, left\_child\_index))  
    
   right\_child\_index = 2 \* index + 2  
   if right\_child\_index < len(nodes) and nodes[right\_child\_index] is not None:  
   curr\_node.right = TreeNode(nodes[right\_child\_index])  
   queue.append((curr\_node.right, right\_child\_index))  
   return root  
    
    
  # Example 1: root = [1,null,2,3] -> Tree: 1 -> right: 2 -> left: 3  
  root1 = TreeNode(1)  
  root1.right = TreeNode(2)  
  root1.right.left = TreeNode(3)  
    
  print("Example 1:")  
  print("Input Tree (structure): 1(R: 2(L: 3))")  
  inorder1 = inorderTraversal(root1)  
  postorder1 = postorderTraversal(root1)  
  print(f"Inorder: {inorder1}") # Expected: [1, 3, 2]  
  print(f"Postorder: {postorder1}") # Expected: [3, 2, 1]  
    
  # Example 2: root = [1,2,3,4,5,null,8,null,null,6,7,null,null,9] (approximate structure)  
  # This structure is complex to build manually, using a simplified build  
  # Let's assume a structure like:  
  # 1  
  # / \  
  # 2 3  
  # / \ \  
  # 4 5 8  
  # / \ /  
  # 6 7 9  
  root2 = TreeNode(1)  
  root2.left = TreeNode(2)  
  root2.right = TreeNode(3)  
  root2.left.left = TreeNode(4)  
  root2.left.right = TreeNode(5)  
  root2.right.right = TreeNode(8)  
  root2.left.right.left = TreeNode(6)  
  root2.left.right.right = TreeNode(7)  
  root2.right.right.left = TreeNode(9)  
    
    
  print("\nExample 2:")  
  print("Input Tree (structure): 1(L:2(L:4, R:5(L:6, R:7)), R:3(R:8(L:9)))")  
  inorder2 = inorderTraversal(root2)  
  postorder2 = postorderTraversal(root2)  
  print(f"Inorder: {inorder2}") # Expected: [4, 2, 6, 5, 7, 1, 3, 9, 8]  
  print(f"Postorder: {postorder2}") # Expected: [4, 6, 7, 5, 2, 9, 8, 3, 1] (Note: Example in prompt might be slightly off for postorder)
* **Output (Sample):**  
  Example 1:  
  Input Tree (structure): 1(R: 2(L: 3))  
  Inorder: [1, 3, 2]  
  Postorder: [3, 2, 1]  
    
  Example 2:  
  Input Tree (structure): 1(L:2(L:4, R:5(L:6, R:7)), R:3(R:8(L:9)))  
  Inorder: [4, 2, 6, 5, 7, 1, 3, 9, 8]  
  Postorder: [4, 6, 7, 5, 2, 9, 8, 3, 1]

## **Experiment 7**

**1. Python - Creating Functions**

* **Aim:** To demonstrate how to define and use functions in Python, including functions with parameters, return values, default arguments, variable arguments (\*args, \*\*kwargs), and lambda functions.
* **Algorithm/Approach:**
  1. **Basic Function:** Define a function greet(name) using def that takes one argument and prints a greeting. Call it.
  2. **Return Value:** Define add\_numbers(x, y) that uses return to give back the sum of x and y. Call it and print the returned value.
  3. **Default Parameter:** Define power(base, exponent=2) where exponent has a default value. Call it with one argument (using the default) and with two arguments (overriding the default). Print results.
  4. **Keyword Arguments:** Define describe\_pet(animal\_type, pet\_name). Call it once using positional arguments and once using keyword arguments (pet\_name=..., animal\_type=...) to show order doesn't matter for keywords.
  5. **Variable Positional Arguments (\*args):** Define sum\_all(\*numbers) that uses \*numbers to accept any number of positional arguments. Iterate through the numbers tuple and sum them. Call it with different numbers of arguments.
  6. **Variable Keyword Arguments (\*\*kwargs):** Define build\_profile(first, last, \*\*user\_info) that accepts required arguments and arbitrary keyword arguments via \*\*user\_info. Create and return a dictionary profile including all information. Call it with extra keyword arguments.
  7. **Lambda Function:** Define a small anonymous function multiply = lambda x, y: x \* y. Call it and print the result.
* **Code/Program:**  
  # Question 1: Python - Creating Functions  
    
  # --- Basic Function ---  
  print("--- Basic Function ---")  
  def greet(name):  
   print(f"Hello, {name}!")  
    
  greet("Alice")  
    
  # --- Function with Return Value ---  
  print("\n--- Function with Return Value ---")  
  def add\_numbers(x, y):  
   return x + y  
    
  sum\_result = add\_numbers(5, 3)  
  print(f"Sum of 5 and 3: {sum\_result}")  
    
  # --- Function with Default Parameter Value ---  
  print("\n--- Function with Default Parameter Value ---")  
  def power(base, exponent=2):  
   return base \*\* exponent  
    
  print(f"3 to the power of 2 (default): {power(3)}")  
  print(f"3 to the power of 3: {power(3, 3)}")  
    
  # --- Function with Keyword Arguments ---  
  print("\n--- Function with Keyword Arguments ---")  
  def describe\_pet(animal\_type, pet\_name):  
   print(f"I have a {animal\_type} named {pet\_name}.")  
    
  describe\_pet(animal\_type="hamster", pet\_name="Harry")  
  describe\_pet(pet\_name="Lucy", animal\_type="dog") # Order doesn't matter  
    
  # --- Function with Variable Positional Arguments (\*args) ---  
  print("\n--- Function with \*args ---")  
  def sum\_all(\*numbers):  
   total = 0  
   for num in numbers:  
   total += num  
   return total  
    
  print(f"Sum of 1, 2, 3: {sum\_all(1, 2, 3)}")  
  print(f"Sum of 10, 20, 30, 40: {sum\_all(10, 20, 30, 40)}")  
    
  # --- Function with Variable Keyword Arguments (\*\*kwargs) ---  
  print("\n--- Function with \*\*kwargs ---")  
  def build\_profile(first, last, \*\*user\_info):  
   profile = {}  
   profile['first\_name'] = first  
   profile['last\_name'] = last  
   for key, value in user\_info.items():  
   profile[key] = value  
   return profile  
    
  user\_profile = build\_profile('albert', 'einstein',  
   location='princeton',  
   field='physics')  
  print(f"User Profile: {user\_profile}")  
    
  # --- Lambda Function (Anonymous Function) ---  
  print("\n--- Lambda Function ---")  
  multiply = lambda x, y: x \* y  
  print(f"Lambda multiplication (5 \* 4): {multiply(5, 4)}")
* **Output (Sample):**  
  --- Basic Function ---  
  Hello, Alice!  
    
  --- Function with Return Value ---  
  Sum of 5 and 3: 8  
    
  --- Function with Default Parameter Value ---  
  3 to the power of 2 (default): 9  
  3 to the power of 3: 27  
    
  --- Function with Keyword Arguments ---  
  I have a hamster named Harry.  
  I have a dog named Lucy.  
    
  --- Function with \*args ---  
  Sum of 1, 2, 3: 6  
  Sum of 10, 20, 30, 40: 100  
    
  --- Function with \*\*kwargs ---  
  User Profile: {'first\_name': 'albert', 'last\_name': 'einstein', 'location': 'princeton', 'field': 'physics'}  
    
  --- Lambda Function ---  
  Lambda multiplication (5 \* 4): 20

**2. Python - Kth Smallest Element in a BST**

* **Aim:** To find the kth smallest value (1-indexed) among all node values in a given Binary Search Tree (BST).
* **Algorithm/Approach:** Iterative Inorder Traversal with Stack:
  1. **Define Node:** Define a TreeNode class.
  2. **Initialize:** Create an empty list stack. Set current = root. Initialize count = 0.
  3. **Traversal Loop:** Start a while loop that continues as long as current is not null or stack is not empty.
  4. **Go Left:** Inside the loop, have an inner while loop: while current is not null, push current onto the stack and move current = current.left.
  5. **Process Node:** When the inner loop finishes (reached the leftmost node or null), pop a node from the stack and assign it back to current.
  6. **Increment Count:** Increment the count.
  7. **Check K:** If count equals k, we have found the kth smallest element. Return current.val.
  8. **Go Right:** Move to the right subtree to continue the inorder traversal: current = current.right.
  9. **Handle Invalid K:** If the loop finishes without finding the kth element (e.g., k is out of bounds), return an indicator like -1.
* **Code/Program:**  
  # Question 2: Python - Kth Smallest Element in a BST  
    
  from typing import Optional  
    
  class TreeNode:  
   def \_\_init\_\_(self, val=0, left=None, right=None):  
   self.val = val  
   self.left = left  
   self.right = right  
    
  def kthSmallest(root: Optional[TreeNode], k: int) -> int:  
   stack = []  
   count = 0  
   current = root  
    
   while current or stack:  
   while current:  
   stack.append(current)  
   current = current.left  
    
   current = stack.pop()  
   count += 1  
   if count == k:  
   return current.val  
    
   current = current.right  
   return -1 # Should not happen if k is valid and tree is not empty  
    
  # Helper function to build tree (can be simplified for BST)  
  def insert\_into\_bst(root: Optional[TreeNode], val: int) -> TreeNode:  
   if not root:  
   return TreeNode(val)  
   if val < root.val:  
   root.left = insert\_into\_bst(root.left, val)  
   else:  
   root.right = insert\_into\_bst(root.right, val)  
   return root  
    
  def build\_bst\_from\_list(nodes: list) -> Optional[TreeNode]:  
   if not nodes:  
   return None  
   root = None  
   for val in nodes:  
   if val is not None: # Allow None for potential level-order representation, though insert handles it  
   root = insert\_into\_bst(root, val)  
   return root  
    
  # Example 1: root = [3,1,4,null,2], k = 1  
  # BST structure: 3(L:1(R:2), R:4)  
  root1 = TreeNode(3)  
  root1.left = TreeNode(1)  
  root1.right = TreeNode(4)  
  root1.left.right = TreeNode(2)  
  k1 = 1  
  output1 = kthSmallest(root1, k1)  
  print("Example 1:")  
  print("Input Tree: [3,1,4,null,2]")  
  print(f"k = {k1}")  
  print(f"Output: {output1}") # Expected: 1  
    
  # Example 2: root = [5,3,6,2,4,null,null,1], k = 3  
  # BST structure: 5(L:3(L:2(L:1), R:4), R:6)  
  root2 = TreeNode(5)  
  root2.left = TreeNode(3)  
  root2.right = TreeNode(6)  
  root2.left.left = TreeNode(2)  
  root2.left.right = TreeNode(4)  
  root2.left.left.left = TreeNode(1)  
  k2 = 3  
  output2 = kthSmallest(root2, k2)  
  print("\nExample 2:")  
  print("Input Tree: [5,3,6,2,4,null,null,1]")  
  print(f"k = {k2}")  
  print(f"Output: {output2}") # Expected: 3
* **Output (Sample):**  
  Example 1:  
  Input Tree: [3,1,4,null,2]  
  k = 1  
  Output: 1  
    
  Example 2:  
  Input Tree: [5,3,6,2,4,null,null,1]  
  k = 3  
  Output: 3

## **Experiment 8**

**1. Python - Lists and Tuples**

* **Aim:** To demonstrate the creation, access, modification (for lists), and common operations on Python's list (mutable sequence) and tuple (immutable sequence) data types.
* **Algorithm/Approach:**
  1. **List Creation & Access:** Create a list my\_list using [] with mixed data types. Print it. Access and print elements using positive ([0]) and negative ([-1]) indices. Print a slice ([1:3]).
  2. **List Modification:** Change an element using index assignment (my\_list[1] = ...). Append an element using .append(). Insert an element at a specific index using .insert(). Remove an element by value using .remove(). Remove and return an element by index using .pop(). Print the list after each modification.
  3. **List Info & Ordering:** Print the list length using len(). Create a numeric list, sort it in place using .sort(), and print it. Reverse it in place using .reverse() and print it.
  4. **Tuple Creation & Access:** Create a tuple my\_tuple using () with mixed data types. Print it. Access and print elements using indexing and slicing, similar to lists.
  5. **Tuple Immutability:** Attempt to modify an element using index assignment (my\_tuple[1] = ...) within a try...except TypeError block to demonstrate that tuples are immutable. Print the error message.
  6. **Tuple Info & Concatenation:** Print the tuple length using len(). Create a new tuple by concatenating my\_tuple with another tuple using +. Print the new tuple.
  7. **Iteration:** Use a for loop to iterate through the items in my\_tuple and print each item.
* **Code/Program:**  
  # Question 1: Python - Lists and Tuples  
    
  # --- Lists (Mutable) ---  
  print("--- Lists ---")  
  my\_list = [1, "hello", 3.14, True]  
  print(f"Original List: {my\_list}")  
    
  # Accessing elements  
  print(f"First element: {my\_list[0]}")  
  print(f"Last element: {my\_list[-1]}")  
    
  # Slicing  
  print(f"Slice [1:3]: {my\_list[1:3]}")  
    
  # Modifying elements  
  my\_list[1] = "world"  
  print(f"Modified List: {my\_list}")  
    
  # Appending  
  my\_list.append(False)  
  print(f"Appended List: {my\_list}")  
    
  # Inserting  
  my\_list.insert(2, "inserted")  
  print(f"Inserted List: {my\_list}")  
    
  # Removing (by value)  
  my\_list.remove(3.14)  
  print(f"Removed 3.14: {my\_list}")  
    
  # Popping (by index)  
  popped\_element = my\_list.pop(0)  
  print(f"Popped index 0 ('{popped\_element}'): {my\_list}")  
    
  # Length  
  print(f"Length of list: {len(my\_list)}")  
    
  # Sorting (if elements are comparable)  
  num\_list = [3, 1, 4, 1, 5, 9, 2]  
  num\_list.sort()  
  print(f"Sorted number list: {num\_list}")  
    
  # Reversing  
  num\_list.reverse()  
  print(f"Reversed number list: {num\_list}")  
    
    
  # --- Tuples (Immutable) ---  
  print("\n--- Tuples ---")  
  my\_tuple = (1, "hello", 3.14, True)  
  print(f"Original Tuple: {my\_tuple}")  
    
  # Accessing elements  
  print(f"First element: {my\_tuple[0]}")  
  print(f"Last element: {my\_tuple[-1]}")  
    
  # Slicing  
  print(f"Slice [1:3]: {my\_tuple[1:3]}")  
    
  # Attempting to modify (will cause TypeError)  
  try:  
   my\_tuple[1] = "world"  
  except TypeError as e:  
   print(f"Attempted modification failed: {e}")  
    
  # Length  
  print(f"Length of tuple: {len(my\_tuple)}")  
    
  # Concatenation (creates a new tuple)  
  new\_tuple = my\_tuple + (False, "extra")  
  print(f"Concatenated tuple: {new\_tuple}")  
    
  # Iteration (works for both lists and tuples)  
  print("Iterating through tuple:")  
  for item in my\_tuple:  
   print(item, end=" ")  
  print()
* **Output (Sample):**  
  --- Lists ---  
  Original List: [1, 'hello', 3.14, True]  
  First element: 1  
  Last element: True  
  Slice [1:3]: ['hello', 3.14]  
  Modified List: [1, 'world', 3.14, True]  
  Appended List: [1, 'world', 3.14, True, False]  
  Inserted List: [1, 'world', 'inserted', 3.14, True, False]  
  Removed 3.14: [1, 'world', 'inserted', True, False]  
  Popped index 0 ('1'): ['world', 'inserted', True, False]  
  Length of list: 4  
  Sorted number list: [1, 1, 2, 3, 4, 5, 9]  
  Reversed number list: [9, 5, 4, 3, 2, 1, 1]  
    
  --- Tuples ---  
  Original Tuple: (1, 'hello', 3.14, True)  
  First element: 1  
  Last element: True  
  Slice [1:3]: ('hello', 3.14)  
  Attempted modification failed: 'tuple' object does not support item assignment  
  Length of tuple: 4  
  Concatenated tuple: (1, 'hello', 3.14, True, False, 'extra')  
  Iterating through tuple:  
  1 hello 3.14 True

**2. Python - Word Break**

* **Aim:** To determine if a given string s can be segmented into a space-separated sequence of one or more words from a provided dictionary wordDict.
* **Algorithm/Approach:** Dynamic Programming:
  1. **Prepare Dictionary:** Convert the input list wordDict into a set (word\_set) for O(1) average time complexity lookups.
  2. **Initialize DP Array:** Get the length n of the input string s. Create a boolean array dp of size n + 1. Initialize all elements to False. dp[i] will represent whether the prefix s[0...i-1] can be segmented.
  3. **Base Case:** Set dp[0] = True, because an empty string (prefix of length 0) can always be segmented.
  4. **Outer Loop (End of Substring):** Iterate i from 1 to n (inclusive). i represents the ending index (exclusive) of the prefix substring we are currently checking (s[0...i-1]).
  5. **Inner Loop (Start of Word):** For each i, iterate j from 0 to i-1. j represents a potential starting index for the last word in the segmentation of s[0...i-1].
  6. **Check Segmentation:** Inside the inner loop, check two conditions:
     + Is the prefix s[0...j-1] segmentable? (Check if dp[j] is True).
     + Is the substring s[j:i] (from index j up to, but not including, i) present in the word\_set?
  7. **Mark DP:** If both conditions are true, it means we found a valid segmentation for the prefix s[0...i-1]. Set dp[i] = True.
  8. **Optimization:** Once dp[i] is set to True, we can break the inner loop (over j) because we only need to know if *at least one* segmentation exists for the prefix ending at i.
  9. **Return Result:** After the loops complete, dp[n] will hold whether the entire string s (i.e., s[0...n-1]) can be segmented. Return dp[n].
* **Code/Program:**  
  # Question 2: Python - Word Break  
    
  from typing import List  
    
  def wordBreak(s: str, wordDict: List[str]) -> bool:  
   word\_set = set(wordDict)  
   n = len(s)  
   dp = [False] \* (n + 1)  
   dp[0] = True # Base case: empty string can be segmented  
    
   for i in range(1, n + 1):  
   for j in range(i):  
   # Check if s[0...j-1] can be segmented (dp[j])  
   # AND if s[j...i-1] is a word in the dictionary  
   if dp[j] and s[j:i] in word\_set:  
   dp[i] = True  
   break # Found a way to segment s[0...i-1], move to next i  
    
   return dp[n]  
    
  # Example usage:  
  s1 = "leetcode"  
  wordDict1 = ["leet", "code"]  
  output1 = wordBreak(s1, wordDict1)  
  print(f"Input: s = \"{s1}\", wordDict = {wordDict1}")  
  print(f"Output: {output1}")  
    
  s2 = "applepenapple"  
  wordDict2 = ["apple", "pen"]  
  output2 = wordBreak(s2, wordDict2)  
  print(f"Input: s = \"{s2}\", wordDict = {wordDict2}")  
  print(f"Output: {output2}")  
    
  s3 = "catsandog"  
  wordDict3 = ["cats", "dog", "sand", "and", "cat"]  
  output3 = wordBreak(s3, wordDict3)  
  print(f"Input: s = \"{s3}\", wordDict = {wordDict3}")  
  print(f"Output: {output3}")  
    
  s4 = "cars"  
  wordDict4 = ["car", "ca", "rs"]  
  output4 = wordBreak(s4, wordDict4)  
  print(f"Input: s = \"{s4}\", wordDict = {wordDict4}")  
  print(f"Output: {output4}")
* **Output (Sample):**  
  Input: s = "leetcode", wordDict = ['leet', 'code']  
  Output: True  
  Input: s = "applepenapple", wordDict = ['apple', 'pen']  
  Output: True  
  Input: s = "catsandog", wordDict = ['cats', 'dog', 'sand', 'and', 'cat']  
  Output: False  
  Input: s = "cars", wordDict = ['car', 'ca', 'rs']  
  Output: True