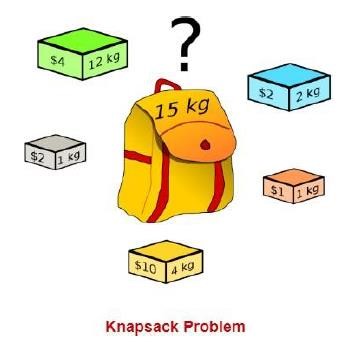
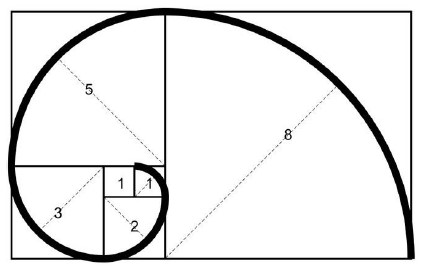
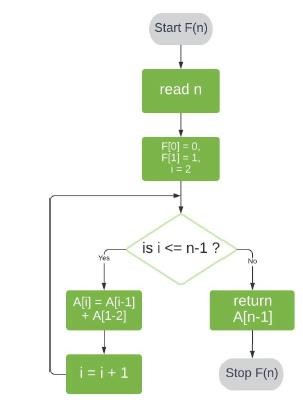
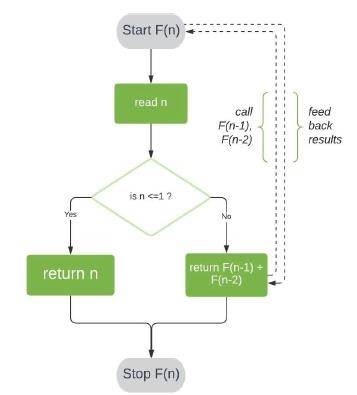
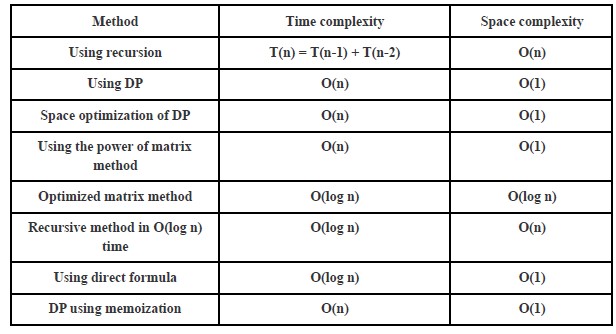
1

P

rogram

3



1 Input

def fibonacci\_iterative(n=2):

if n <= 1: return n a, b = 0, 1 for \_ in range(2, n + 1):

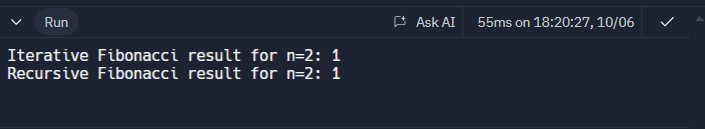
a, b = b, a + b return b

# Example call n = 2 print(f"Iterative Fibonacci result for n={n}: {fibonacci\_iterative(n)}") def fibonacci\_recursive(n=2):

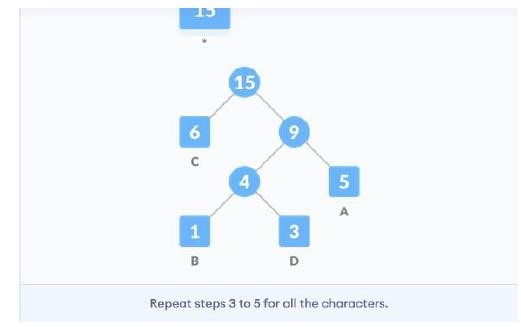
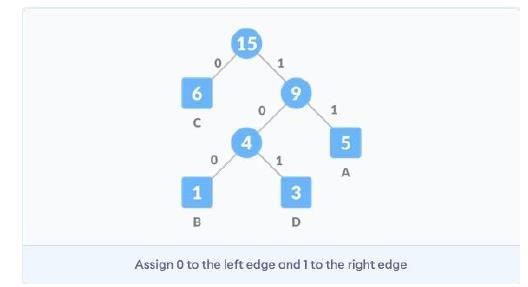
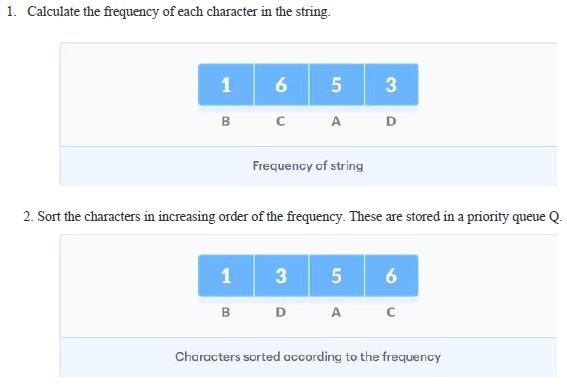
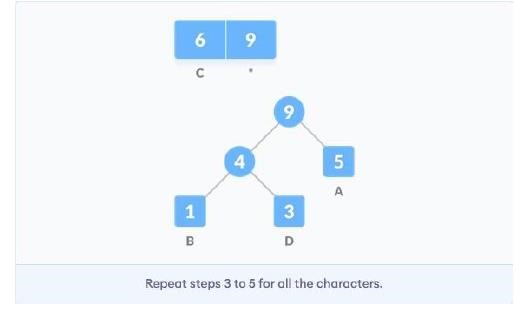
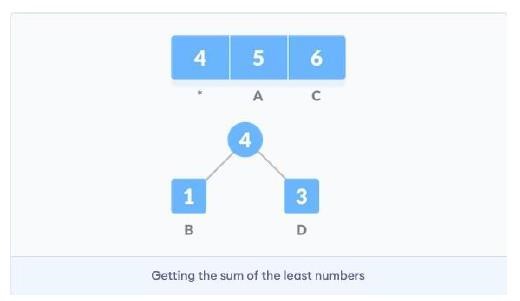
if n <= 1: return n return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

# Example call n = 2 print(f"Recursive Fibonacci result for n={n}: {fibonacci\_recursive(n)}")

Program 1 Output



2



Program 2 Input import heapq from collections import defaultdict # A Huffman Tree Node class Node: def \_\_init\_\_(self, char, freq):

self.char = char self.freq = freq self.left = None self.right = None

# Overriding < operator for priority queue def \_\_lt\_\_(self, other):

return self.freq < other.freq

# Function to build the Huffman Tree def build\_huffman\_tree(frequency): # Create a priority queue (min-heap) heap = [Node(char, freq) for char, freq in frequency.items()] heapq.heapify(heap) while len(heap) > 1:

# Extract two nodes with the lowest frequency left = heapq.heappop(heap) right = heapq.heappop(heap)

# Create a new internal node with frequency equal to the sum of the two nodes merged = Node(None, left.freq + right.freq) merged.left = left merged.right = right

# Add the new node to the heap heapq.heappush(heap, merged)

# The remaining node is the root of the Huffman Tree

return heap[0]

# Function to generate the Huffman Codes def generate\_huffman\_codes(root, current\_code, huffman\_codes):

if root is None:

return

# If it's a leaf node, store the character's code if root.char is not None:

huffman\_codes[root.char] = current\_code # Traverse the left and right subtrees generate\_huffman\_codes(root.left, current\_code + "0", huffman\_codes) generate\_huffman\_codes(root.right, current\_code + "1", huffman\_codes)

# Function to perform Huffman Encoding def huffman\_encoding(data):

if not data: return {}, ""

# Calculate frequency of each character frequency = defaultdict(int) for char in data:

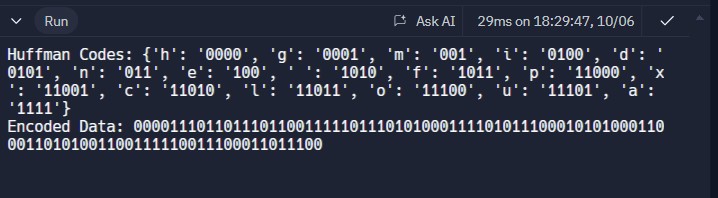
frequency[char] += 1

huffman\_tree\_root = build\_huffman\_tree(frequency) huffman\_codes = {} generate\_huffman\_codes(huffman\_tree\_root, "", huffman\_codes)

# Encode the input data encoded\_data = ''.join([huffman\_codes[char] for char in data]) return huffman\_codes, encoded\_data

# Example usage data = "huffman encoding example" huffman\_codes, encoded\_data = huffman\_encoding(data) print("Huffman Codes:", huffman\_codes) print("Encoded Data:", encoded\_data)

Program 2 Output



Program 3 Input

class Item: def \_\_init\_\_(self, value, weight):

self.value = value self.weight = weight

# To support sorting by value-to-weight ratio def \_\_lt\_\_(self, other):

return (self.value / self.weight) > (other.value / other.weight)

# Function to solve the Fractional Knapsack problem def fractional\_knapsack(capacity, items):

# Sort items by value-to-weight ratio in descending order

items.sort()

total\_value = 0 # Total value accumulated for item in items:

if capacity == 0: # If the knapsack is full break

# Take as much as we can from the current item if item.weight <= capacity: total\_value += item.value capacity -= item.weight

else:

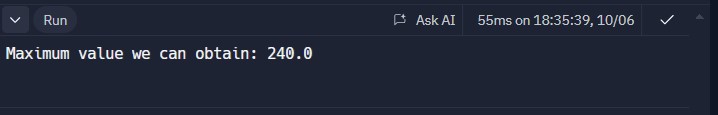
# Take the fraction of the item that fits in the remaining capacity fraction = capacity / item.weight total\_value += item.value \* fraction capacity = 0

return total\_value

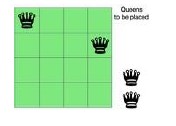
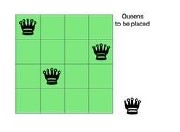
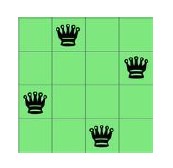
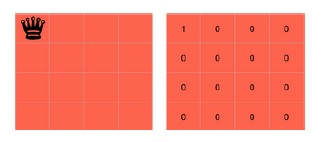
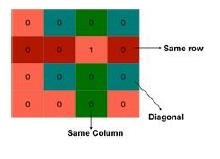
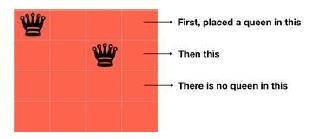
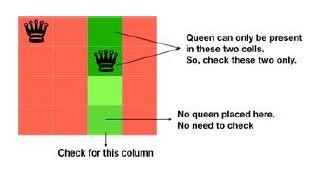
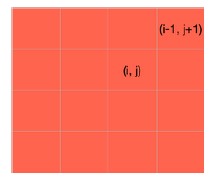
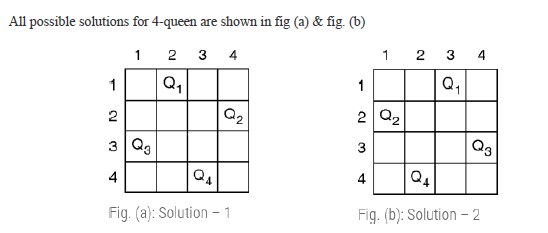
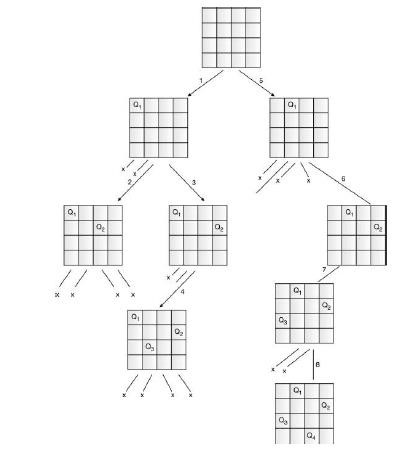
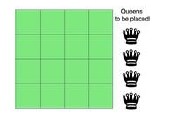
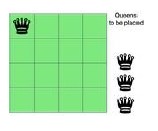
# Example usage items = [Item(60, 10), Item(100, 20), Item(120, 30)] capacity = 50

max\_value = fractional\_knapsack(capacity, items) print(f"Maximum value we can obtain: {max\_value}")

Program 3 Output



Program 4



Program 5 Input

# Function to print the n-Queens matrix def print\_board(board): for row in board: print(" ".join(row)) print()

# Check if placing a queen at board[row][col] is safe def is\_safe(board, row, col, n): # Check this row on left side for i in range(col): if board[row][i] == 'Q':

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if board[i][j] == 'Q': return False

# Check lower diagonal on left side for i, j in zip(range(row, n, 1), range(col, -1, -1)):

if board[i][j] == 'Q': return False

return True

# Recursive function to solve the n-Queens problem def solve\_n\_queens(board, col, n): # Base case: All queens are placed

if col >= n: return True

# Consider this column and try placing the queen in all rows

for i in range(n):

if is\_safe(board, i, col, n): # Place this queen in board[i][col] board[i][col] = 'Q'

# Recur to place rest of the queens if solve\_n\_queens(board, col + 1, n):

return True

# If placing the queen in board[i][col] doesn't lead to a solution board[i][col] = '.' # Backtrack return False

# Function to initialize the board with the first queen placed def place\_first\_queen(n, row, col): # Initialize board with empty cells board = [['.' for \_ in range(n)] for \_ in range(n)] board[row][col] = 'Q' # Place the first queen return board

# Driver function to solve n-Queens with the first queen placed def n\_queens\_with\_first\_queen\_placed(n, first\_row, first\_col):

board = place\_first\_queen(n, first\_row, first\_col)

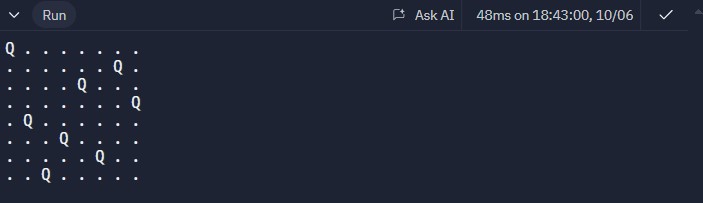
# Start solving for the remaining queens starting from the next column if not solve\_n\_queens(board, first\_col + 1, n):

print("Solution does not exist")

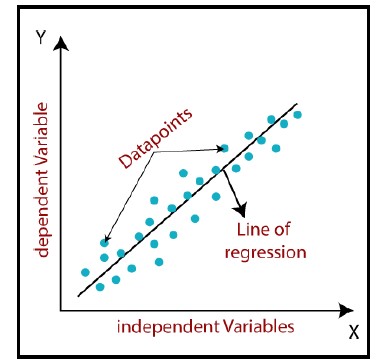
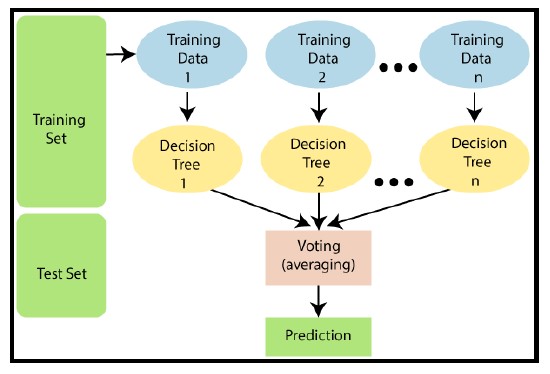
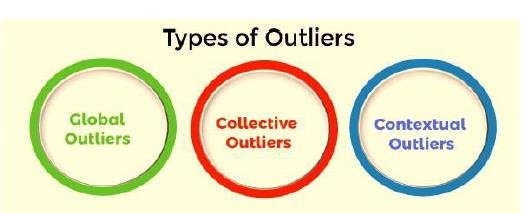
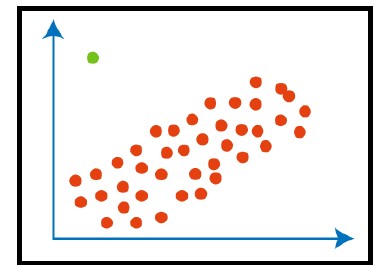
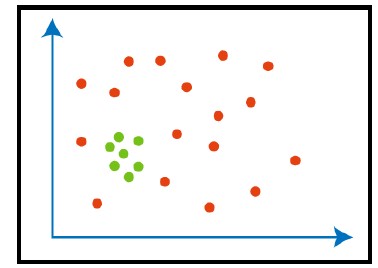
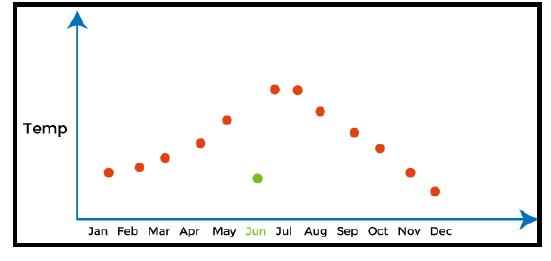
else:

print\_board(board) # Example usage

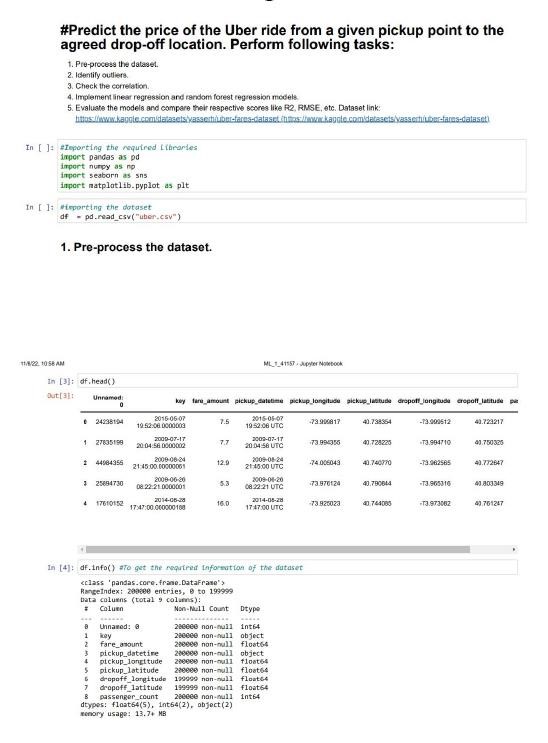
n = 8 # Size of the board first\_row, first\_col = 0, 0 # Position of the first queen n\_queens\_with\_first\_queen\_placed(n, first\_row, first\_col) Program 5 Output

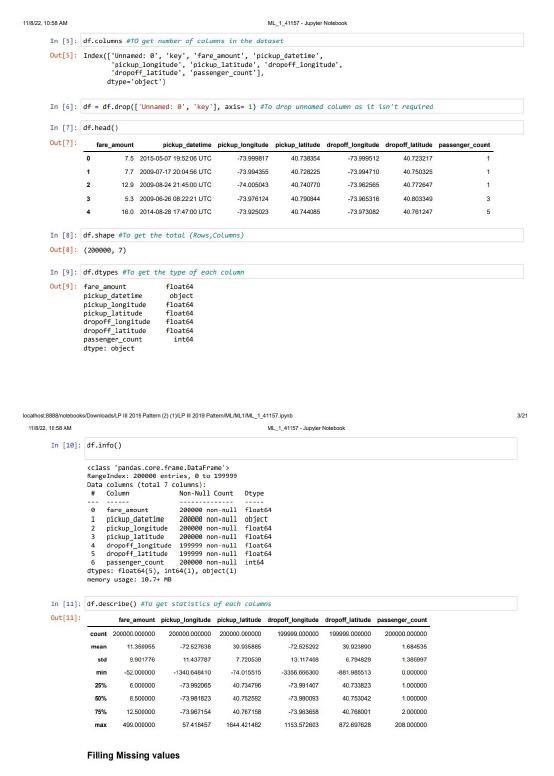


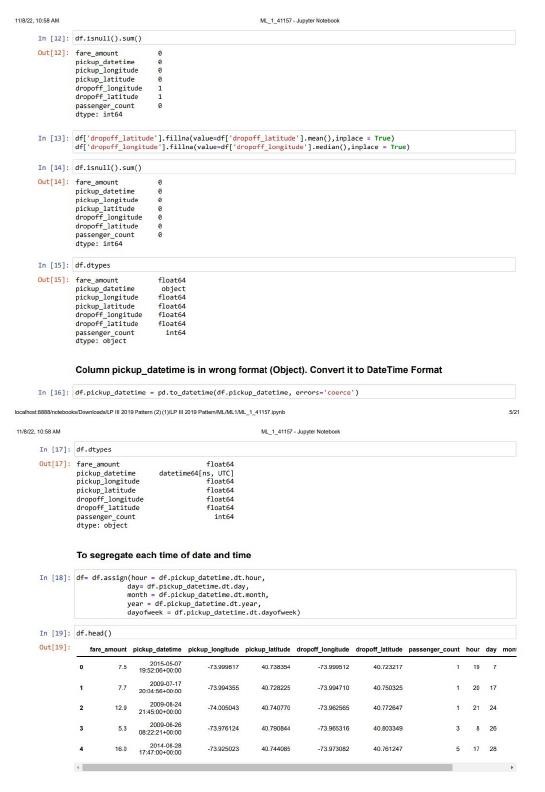
Program 6

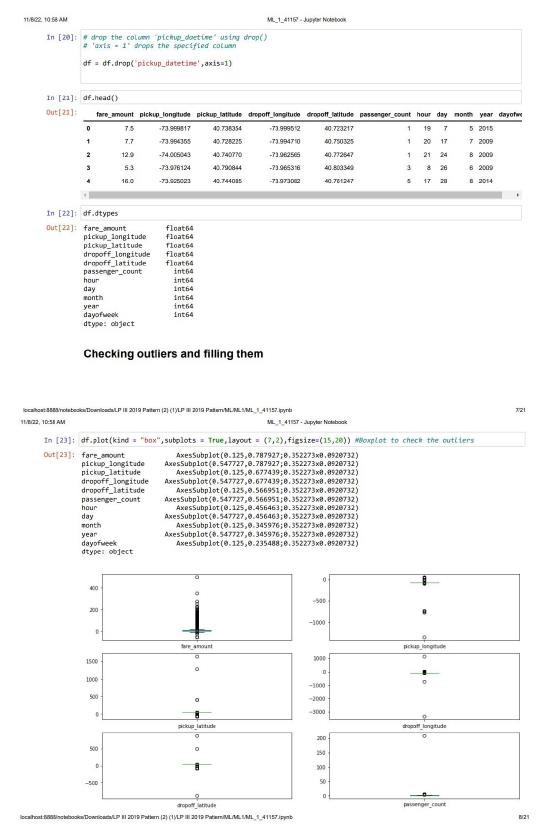


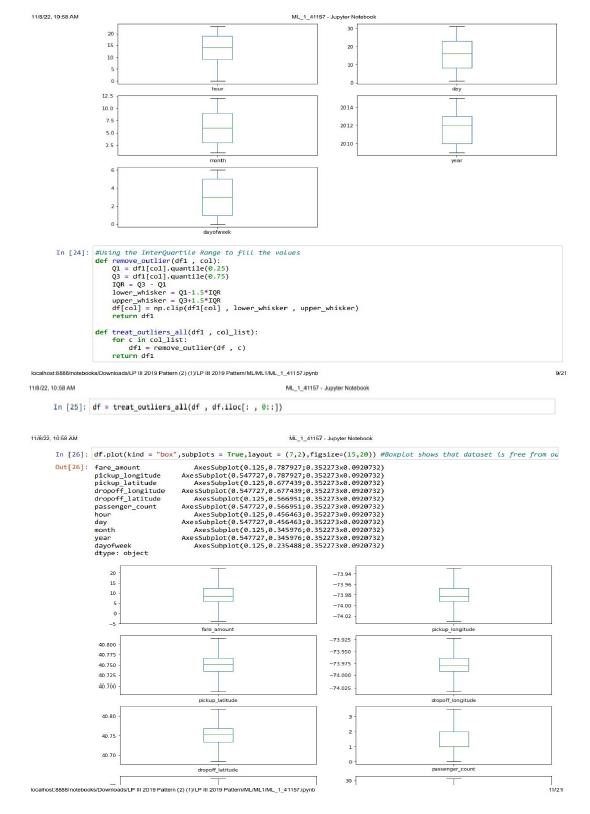
Program 6 Input Output

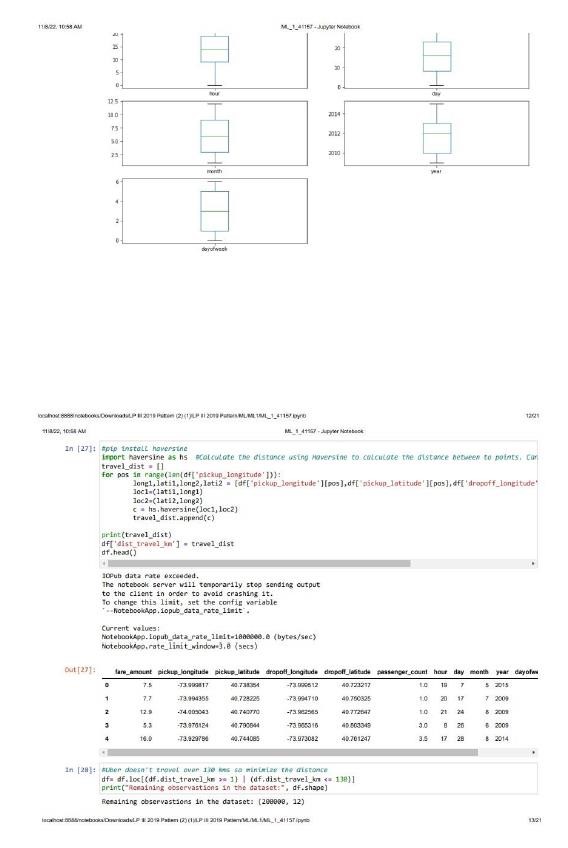


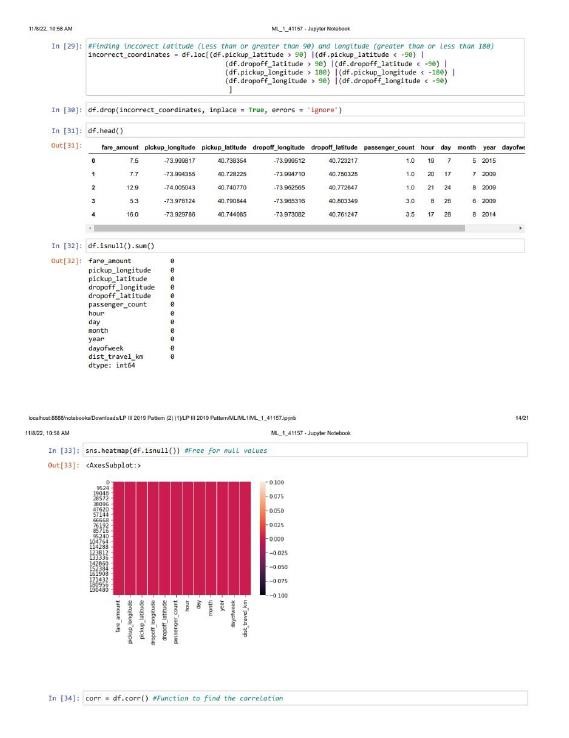


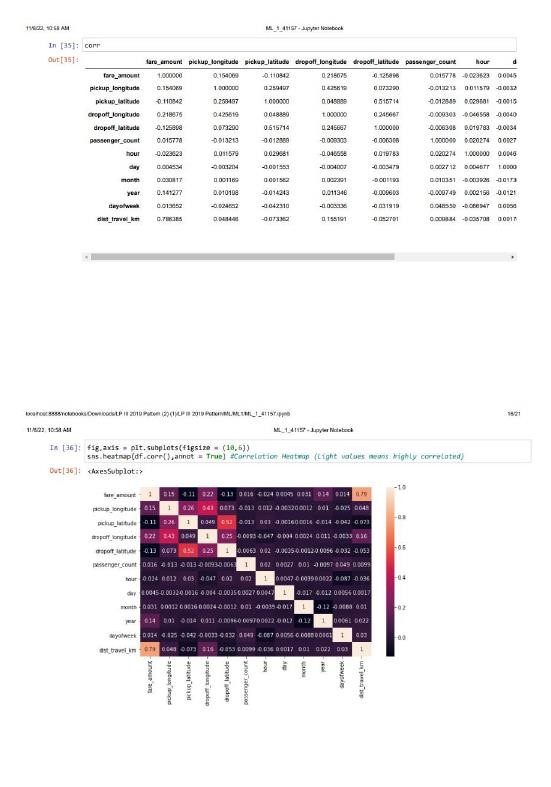


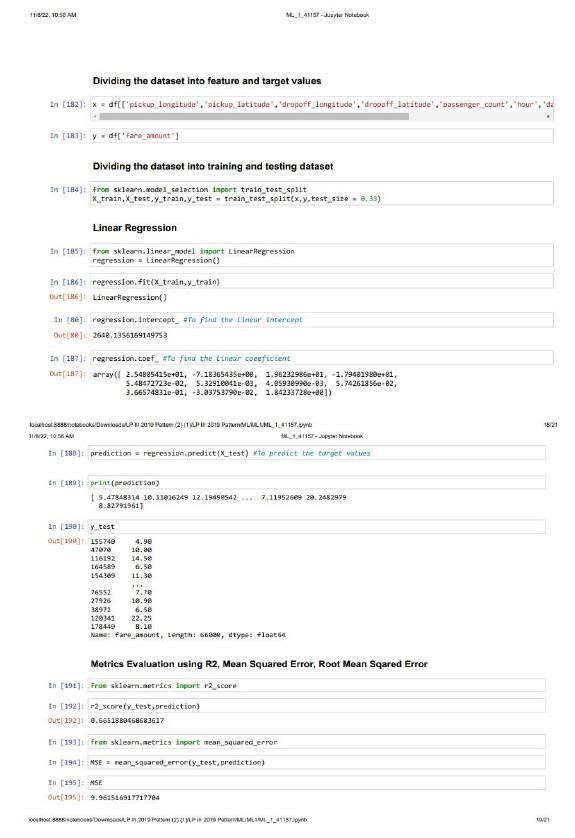


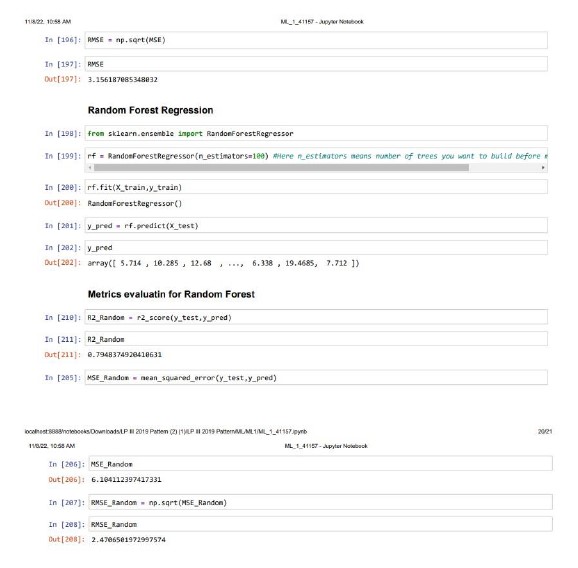




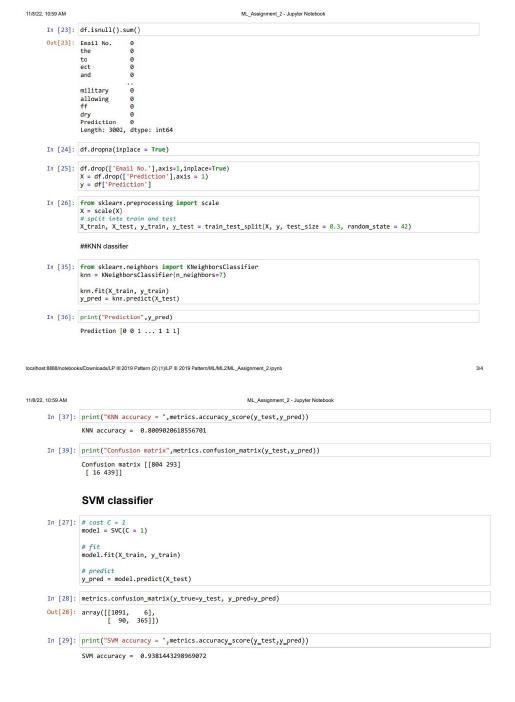




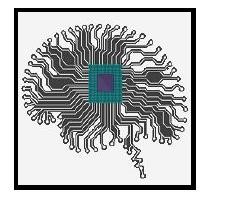
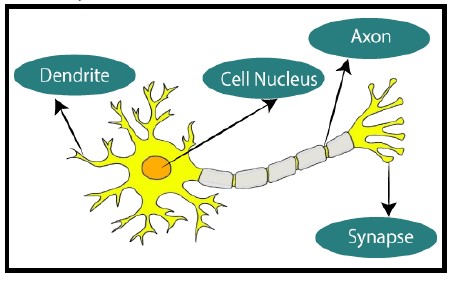
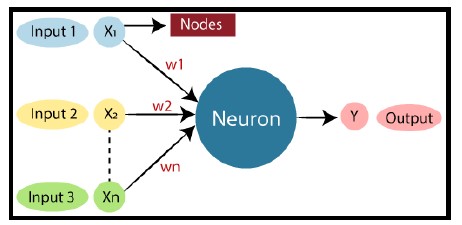
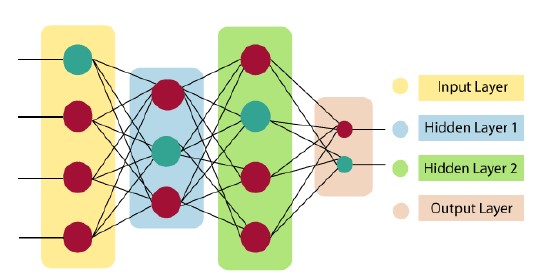
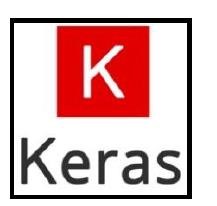
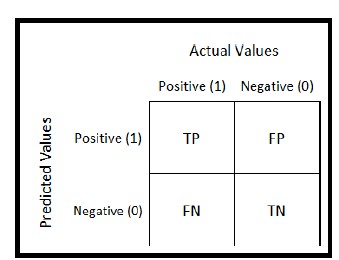




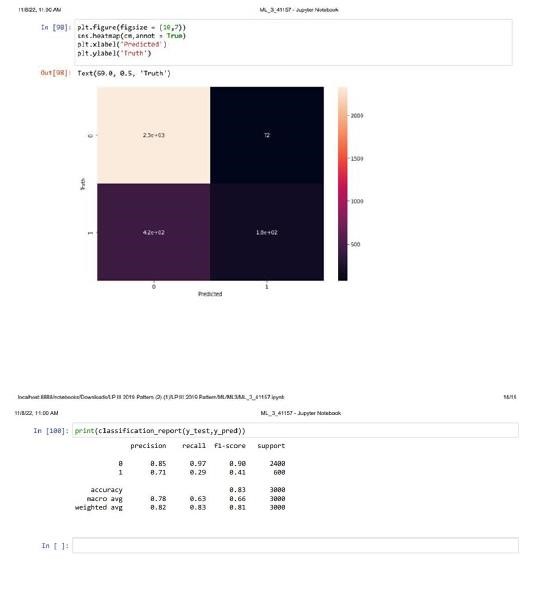
Program 7

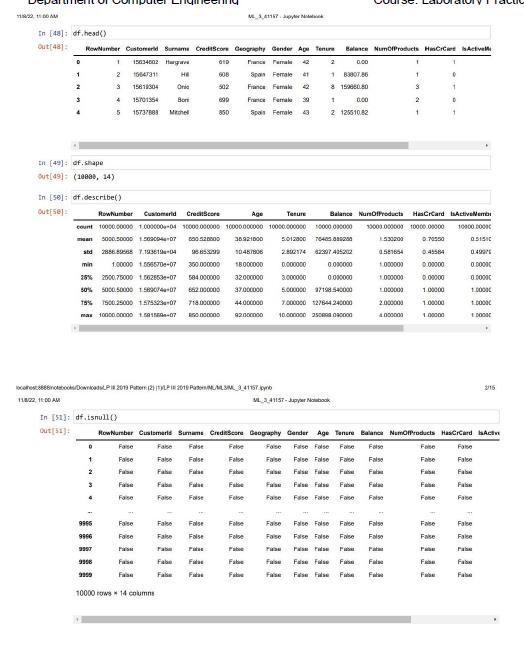


Program 8

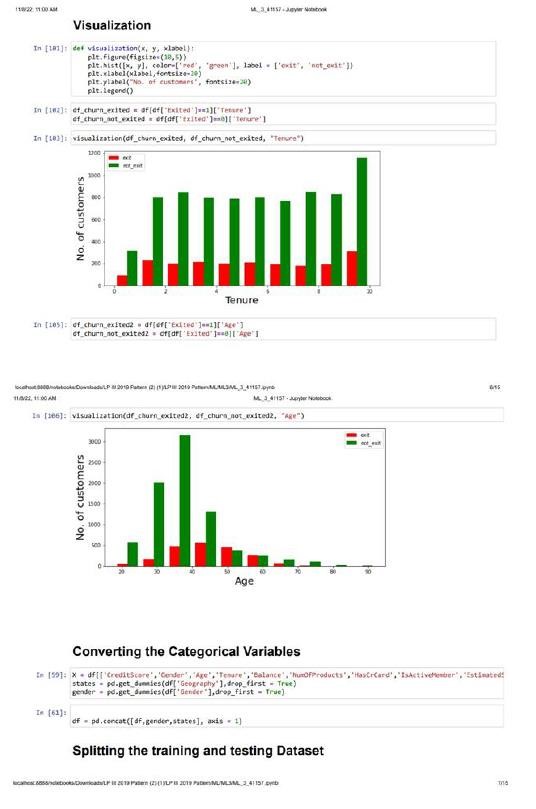


Program 8 Input Output

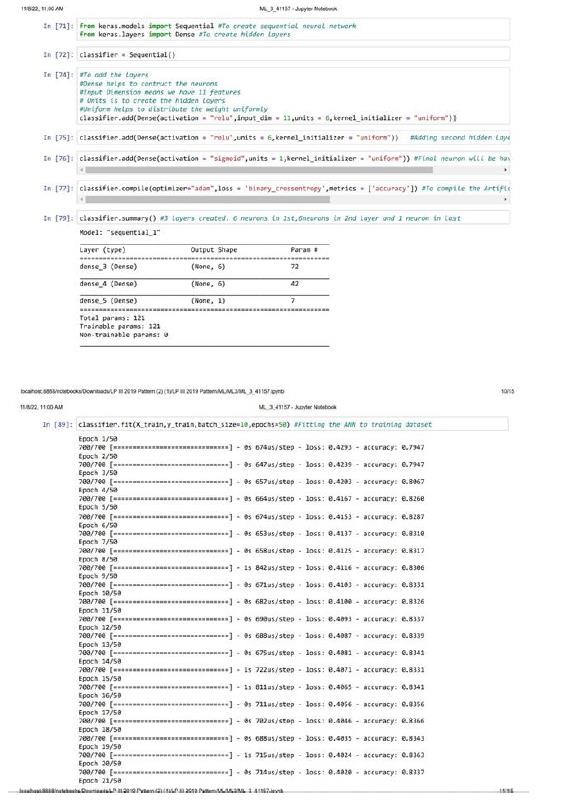


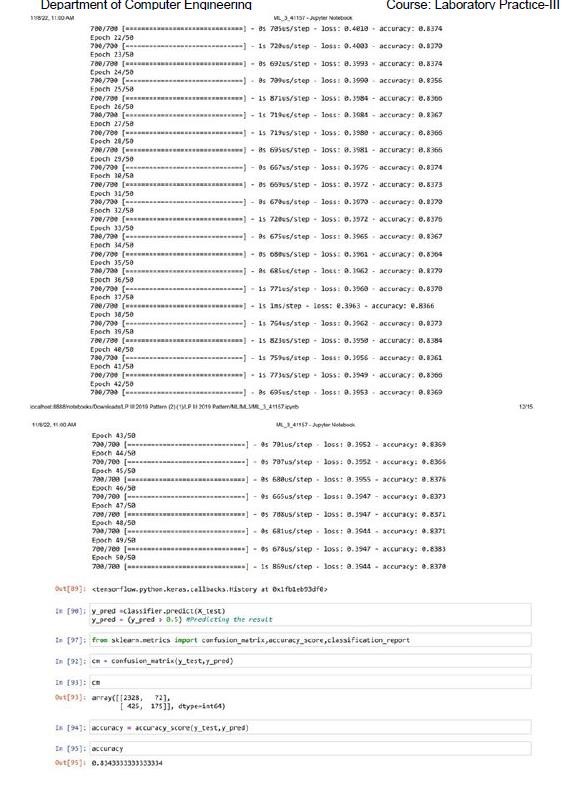


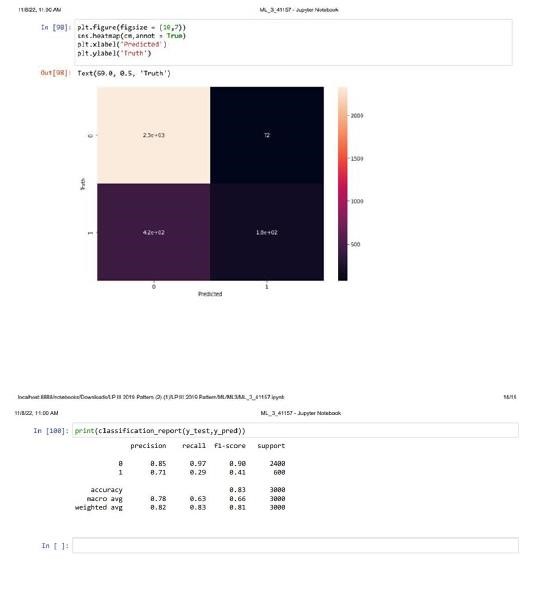








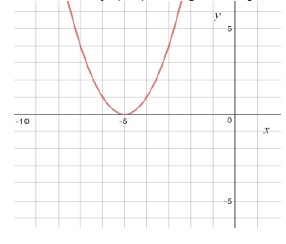
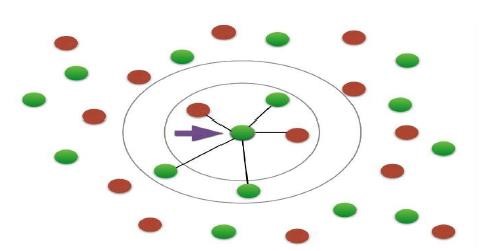
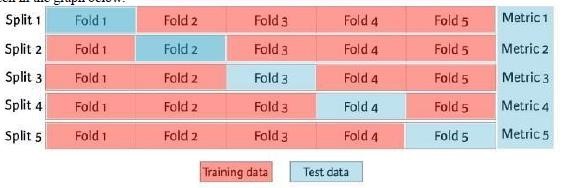




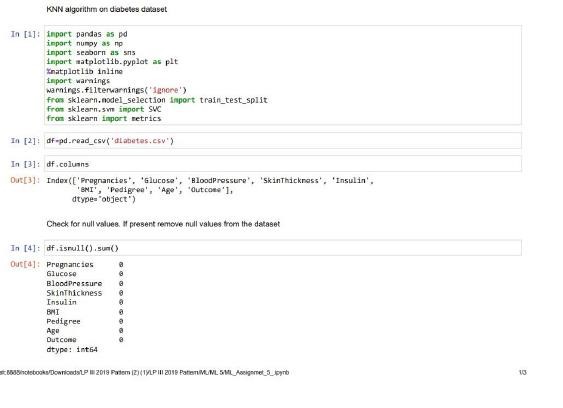
Program 9

program

10



Program 9 input output





Program 10 Input

import numpy as np

# Function to compute the cost (Mean Squared Error) def compute\_cost(X, y, theta):

m = len(y) # Number of training examples predictions = X.dot(theta) # Predicted values cost = (1 / (2 \* m)) \* np.sum(np.square(predictions - y)) # Mean Squared Error

return cost

# Gradient Descent Function def gradient\_descent(X, y, theta, learning\_rate, iterations):

m = len(y) # Number of training examples cost\_history = np.zeros(iterations) # To record cost at each step

for i in range(iterations):

predictions = X.dot(theta) # Calculate hypothesis/predictions errors = predictions - y # Errors in predictions

# Update theta values (gradient step) theta = theta - (learning\_rate / m) \* X.T.dot(errors)

# Record the cost after each iteration cost\_history[i] = compute\_cost(X, y, theta)

return theta, cost\_history

# Example usage

# Data for a simple linear regression (X and y values)

X = np.array([[1, 1], [1, 2], [1, 3], [1, 4], [1, 5]]) # Adding bias term (x0 = 1) y = np.array([1, 2, 3, 3.5, 5])

# Initialize theta (parameters) to 0 theta = np.zeros(2)

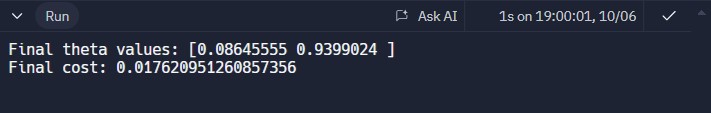
# Hyperparameters

learning\_rate = 0.01 iterations = 1000

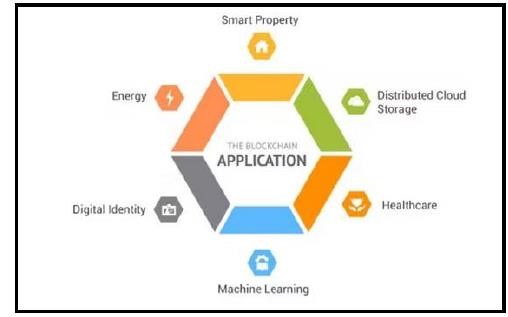
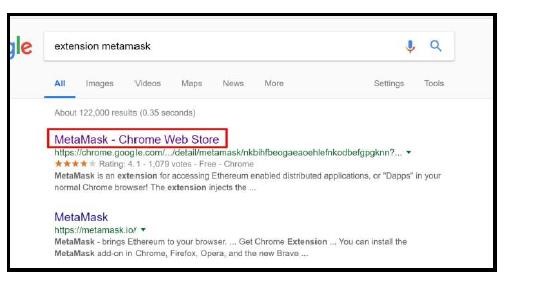
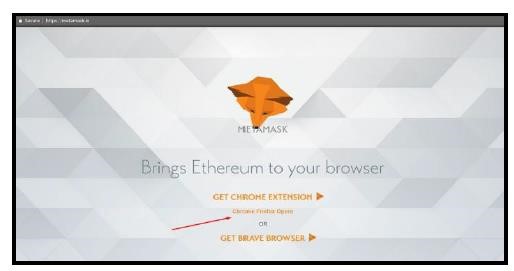
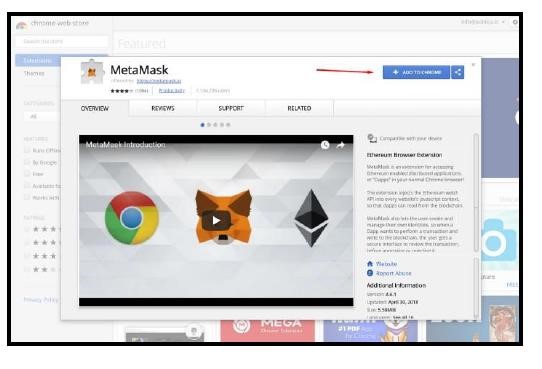
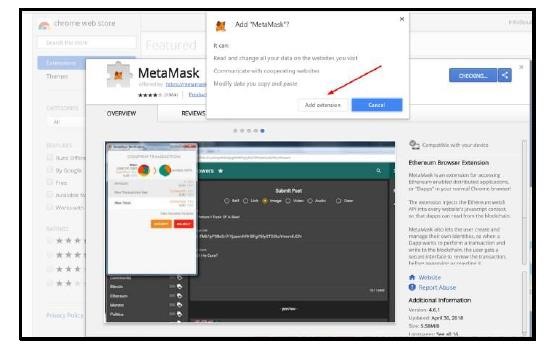
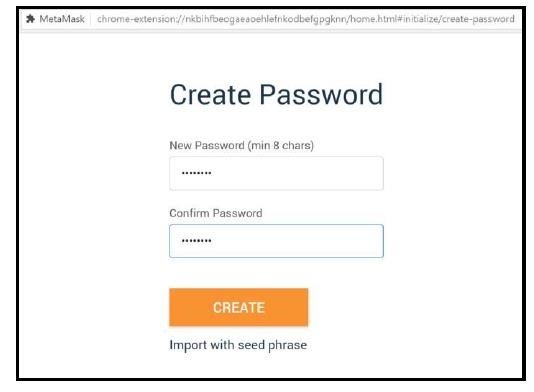
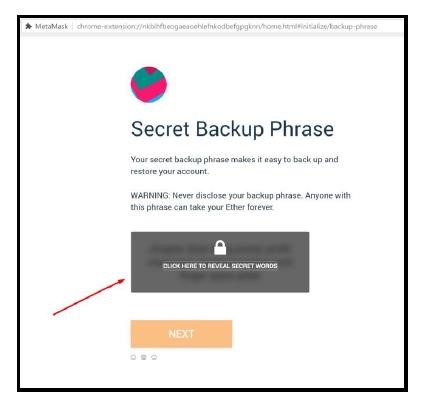
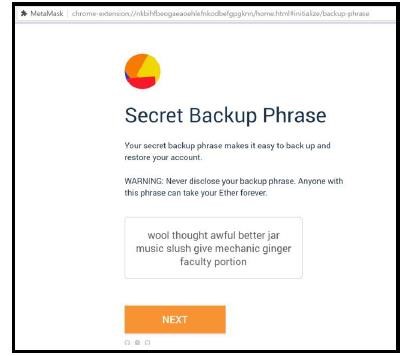
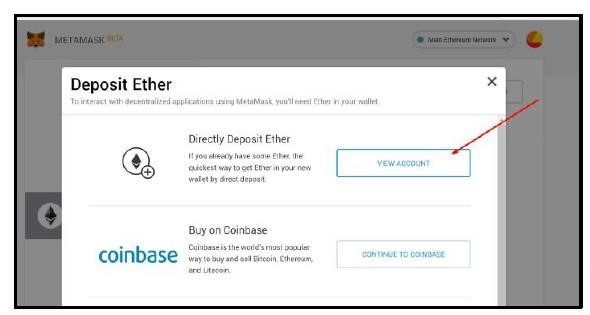
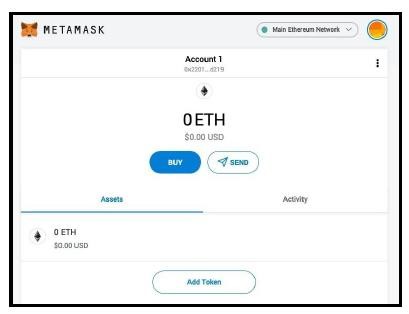
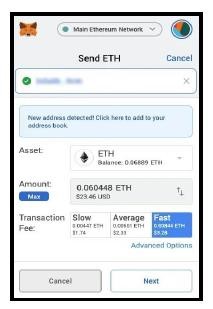
# Perform gradient descent theta, cost\_history = gradient\_descent(X, y, theta, learning\_rate, iterations)

print(f"Final theta values: {theta}") print(f"Final cost: {cost\_history[-1]}")

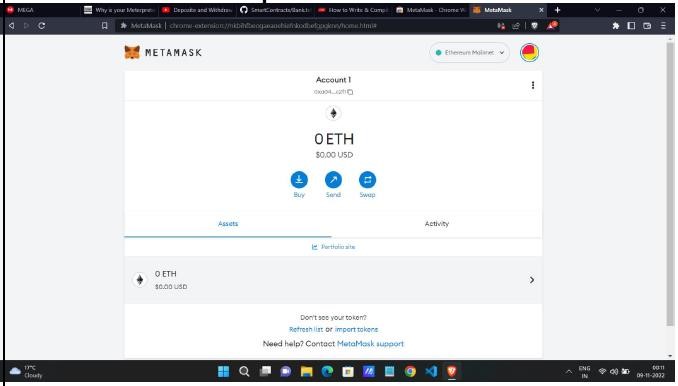
Program 10 Output



Program 10



Program 11



Program 13

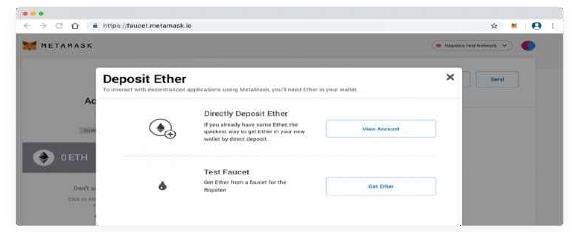
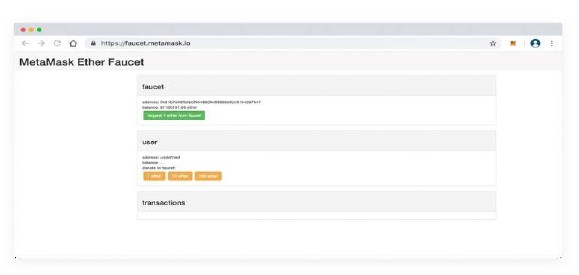
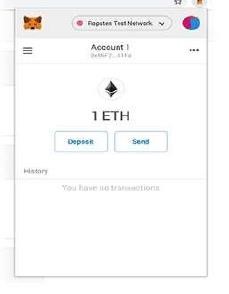
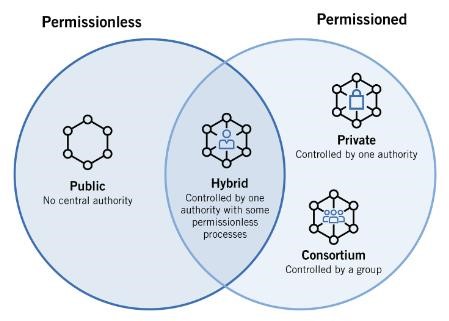
P

rogra

m

1

5



Program 15 Input

// SPDX-License-Identifier: MIT pragma solidity ^0.8.0; contract StudentRecords {

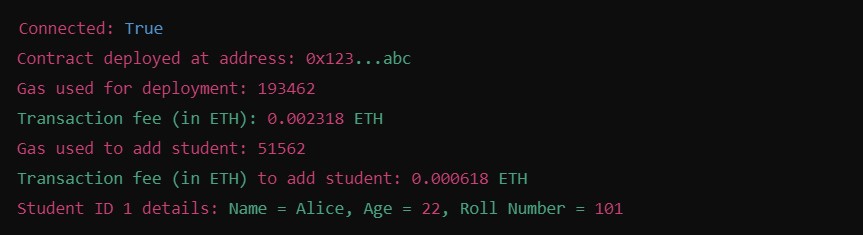
// Structure to store student information struct Student { string name; uint age; uint rollNumber; }

// Mapping to store students by their student ID mapping(uint => Student) public students; uint public studentCount;

// Event to log when a student is added event StudentAdded(uint studentId, string name, uint age, uint rollNumber);

// Add a new student function addStudent(string memory \_name, uint \_age, uint \_rollNumber) public { studentCount += 1; students[studentCount] = Student(\_name, \_age, \_rollNumber); emit StudentAdded(studentCount, \_name, \_age, \_rollNumber); }

// Retrieve student details by their student ID function getStudent(uint studentId) public view returns (string memory, uint, uint) { Student memory student = students[studentId]; return (student.name, student.age, student.rollNumber); }} Program 15 Output



Program 4

def knapsack(weights, values, capacity):

n = len(weights)

# DP table, where dp[i][w] is the max value that can be obtained with i items and a capacity of w

dp = [[0 for w in range(capacity + 1)] for i in range(n + 1)]

# Build the DP table

for i in range(1, n + 1):

for w in range(1, capacity + 1):

if weights[i-1] <= w:

# Item can be included, take the maximum of including or excluding the item

dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])

else:

# Item cannot be included, so we exclude it

dp[i][w] = dp[i-1][w]

# dp[n][capacity] will have the maximum value that can be achieved

return dp[n][capacity]

# Example usage:

weights = [1, 2, 3, 4]

values = [1, 4, 5, 7]

capacity = 7

max\_value = knapsack(weights, values, capacity)

print(f"The maximum value that can be obtained in the knapsack is: {max\_value}")

Output

