1. 2 Queens problem

CODE:

*def* is\_safe(*board*, *row*, *col*):

    for i in range(*row*):

        if *board*[i][*col*] == 'Q' or *board*[i][*row* - i] == 'Q':

            return False

    return True

*def* solve\_queens(*n*):

    if *n* != 2:

        print("The 2x2 Queens Problem is not solvable.")

        return

    board = [['.' for \_ in range(*n*)] for \_ in range(*n*)]

    if solve(board, 0, *n*):

        for row in board:

            print(' '.join(row))

    else:

        print("No solution exists.")

*def* solve(*board*, *row*, *n*):

    if *row* == *n*:

        return True

    for col in range(*n*):

        if is\_safe(*board*, *row*, col):

*board*[*row*][col] = 'Q'

            if solve(*board*, *row* + 1, *n*):

                return True

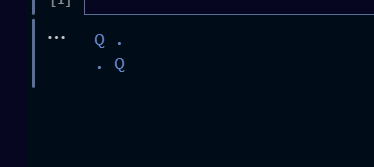
*board*[*row*][col] = '.'

    return False

if \_\_name\_\_ == "\_\_main\_\_":

    solve\_queens(2)

OUTPUT



1. 8 Queens problem

Code

*def* is\_safe(*board*, *row*, *col*):

    # Check the left side of the current row

    for i in range(*col*):

        if *board*[*row*][i] == 1:

            return False

    # Check upper diagonal on the left side

    for i, j in zip(range(*row*, -1, -1), range(*col*, -1, -1)):

        if *board*[i][j] == 1:

            return False

    # Check lower diagonal on the left side

    for i, j in zip(range(*row*, len(*board*), 1), range(*col*, -1, -1)):

        if *board*[i][j] == 1:

            return False

    return True

*def* solve\_queens(*board*, *col*):

    if *col* >= len(*board*):

        return True

    for i in range(len(*board*)):

        if is\_safe(*board*, i, *col*):

*board*[i][*col*] = 1

            if solve\_queens(*board*, *col* + 1):

                return True

*board*[i][*col*] = 0

    return False

*def* print\_solution(*board*):

    for row in *board*:

        print(" ".join(["Q" if x else "." for x in row]))

*def* solve\_8\_queens():

    board = [[0] \* 8 for \_ in range(8)]

    if solve\_queens(board, 0):

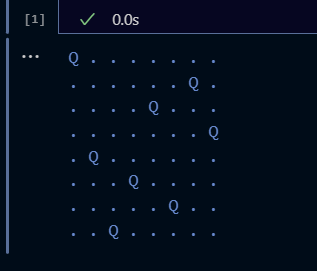
        print\_solution(board)

    else:

        print("No solution exists.")

if \_\_name\_\_ == "\_\_main\_\_":

    solve\_8\_queens()

output:  


3)Python Program for Water Jug Problem

Code

from collections import deque

def water\_jug\_problem(capacity\_x, capacity\_y, target):

    visited = set()

    initial\_state = (0, 0)  # Initial state with both jugs empty

    queue = deque([initial\_state])

    while queue:

        current\_state = queue.popleft()

        x, y = current\_state

        if x == target or y == target:

            return current\_state

        # Define possible actions: Fill jug x, fill jug y, empty jug x, empty jug y, pour from x to y, pour from y to x

        actions = [

            (capacity\_x, y),

            (x, capacity\_y),

            (0, y),

            (x, 0),

            (min(x + y, capacity\_x), max(0, x + y - capacity\_x)),

            (max(0, x + y - capacity\_y), min(x + y, capacity\_y))

        ]

        for action in actions:

            if action not in visited:

                queue.append(action)

                visited.add(action)

    return None  # If no solution is found

def print\_solution(solution, capacity\_x, capacity\_y):

    if solution:

        x, y = solution

        print("Solution:")

        print(f"Jug X: {x}/{capacity\_x}")

        print(f"Jug Y: {y}/{capacity\_y}")

    else:

        print("No solution exists.")

if \_\_name\_\_ == "\_\_main\_\_":

    capacity\_x = 4  # Capacity of jug X

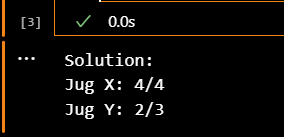
    capacity\_y = 3  # Capacity of jug Y

    target = 2  # Desired amount of water

    solution = water\_jug\_problem(capacity\_x, capacity\_y, target)

    print\_solution(solution, capacity\_x, capacity\_y)

OUTPUT:



4)Python Program for Crypt-Arithmatic Problem

Code:

import itertools

def is\_valid\_solution(words, mapping):

    values = []

    for word in words:

        value = 0

        for letter in word:

            value = value \* 10 + mapping[letter]

        values.append(value)

    return all(values[0] + values[1] == values[2] for values in itertools.permutations(values))

def solve\_cryptarithmetic(words):

    # Extract all unique letters from the words

    unique\_letters = set("".join(words))

    # Check if the words have the same length, if not, no solution is possible

    if len(unique\_letters) > 10 or len(words[0]) < len(words[-1]):

        return None

    for perm in itertools.permutations("0123456789", len(unique\_letters)):

        mapping = dict(zip(unique\_letters, perm))

        if is\_valid\_solution(words, mapping):

            return mapping

    return None

if \_\_name\_\_ == "\_\_main\_\_":

    word1 = "SEND"

    word2 = "MORE"

    result\_word = "MONEY"

    words = [word1, word2, result\_word]

    solution = solve\_cryptarithmetic(words)

    if solution:

        print("Solution found:")

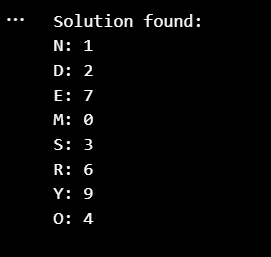
        for word in words:

            print("".join(str(solution[letter]) for letter in word))

    else:

        print("No solution exists.")

OUTPUT



5) Write the python program for Missionaries Cannibal problem

Code:

from collections import deque

# Define the initial and goal states

initial\_state = (3, 3, 1)  # (Missionaries on the left, Cannibals on the left, Boat on the left)

goal\_state = (0, 0, 0)

# Define valid actions

valid\_actions = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

def is\_valid(state):

    m, c, b = state

    if m < 0 or c < 0 or m > 3 or c > 3 or (m != 0 and m < c) or (m != 3 and 3 - m < 3 - c):

        return False

    return True

def get\_neighbors(state):

    neighbors = []

    for action in valid\_actions:

        if state[2] == 1:  # Boat is on the left

            new\_state = tuple(state[i] - action[i] if i < 2 else 1 - state[i] for i in range(3))

        else:  # Boat is on the right

            new\_state = tuple(state[i] + action[i] if i < 2 else 1 - state[i] for i in range(3))

        if is\_valid(new\_state) and new\_state != state:

            neighbors.append(new\_state)

    return neighbors

def solve\_missionaries\_and\_cannibals():

    visited = set()

    queue = deque([(initial\_state, [])])

    while queue:

        state, path = queue.popleft()

        if state == goal\_state:

            return path

        for neighbor in get\_neighbors(state):

            if neighbor not in visited:

                visited.add(neighbor)

                new\_path = path + [state, neighbor]

                queue.append((neighbor, new\_path))

    return None

def print\_solution(solution):

    if solution:

        print("Solution:")

        for state in solution:

            print(f"Missionaries: {state[0]}, Cannibals: {state[1]}, Boat: {'Left' if state[2] == 1 else 'Right'}")

    else:

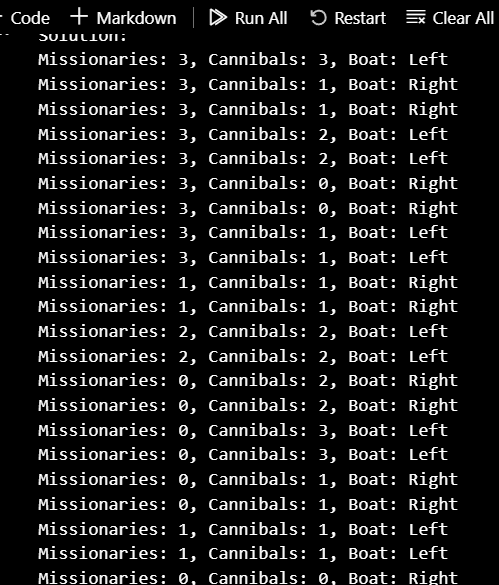
        print("No solution exists.")

if \_\_name\_\_ == "\_\_main\_\_":

    solution = solve\_missionaries\_and\_cannibals()

    print\_solution(solution)

OUTPUT



6) Write the python program for Vacuum Cleaner problem

class VacuumCleaner:

    def \_\_init\_\_(self, grid):

        self.grid = grid

        self.rows = len(grid)

        self.cols = len(grid[0])

        self.row = 0

        self.col = 0

    def clean(self):

        while True:

            self.clean\_current\_cell()

            if not self.move\_to\_next\_cell():

                break

    def clean\_current\_cell(self):

        if self.grid[self.row][self.col] == 1:

            print(f"Cleaning cell at row {self.row}, col {self.col}")

            self.grid[self.row][self.col] = 0

    def move\_to\_next\_cell(self):

        if self.row < self.rows - 1:

            self.row += 1

        elif self.col < self.cols - 1:

            self.row = 0

            self.col += 1

        else:

            return False

        return True

if \_\_name\_\_ == "\_\_main\_\_":

    # Define a grid where 1 represents a dirty cell

    grid = [[1, 0, 1],

            [0, 1, 0],

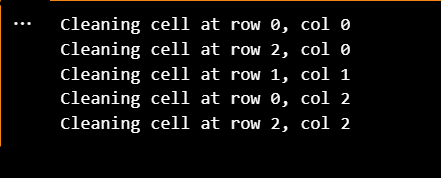
            [1, 0, 1]]

    # Create a vacuum cleaner agent and start cleaning

    vacuum = VacuumCleaner(grid)

    vacuum.clean()

OUTPUT:



7) Write the python program to implement BFS.

CODE:

from collections import defaultdict, deque

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def add\_edge(self, vertex, neighbor):

        self.graph[vertex].append(neighbor)

    def bfs(self, start\_vertex):

        visited = set()

        queue = deque()

        visited.add(start\_vertex)

        queue.append(start\_vertex)

        while queue:

            current\_vertex = queue.popleft()

            print(current\_vertex, end=" ")

            for neighbor in self.graph[current\_vertex]:

                if neighbor not in visited:

                    visited.add(neighbor)

                    queue.append(neighbor)

if \_\_name\_\_ == "\_\_main\_\_":

    # Create a graph

    g = Graph()

    g.add\_edge(0, 1)

    g.add\_edge(0, 2)

    g.add\_edge(1, 2)

    g.add\_edge(2, 0)

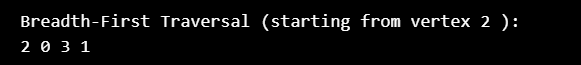
    g.add\_edge(2, 3)

    g.add\_edge(3, 3)

    start\_vertex = 2  # Starting vertex for BFS

    print("Breadth-First Traversal (starting from vertex", start\_vertex, "):")

    g.bfs(start\_vertex)

OUTPUT:  


8) Write the python program to implement DFS

CODE:

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def add\_edge(self, vertex, neighbor):

        self.graph[vertex].append(neighbor)

    def dfs(self, start\_vertex, visited):

        visited.add(start\_vertex)

        print(start\_vertex, end=" ")

        for neighbor in self.graph[start\_vertex]:

            if neighbor not in visited:

                self.dfs(neighbor, visited)

if \_\_name\_\_ == "\_\_main\_\_":

    # Create a graph

    g = Graph()

    g.add\_edge(0, 1)

    g.add\_edge(0, 2)

    g.add\_edge(1, 2)

    g.add\_edge(2, 0)

    g.add\_edge(2, 3)

    g.add\_edge(3, 3)

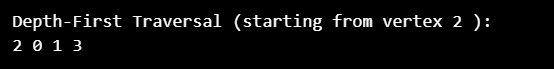
    start\_vertex = 2  # Starting vertex for DFS

    visited = set()

    print("Depth-First Traversal (starting from vertex", start\_vertex, "):")

    g.dfs(start\_vertex, visited)

OUTPUT:



9)Travelling salesman Problem to find the shortest distance using python

CODE:

import itertools

def calculate\_total\_distance(path, distances):

    total\_distance = 0

    for i in range(len(path) - 1):

        total\_distance += distances[path[i]][path[i+1]]

    total\_distance += distances[path[-1]][path[0]]  # Return to the starting city

    return total\_distance

def traveling\_salesman\_bruteforce(distances):

    num\_cities = len(distances)

    cities = list(range(num\_cities))

    shortest\_path = None

    shortest\_distance = float('inf')

    for path in itertools.permutations(cities):

        distance = calculate\_total\_distance(path, distances)

        if distance < shortest\_distance:

            shortest\_path = path

            shortest\_distance = distance

    return shortest\_path, shortest\_distance

if \_\_name\_\_ == "\_\_main\_\_":

    # Example: Distances between cities

    distances = [

        [0, 29, 20, 21],

        [29, 0, 15, 22],

        [20, 15, 0, 24],

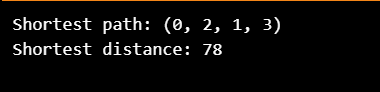
        [21, 22, 24, 0]

    ]

    shortest\_path, shortest\_distance = traveling\_salesman\_bruteforce(distances)

    print("Shortest path:", shortest\_path)

    print("Shortest distance:", shortest\_distance)

OUTPUT:  


10) Write the python program to implement A\* algorithm

import heapq

class Node:

    def \_\_init\_\_(self, x, y, parent=None):

        self.x = x

        self.y = y

        self.parent = parent

        self.g = 0

        self.h = 0

    def \_\_lt\_\_(self, other):

        return (self.g + self.h) < (other.g + other.h)

def heuristic(node, goal):

    return abs(node.x - goal.x) + abs(node.y - goal.y)

def astar(grid, start, goal):

    open\_set = []

    closed\_set = set()

    start\_node = Node(start[0], start[1])

    goal\_node = Node(goal[0], goal[1])

    heapq.heappush(open\_set, start\_node)

    while open\_set:

        current\_node = heapq.heappop(open\_set)

        if current\_node.x == goal\_node.x and current\_node.y == goal\_node.y:

            path = []

            while current\_node:

                path.append((current\_node.x, current\_node.y))

                current\_node = current\_node.parent

            return path[::-1]

        closed\_set.add((current\_node.x, current\_node.y))

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

            new\_x, new\_y = current\_node.x + dx, current\_node.y + dy

            if 0 <= new\_x < len(grid) and 0 <= new\_y < len(grid[0]) and grid[new\_x][new\_y] != 1:

                if (new\_x, new\_y) not in closed\_set:

                    child\_node = Node(new\_x, new\_y, parent=current\_node)

                    child\_node.g = current\_node.g + 1

                    child\_node.h = heuristic(child\_node, goal\_node)

                    heapq.heappush(open\_set, child\_node)

    return None

if \_\_name\_\_ == "\_\_main\_\_":

    # Example grid (0 represents empty, 1 represents an obstacle)

    grid = [

        [0, 0, 0, 0, 0],

        [0, 1, 1, 0, 0],

        [0, 1, 0, 0, 0],

        [0, 1, 0, 1, 0],

        [0, 0, 0, 0, 0]

    ]

    start = (0, 0)

    goal = (4, 4)

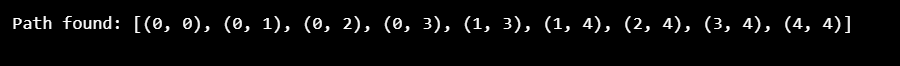
    path = astar(grid, start, goal)

    if path:

        print("Path found:", path)

    else:

        print("No path found.")

OUTPUT:  


11) Write the python program for Map Coloring to implement CSP.

CODE:

def is\_valid\_assignment(graph, assignment, node, color):

    for neighbor in graph[node]:

        if neighbor in assignment and assignment[neighbor] == color:

            return False

    return True

def backtracking(graph, colors, assignment, node):

    if node not in assignment:

        for color in colors:

            if is\_valid\_assignment(graph, assignment, node, color):

                assignment[node] = color

                if len(assignment) == len(graph):

                    return True

                next\_node = get\_unassigned\_node(graph, assignment)

                if backtracking(graph, colors, assignment, next\_node):

                    return True

                assignment.pop(node)

    return False

def get\_unassigned\_node(graph, assignment):

    for node in graph:

        if node not in assignment:

            return node

def map\_coloring(graph, colors):

    assignment = {}

    start\_node = get\_unassigned\_node(graph, assignment)

    if backtracking(graph, colors, assignment, start\_node):

        return assignment

    else:

        return None

if \_\_name\_\_ == "\_\_main\_\_":

    # Define the graph (adjacency list)

    graph = {

        "WA": ["NT", "SA"],

        "NT": ["WA", "SA", "Q"],

        "SA": ["WA", "NT", "Q", "NSW", "V"],

        "Q": ["NT", "SA", "NSW"],

        "NSW": ["Q", "SA", "V"],

        "V": ["SA", "NSW"]

    }

    # Define the available colors

    colors = ["Red", "Green", "Blue"]

    # Solve the map coloring problem

    solution = map\_coloring(graph, colors)

    if solution:

        print("Map coloring solution:")

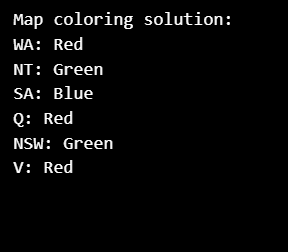
        for node, color in solution.items():

            print(f"{node}: {color}")

    else:

        print("No solution exists.")

OUTPUT:



12) Write the python program for Tic Tac Toe game

CODE:

def print\_board(board):

    for row in board:

        print(" | ".join(row))

        print("-" \* 9)

def check\_winner(board, player):

    for row in board:

        if all(cell == player for cell in row):

            return True

    for col in range(3):

        if all(row[col] == player for row in board):

            return True

    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):

        return True

    return False

def is\_board\_full(board):

    return all(cell != " " for row in board for cell in row)

def play\_tic\_tac\_toe():

    board = [[" " for \_ in range(3)] for \_ in range(3)]

    current\_player = "X"

    while True:

        print\_board(board)

        try:

            while True:

                row, col = map(int, input(f"Player {current\_player}, enter row (0-2) and column (0-2) separated by space: ").split())

                if row not in [0, 1, 2] or col not in [0, 1, 2]:

                    print("Invalid input. Please enter numbers between 0 and 2.")

                    continue

                if board[row][col] == " ":

                    break

                else:

                    print("That cell is already taken. Try again.")

        except ValueError:

            print("Invalid input. Please enter numbers.")

            continue

        except Exception as e:

            print(f"An error occurred: {str(e)}")

            continue

        board[row][col] = current\_player

        if check\_winner(board, current\_player):

            print\_board(board)

            print(f"Player {current\_player} wins!")

            break

        if is\_board\_full(board):

            print\_board(board)

            print("It's a tie!")

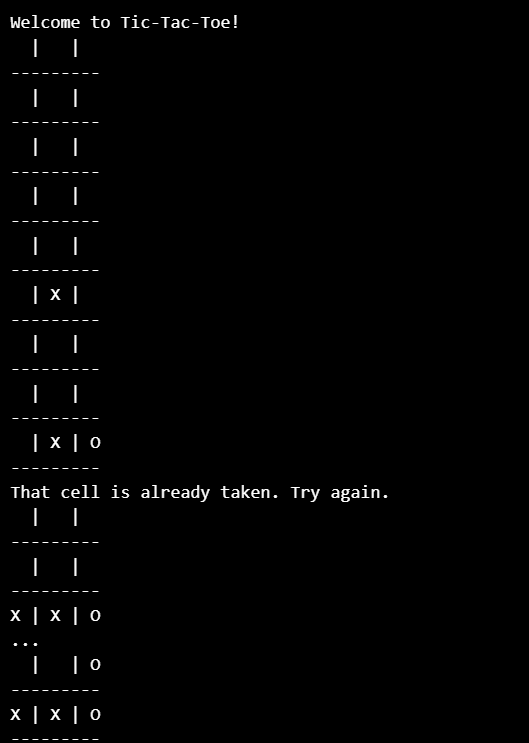
            break

        current\_player = "O" if current\_player == "X" else "X"

if \_\_name\_\_ == "\_\_main\_\_":

    print("Welcome to Tic-Tac-Toe!")

    play\_tic\_tac\_toe()

OUTPUT:  


13) Write the python program to implement Minimax algorithm for gaming

CODE:  
def is\_game\_over(state):

    # Check if the game is over and return the winner or tie

    for row in state:

        if all(cell == 'X' for cell in row):

            return 'X'

        if all(cell == 'O' for cell in row):

            return 'O'

    for col in range(3):

        if all(state[row][col] == 'X' for row in range(3)):

            return 'X'

        if all(state[row][col] == 'O' for row in range(3)):

            return 'O'

    if all(state[i][i] == 'X' for i in range(3)) or all(state[i][2 - i] == 'X' for i in range(3)):

        return 'X'

    if all(state[i][i] == 'O' for i in range(3)) or all(state[i][2 - i] == 'O' for i in range(3)):

        return 'O'

    if all(cell != ' ' for row in state for cell in row):

        return None  # It's a tie

    return False  # Game is not over

def get\_valid\_moves(state):

    # Return a list of valid moves (empty cells) from the given state

    moves = []

    for row in range(3):

        for col in range(3):

            if state[row][col] == ' ':

                moves.append((row, col))

    return moves

def make\_move(state, move, player):

    # Return a new state after applying the move to the given state

    new\_state = [row.copy() for row in state]

    new\_state[move[0]][move[1]] = player

    return new\_state

def minimax(state, player):

    winner = is\_game\_over(state)

    if winner is not False:

        if winner == 'X':

            return 1

        elif winner == 'O':

            return -1

        else:  # Tie

            return 0

    moves = get\_valid\_moves(state)

    if player == 'X':

        best\_score = float('-inf')

        for move in moves:

            new\_state = make\_move(state, move, player)

            score = minimax(new\_state, 'O')

            best\_score = max(score, best\_score)

        return best\_score

    else:  # player == 'O'

        best\_score = float('inf')

        for move in moves:

            new\_state = make\_move(state, move, player)

            score = minimax(new\_state, 'X')

            best\_score = min(score, best\_score)

        return best\_score

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    board = [['X', ' ', 'O'],

             ['O', 'X', ' '],

             ['X', 'O', ' ']]

    player = 'X'

    print("The Tic-Tac-Toe board state:")

    for row in board:

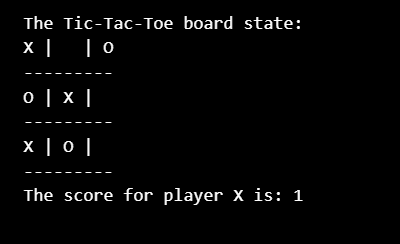
        print(" | ".join(row))

        print("-" \* 9)

    score = minimax(board, player)

    print(f"The score for player {player} is: {score}")

output



14)Write the python program to implement Apha & Beta pruning algorithm for gaming

# Python3 program to demonstrate

# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta

MAX, MIN = 1000, -1000

# Returns optimal value for current player

#(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

            values, alpha, beta):

    # Terminating condition. i.e

    # leaf node is reached

    if depth == 3:

        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex \* 2 + i,

                          False, values, alpha, beta)

            best = max(best, val)

            alpha = max(alpha, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best

    else:

        best = MAX

        # Recur for left and

        # right children

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex \* 2 + i,

                            True, values, alpha, beta)

            best = min(best, val)

            beta = min(beta, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best

# Driver Code

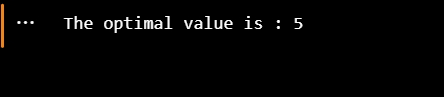
if \_\_name\_\_ == "\_\_main\_\_":

    values = [3, 5, 6, 9, 1, 2, 0, -1]

    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

# This code is contributed by Rituraj Jain

OUTPUT:



15)Write the python program to implement Decision Tree

CODE:

OUTPUT:

16) Write the python program to implement Feed forward neural Network

CODE:

OUTPUT: