|  |  |
| --- | --- |
| Sr.No. | Experiments name: |
| 1) | a. Implement Depth First Search (DFS) for graph traversal. b. Implement Breadth First Search (BFS) for graph traversal. |
| 2) | Solve 8-puzzle problem using A\* Search. i. using misplaced tiles as the heuristic function and ii. using Manhattan distance as the heuristic function. |
| 3) | Write a program to play Tic-Tac-Toe Game. |
| 4) | Implement Truth Table for the following equations:   1. P and Q 2. P or Q 3. (P and Q or R) or (P or Q and R)   (P or Q and R) and (P and Q or R) |
| 5) | Write a Program to Solve 8 Queens Problem. |
| 6) | Write a program to list the name of enemies of America the names of criminals using Forward Chaining from the following statements :   1. It is a crime for an American to sell weapons to the enemy of America. 2. Country Nono is a enemy of America. 3. Nono has some Missiles. 4. All the missiles were sold to Nono by Colonel. 5. Missile is weapon. 6. Colonel is an American |

**Experiment No. 01:**

**Program: Implement DFS for graph traversal.**

Code:

graph = {

    # graph diagram would be given

  'A' : ['B','C','D'],

  'B' : ['A','E'],

  'C' : ['A','E','F'],

  'D' : ['A','F'],

  'E' : ['B','C','H'],

  'F' : ['C','D','G'],

  'G' : ['F','H','K','I'],

  'H' : ['E','G'],

  'I' : ['G','J'],

  'J' : ['I','K'],

  'K' : ['G','J'],

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs

    if node not in visited:

        print (node)

        visited.add(node)

        for neighbour in graph[node]:

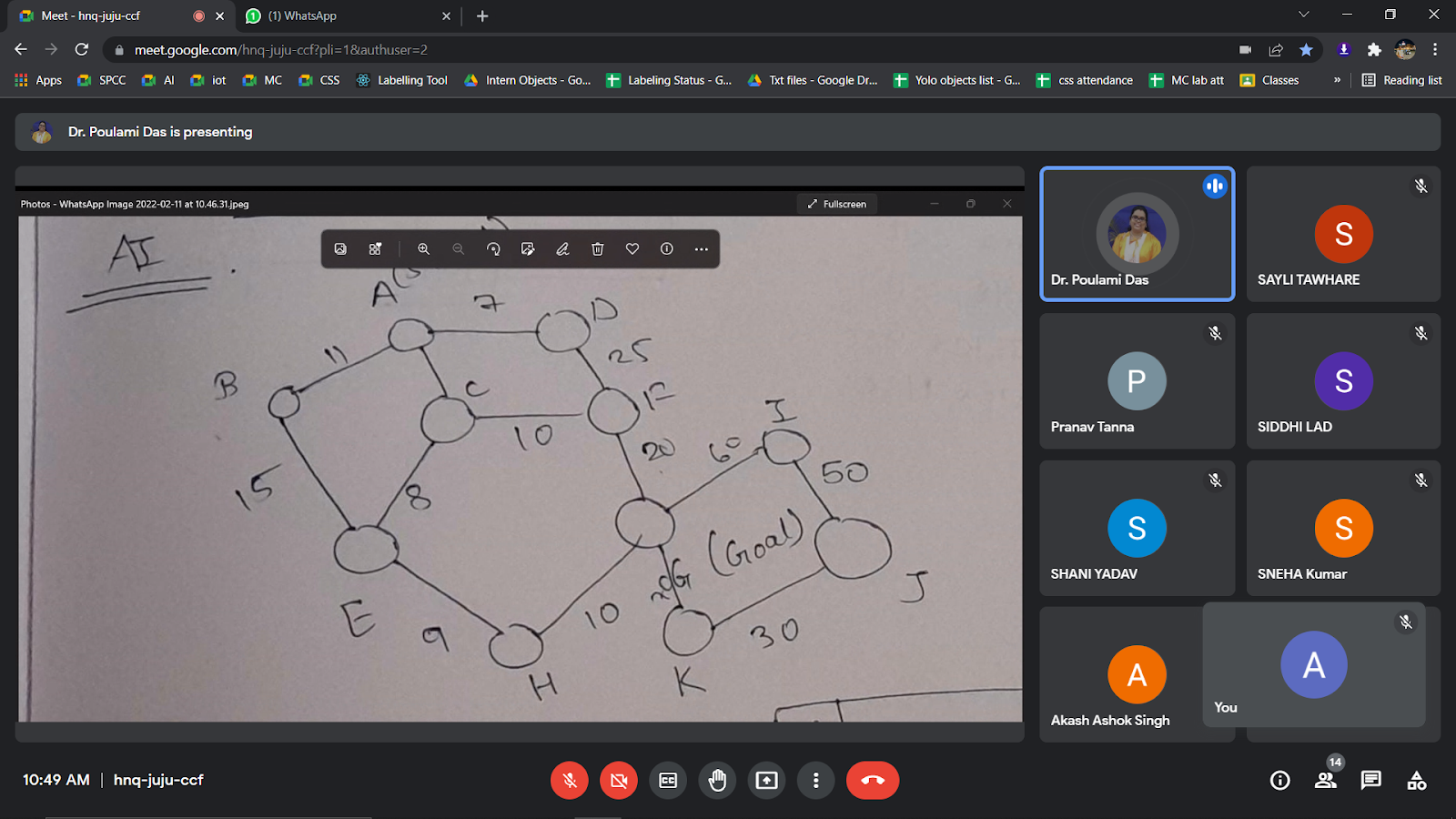
            dfs(visited, graph, neighbour)

# Driver Code

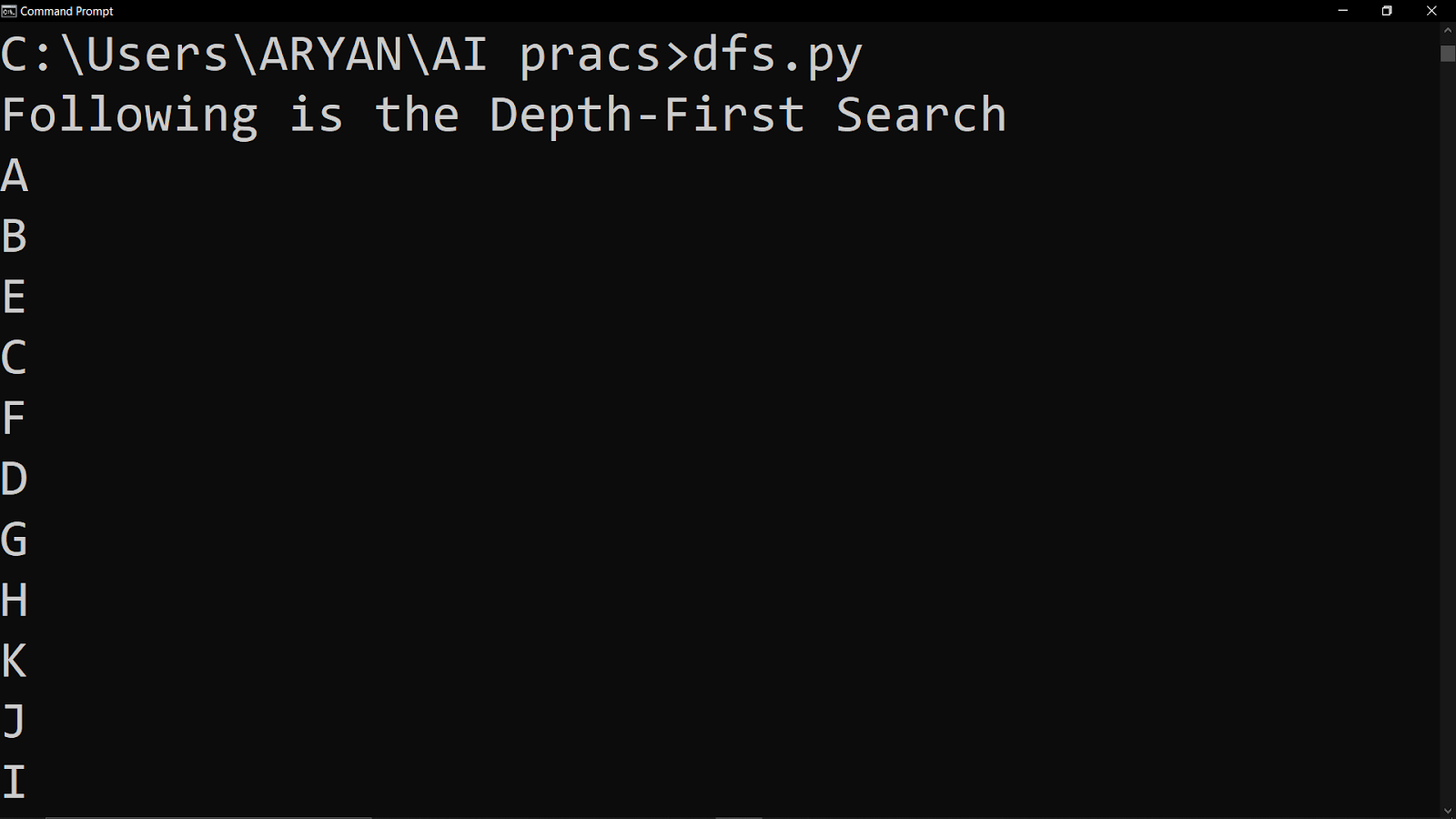
print("Following is the Depth-First Search")

dfs(visited, graph, 'A')

**Input Graph:**



**Output:**



**Program: Implement BFS for graph traversal.**

**Code:**

graph = {

  'S' : ['1','3'],

  '1' : ['G'],

  '3' : ['1','G','4'],

  '2' : ['1'],

  '4' : ['2','5'],

  '5' : ['2','G'],

  'G' : ['4']

}

visited = [] # List to keep track of visited nodes.

queue = []     #Initialize a queue

def bfs(visited, graph, node):

  visited.append(node)

  queue.append(node)

  while queue:

    s = queue.pop(0)

    print (s, end = " ")

    for neighbour in graph[s]:

      if neighbour not in visited:

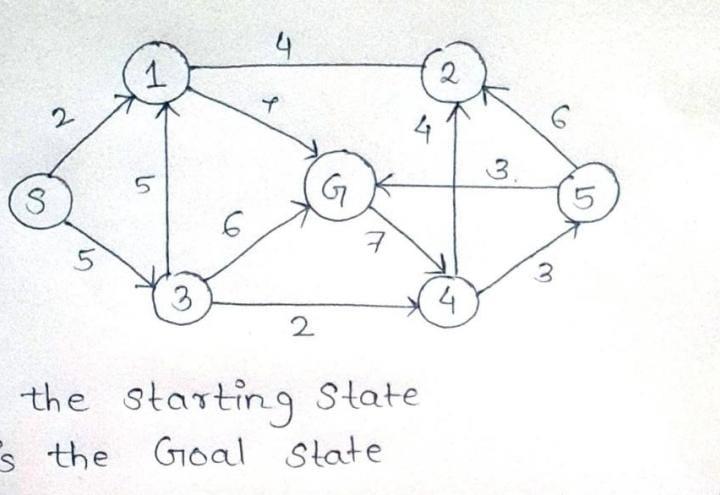
        visited.append(neighbour)

        queue.append(neighbour)

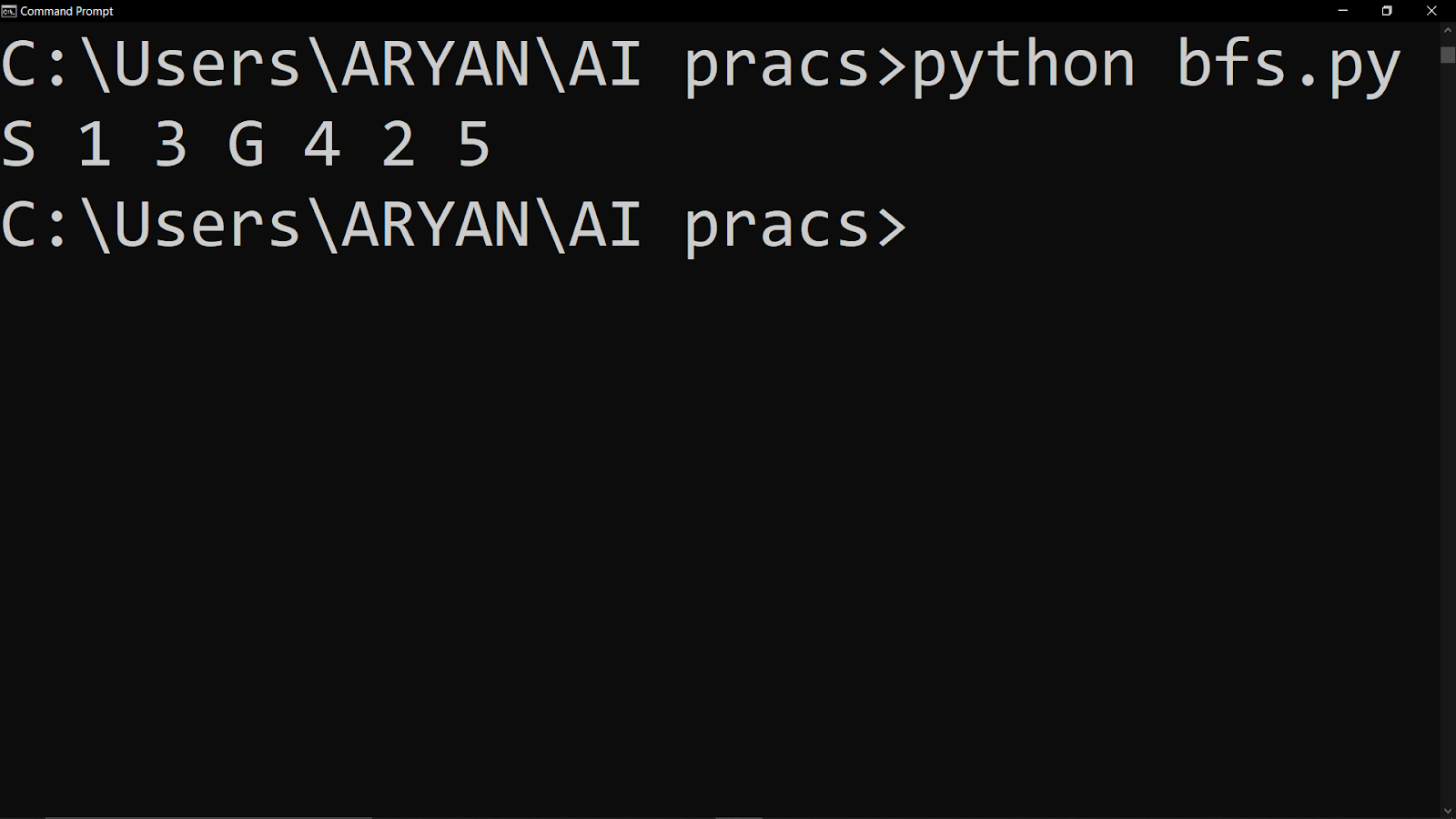
# Driver Code

bfs(visited, graph, 'S')

**Input Graph:**

****

**Output:**



**Experiment No: 02**

**Program:** Solve 8-puzzle problem using A\* Search. i. using misplaced tiles as the heuristic function and ii. using Manhattan distance as the heuristic function.

**Code:**

import numpy as np  
  
goal\_state = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 0]])  
  
  
def heuristic\_fn(state, method="manhattan"):  
 h = 0  
 match method:  
 case "manhattan":  
 for i in range(3):  
 for j in range(3):  
 if state[i][j] == goal\_state[i][j]:  
 continue  
 x, y = np.where(goal\_state == state[i][j])  
 if i == x[0]:  
 h += abs(j - y[0])  
 else:  
 h += abs(x[0] - i) + abs(y[0] - j)  
 case "misplaced":  
 for i in range(3):  
 for j in range(3):  
 if state[i][j] != goal\_state[i][j]:  
 h += 1  
 case \_:  
 raise NotImplementedError("Invalid argument")  
 return h  
  
  
def get\_new\_states(state):  
 new\_states = []  
 neighbours = []  
 # Find empty space  
 x, y = np.where(state == 0)  
 x, y = x[0], y[0]  
  
 # Get neighbours  
 n1 = (x+1, y)  
 n2 = (x-1, y)  
 n3 = (x, y-1)  
 n4 = (x, y+1)  
  
 for n in [n1, n2, n3, n4]:  
 if 0 <= n[0] <= 2 and 0 <= n[1] <= 2:  
 neighbours.append(n)  
  
 for neighbour in neighbours:  
 temp = state[neighbour]  
 new = np.copy(state)  
 new[neighbour] = 0  
 new[x, y] = temp  
 new\_states.append(new)  
  
 return np.array(new\_states)  
  
  
def solve(state, heuristic\_method="misplaced"):  
 previous\_states = []  
 while not (state == goal\_state).all():  
 h\_functions = []  
 new\_states = get\_new\_states(state)  
  
 non\_repeated\_states = []  
 if len(previous\_states) != 0:  
 for s in new\_states:  
 flag = False  
 for p in previous\_states:  
 if (p == s).all():  
 flag = True  
 break  
  
 if not flag:  
 non\_repeated\_states.append(s)  
 else:  
 non\_repeated\_states = new\_states  
  
 for new\_state in non\_repeated\_states:  
 h\_functions.append(heuristic\_fn(new\_state, heuristic\_method))  
  
 previous\_states.append(state)  
  
 new\_state\_index = h\_functions.index(min(h\_functions))  
 state = non\_repeated\_states[new\_state\_index]  
  
 return state  
  
  
# start\_state = np.array([[1, 8, 2],  
# [0, 4, 3],  
# [7, 6, 5]])  
  
start\_state = input("Enter start state (space separated):\n").split(" ")  
start\_state\_array = []  
for i in start\_state:  
 start\_state\_array.append(int(i))  
  
start\_state = np.array(start\_state\_array).reshape((3, 3))  
  
print("Start state: \n", start\_state)  
solve\_method = input("Enter method to solve: ")  
print("Goal state: \n", solve(start\_state, solve\_method))

class Node:

    def \_\_init\_\_(self,data,level,fval):

        """ Initialize the node with the data, level of the node and the calculated fvalue """

        self.data = data

        self.level = level

        self.fval = fval

    def generate\_child(self):

        """ Generate child nodes from the given node by moving the blank space

            either in the four directions {up,down,left,right} """

        x,y = self.find(self.data,'\_')

        """ val\_list contains position values for moving the blank space in either of

            the 4 directions [up,down,left,right] respectively. """

        val\_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]

        children = []

        for i in val\_list:

            child = self.shuffle(self.data,x,y,i[0],i[1])

            if child is not None:

                child\_node = Node(child,self.level+1,0)

                children.append(child\_node)

        return children

    def shuffle(self,puz,x1,y1,x2,y2):

        """ Move the blank space in the given direction and if the position value are out

            of limits the return None """

        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

            temp\_puz = []

            temp\_puz = self.copy(puz)

            temp = temp\_puz[x2][y2]

            temp\_puz[x2][y2] = temp\_puz[x1][y1]

            temp\_puz[x1][y1] = temp

            return temp\_puz

        else:

            return None

    def copy(self,root):

        """ Copy function to create a similar matrix of the given node"""

        temp = []

        for i in root:

            t = []

            for j in i:

                t.append(j)

            temp.append(t)

        return temp

    def find(self,puz,x):

        """ Specifically used to find the position of the blank space """

        for i in range(0,len(self.data)):

            for j in range(0,len(self.data)):

                if puz[i][j] == x:

                    return i,j

class Puzzle:

    def \_\_init\_\_(self,size):

        """ Initialize the puzzle size by the specified size,open and closed lists to empty """

        self.n = size

        self.open = []

        self.closed = []

    def accept(self):

        """ Accepts the puzzle from the user """

        puz = []

        for i in range(0,self.n):

            temp = input().split(" ")

            puz.append(temp)

        return puz

    def f(self,start,goal):

        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """

        return self.h(start.data,goal)+start.level

    def h(self,start,goal):

        """ Calculates the different between the given puzzles """

        temp = 0

        for i in range(0,self.n):

            for j in range(0,self.n):

                if start[i][j] != goal[i][j] and start[i][j] != '\_':

                    temp += 1

        return temp

    def process(self):

        """ Accept Start and Goal Puzzle state"""

        print("Enter the start state matrix \n")

        start = self.accept()

        print("Enter the goal state matrix \n")

        goal = self.accept()

        start = Node(start,0,0)

        start.fval = self.f(start,goal)

        """ Put the start node in the open list"""

        self.open.append(start)

        print("\n\n")

        while True:

            cur = self.open[0]

            print("")

            print("\\next step:\n")

            for i in cur.data:

                for j in i:

                    print(j,end=" ")

                print("")

            """ If the difference between current and goal node is 0 we have reached the goal node"""

            if(self.h(cur.data,goal) == 0):

                break

            for i in cur.generate\_child():

                i.fval = self.f(i,goal)

                self.open.append(i)

            self.closed.append(cur)

            del self.open[0]

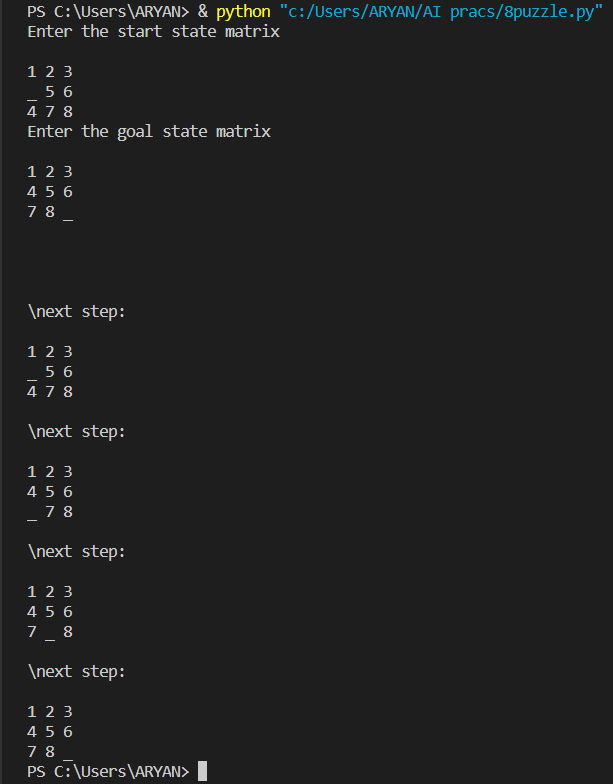
            """ sort the open list based on f value """

            self.open.sort(key = lambda x:x.fval,reverse=False)

puz = Puzzle(3)

puz.process()

**Output:**

****

**Experiment No: 03**

**Program: Write a program to play Tic-Tac-Toe Game.**

**Code:**

def printBoard(board):

    print(board[1] + '|' + board[2] + '|' + board[3])

    print('-+-+-')

    print(board[4] + '|' + board[5] + '|' + board[6])

    print('-+-+-')

    print(board[7] + '|' + board[8] + '|' + board[9])

    print("\n")

def spaceIsFree(position):

    if board[position] == ' ':

        return True

    else:

        return False

def insertLetter(letter, position):

    if spaceIsFree(position):

        board[position] = letter

        printBoard(board)

        if (checkDraw()):

            print("Draw!")

            exit()

        if checkForWin():

            if letter == 'X':

                print("Bot wins!")

                exit()

            else:

                print("Player wins!")

                exit()

        return

    else:

        print("Can't insert there!")

        position = int(input("Please enter new position:  "))

        insertLetter(letter, position)

        return

def checkForWin():

    if (board[1] == board[2] and board[1] == board[3] and board[1] != ' '):

        return True

    elif (board[4] == board[5] and board[4] == board[6] and board[4] != ' '):

        return True

    elif (board[7] == board[8] and board[7] == board[9] and board[7] != ' '):

        return True

    elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):

        return True

    elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):

        return True

    elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):

        return True

    elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):

        return True

    elif (board[7] == board[5] and board[7] == board[3] and board[7] != ' '):

        return True

    else:

        return False

def checkWhichMarkWon(mark):

    if board[1] == board[2] and board[1] == board[3] and board[1] == mark:

        return True

    elif (board[4] == board[5] and board[4] == board[6] and board[4] == mark):

        return True

    elif (board[7] == board[8] and board[7] == board[9] and board[7] == mark):

        return True

    elif (board[1] == board[4] and board[1] == board[7] and board[1] == mark):

        return True

    elif (board[2] == board[5] and board[2] == board[8] and board[2] == mark):

        return True

    elif (board[3] == board[6] and board[3] == board[9] and board[3] == mark):

        return True

    elif (board[1] == board[5] and board[1] == board[9] and board[1] == mark):

        return True

    elif (board[7] == board[5] and board[7] == board[3] and board[7] == mark):

        return True

    else:

        return False

def checkDraw():

    for key in board.keys():

        if (board[key] == ' '):

            return False

    return True

def playerMove():

    position = int(input("Enter the position for 'O':  "))

    insertLetter(player, position)

    return

def compMove():

    bestScore = -800

    bestMove = 0

    for key in board.keys():

        if (board[key] == ' '):

            board[key] = bot

            score = minimax(board, 0, False)

            board[key] = ' '

            if (score > bestScore):

                bestScore = score

                bestMove = key

    insertLetter(bot, bestMove)

    return

def minimax(board, depth, isMaximizing):

    if (checkWhichMarkWon(bot)):

        return 1

    elif (checkWhichMarkWon(player)):

        return -1

    elif (checkDraw()):

        return 0

    if (isMaximizing):

        bestScore = -800

        for key in board.keys():

            if (board[key] == ' '):

                board[key] = bot

                score = minimax(board, depth + 1, False)

                board[key] = ' '

                if (score > bestScore):

                    bestScore = score

        return bestScore

    else:

        bestScore = 800

        for key in board.keys():

            if (board[key] == ' '):

                board[key] = player

                score = minimax(board, depth + 1, True)

                board[key] = ' '

                if (score < bestScore):

                    bestScore = score

        return bestScore

board = {1: ' ', 2: ' ', 3: ' ',

         4: ' ', 5: ' ', 6: ' ',

         7: ' ', 8: ' ', 9: ' '}

printBoard(board)

print("Computer goes first! Good luck.")

print("Positions are as follow:")

print("1, 2, 3 ")

print("4, 5, 6 ")

print("7, 8, 9 ")

print("\n")

player = 'O'

bot = 'X'

global firstComputerMove

firstComputerMove = True

while not checkForWin():

    compMove()

    playerMove()

Alternate Code:

class TicTacToe:

    def \_\_init\_\_(self):

        self.board = []

    def create\_board(self):

        for i in range(3):

            row = []

            for j in range(3):

                row.append('-')

            self.board.append(row)

    def get\_random\_first\_player(self):

        #return random.randint(0, 1)

        no = int(input("Press 1 for Player 'X' \nPress 0 for player 'O' :"))

        return no

    def fix\_spot(self, row, col, player):

        self.board[row][col] = player

    def is\_player\_win(self, player):

        win = None

        n = len(self.board)

        #checking rows

        for i in range(n):

            win = True

            for j in range(n):

                if self.board[i][j] != player:

                    win = False

                    break

            if win:

                return win

        #checking columns

        for i in range(n):

            win = True

            for j in range(n):

                if self.board[j][i] != player:

                    win = False

                    break

            if win:

                return win

        #checking diagonals

        win = True

        for i in range(n):

            if self.board[i][i] != player:

                win = False

                break

        if win:

            return win

        win = True

        for i in range(n):

            if self.board[i][n - 1 - i] != player:

                win = False

                break

        if win:

            return win

        return False

        for row in self.board:

            for item in row:

                if item == '-':

                    return False

        return True

    def is\_board\_filled(self):

        for row in self.board:

            for item in row:

                if item == '-':

                    return False

        return True

    def swap\_player\_turn(self, player):

        return 'X' if player == 'O' else 'O'

    #displaying board

    def show\_board(self):

        for row in self.board:

            for item in row:

                print(item, end=" ")

            print()

    def start(self):

        self.create\_board()

        player = 'X' if self.get\_random\_first\_player() == 1 else 'O'

        while True:

            print(f"Player {player} turn")

            self.show\_board()

            #taking user input as rto and column wise

            row, col = list(

                map(int, input("Enter row and column numbers to fix spot: ").split()))

            print()

            #fixing the user input

            self.fix\_spot(row - 1, col - 1, player)     #row = 1 will place player's value on 0 also if row = 3 will place value to row = 2

            # checking whether current player is won or not

            if self.is\_player\_win(player):

                print(f"Player {player} wins the game!")

                break

            # checking whether the game is draw or not

            if self.is\_board\_filled():

                print("Match Draw!")

                break

            # swapping the turn

            player = self.swap\_player\_turn(player)

        # showing the final view of board

        print()

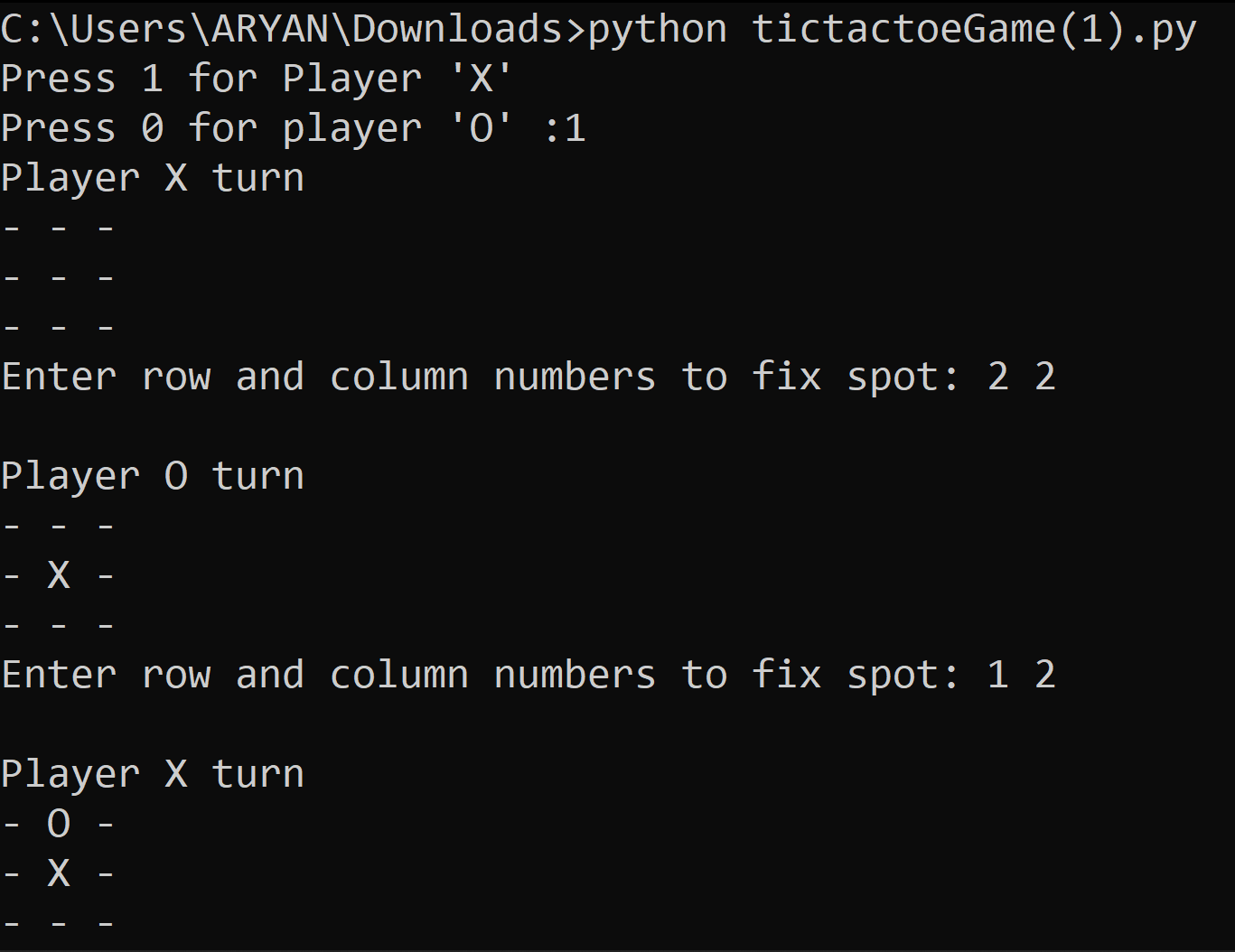
        self.show\_board()

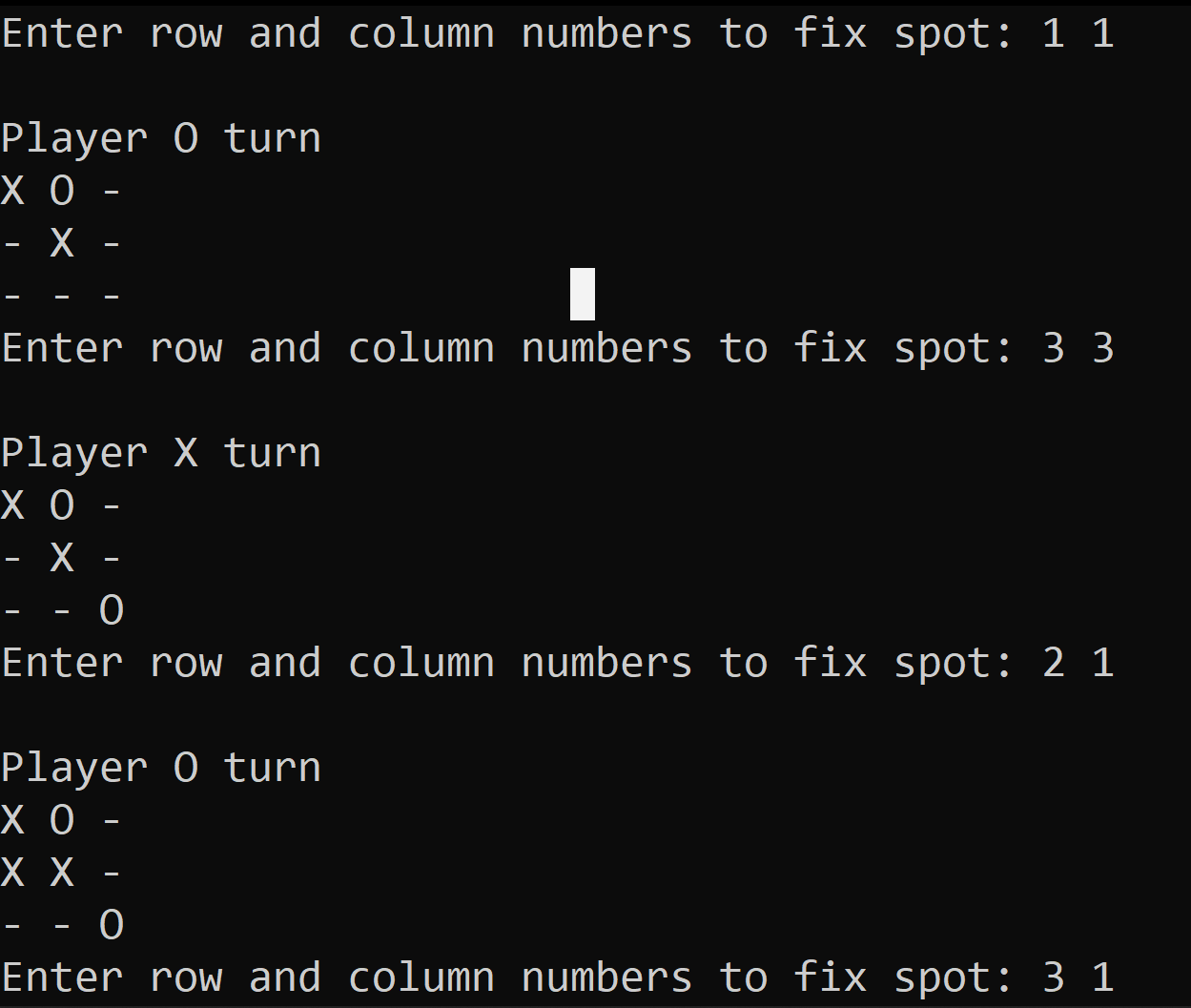
# starting the game

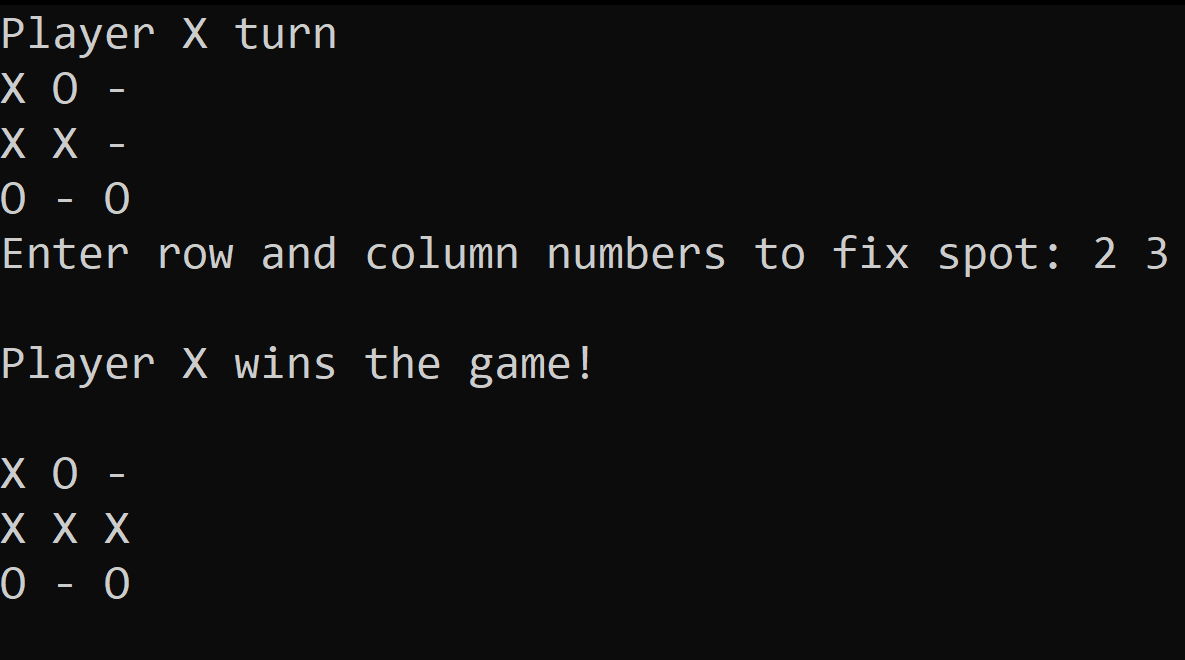
tic\_tac\_toe = TicTacToe()

tic\_tac\_toe.start()

**Output:**







**Experiment No: 04**

**Program:** Implement Truth Table for the following equations:

1. P or Q
2. (P and Q or R) or (P or Q and R)
3. (P or Q and R) and (P and Q or R)

**Code:**

print("Truth Tables for Logical operators\n")

print("---------------------------------------")

print("Table for AND\n")

print("---------------------------------------")

for i in range(4):

  if i<2:

    p = 0

  else:

    p = 1

  if i%2==0:

    q= 0

  else:

    q=1

  if p and q:

    print(p," ",q,":  True")

  else:

    print(p," ",q,":  False")

print("---------------------------------------")

print("\nTable for OR\n")

print("---------------------------------------")

for i in range(4):

  if i<2:

    p = 0

  else:

    p = 1

  if i%2==0:

    q= 0

  else:

    q=1

  if p or q:

    print(p," ",q,":  True")

  else:

    print(p," ",q,":  False")

print("---------------------------------------")

print("\nTable for NOT")

print("---------------------------------------")

print("\nTable for ~p\n")

for i in range(2):

  if i<1:

    p = 0

  else:

    p = 1

  if p==0 :

    print(p,":  True")

  else:

    print(p,":  False")

print("\nTable for ~q\n")

for i in range(2):

  if i<1:

    q = 1

  else:

    q = 0

  if q==0 :

    print(q,":  True")

  else:

    print(q,":  False")

print("---------------------------------------")

print("\nTable for P and Q and R\n")

print("---------------------------------------")

for i in range(8):

  if i<4:

    p = 1

  else:

    p = 0

  if i<2:

    q = 1

  elif i>1 and i<4:

    q=0

  elif i>3 and i<6:

        q=1

  else :

       q = 0

  if i%2==0:

      r=1

  else:

      r=0

  if p and q and r:

        print(p," ",q," ",r,":  True")

  else:

        print(p," ",q," ",r,":  False")

print("---------------------------------------")

print("\nTable for P OR Q OR R\n")

print("---------------------------------------")

for i in range(8):

  if i<4:

    p = 1

  else:

    p = 0

  if i<2:

    q = 1

  elif i>1 and i<4:

    q=0

  elif i>3 and i<6:

        q=1

  else :

       q = 0

  if i%2==0:

      r=1

  else:

      r=0

  if p or q or r:

        print(p," ",q," ",r,":  True")

  else:

        print(p," ",q," ",r,":  False")

print("---------------------------------------")

print("\nTable for (P AND Q OR R) OR (P OR Q AND R)\n")

print("---------------------------------------")

for i in range(8):

  if i<4:

    p = 1

  else:

    p = 0

  if i<2:

    q = 1

  elif i>1 and i<4:

    q=0

  elif i>3 and i<6:

        q=1

  else :

       q = 0

  if i%2==0:

      r=1

  else:

      r=0

  if (p and q or r) or (p or q and r):

        print("(",p,"^",q,"v",r,")","v","(",p,"v",q,"^",r,")",":  True")

  else:

        print("(",p,"^",q,"v",r,")","v","(",p,"v",q,"^",r,")",":  False")

print("---------------------------------------")

print("\nTable for (P OR Q AND R) AND (P AND Q OR R)\n")

print("---------------------------------------")

for i in range(8):

  if i<4:

    p = 1

  else:

    p = 0

  if i<2:

    q = 1

  elif i>1 and i<4:

    q=0

  elif i>3 and i<6:

        q=1

  else :

       q = 0

  if i%2==0:

      r=1

  else:

      r=0

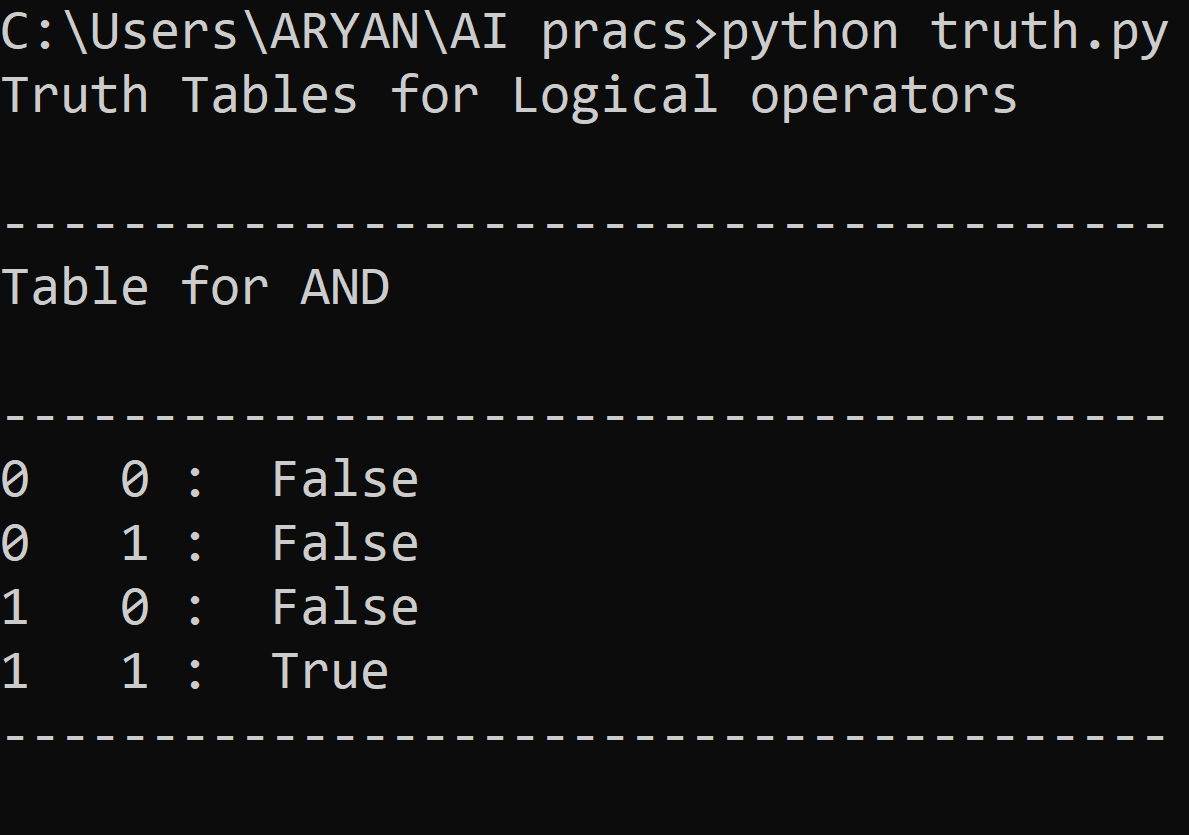
  if (p or q and r) and (p and q or r):

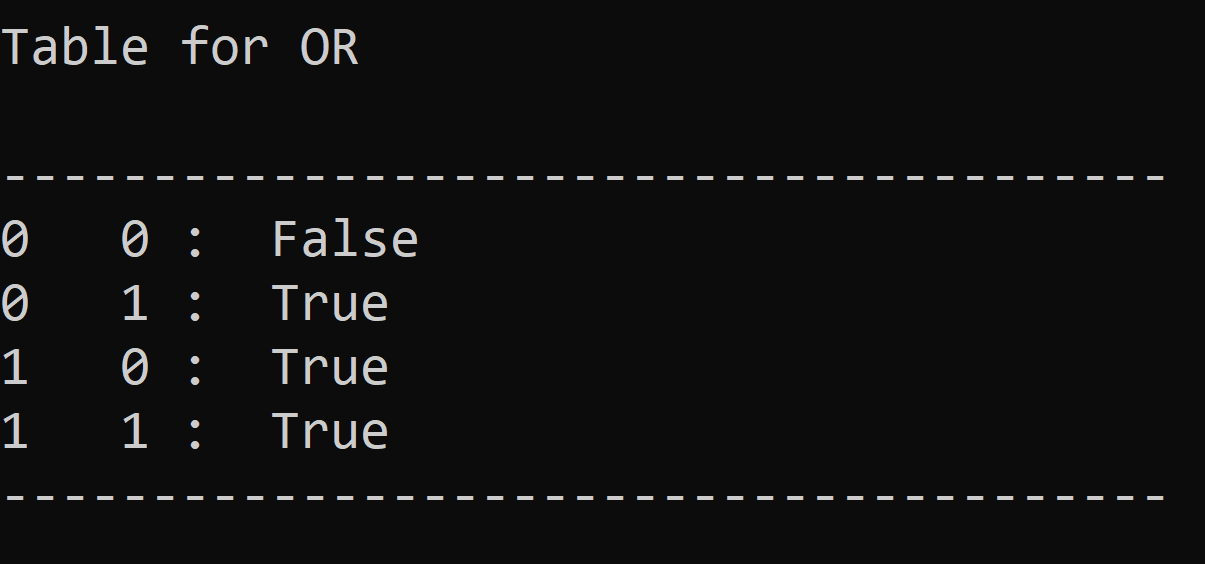
        print("(",p,"v",q,"^",r,")","^","(",p,"^",q,"v",r,")",":  True")

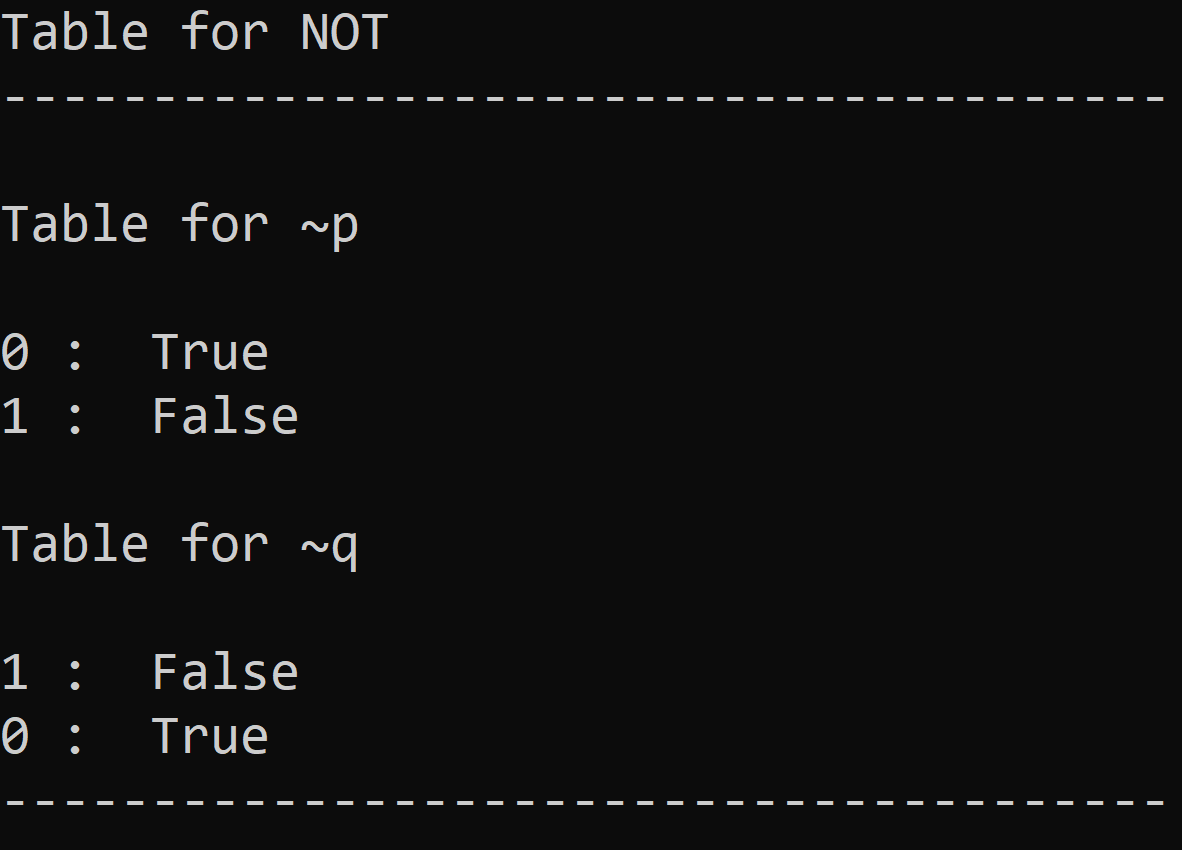
  else:

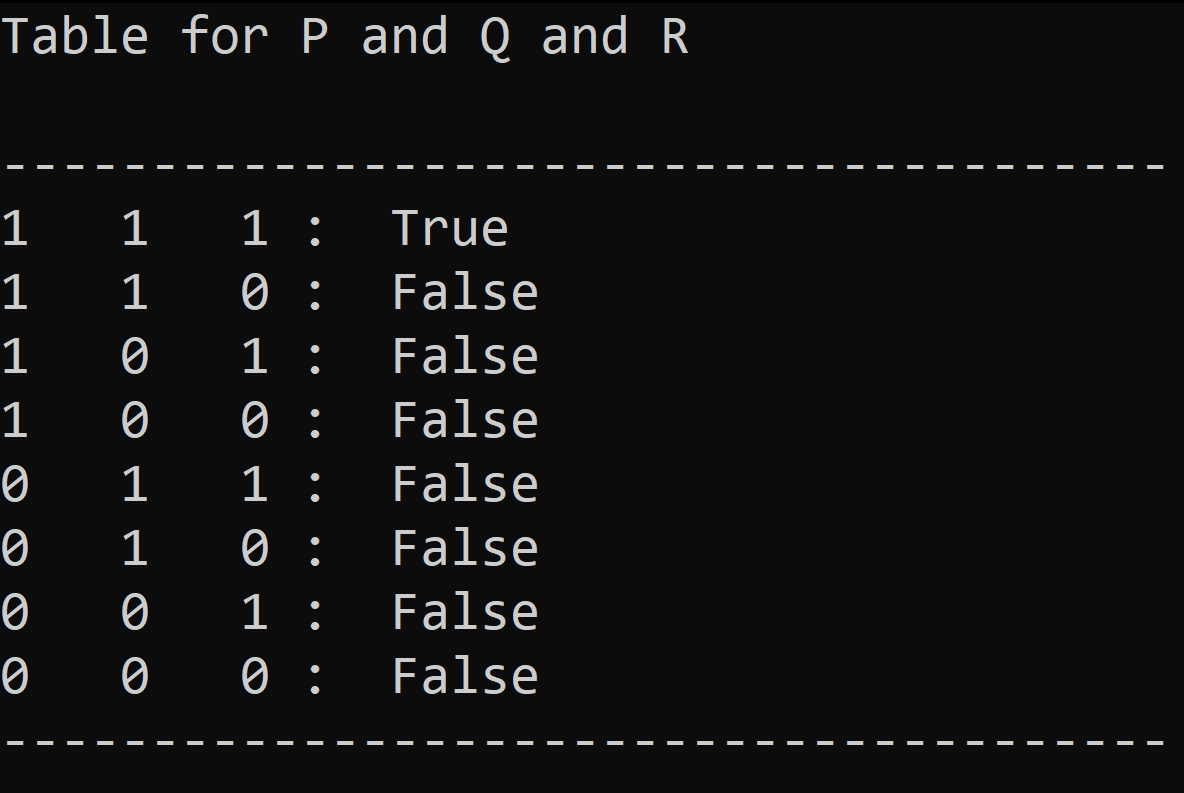
        print("(",p,"v",q,"^",r,")","^","(",p,"^",q,"v",r,")",":  False")

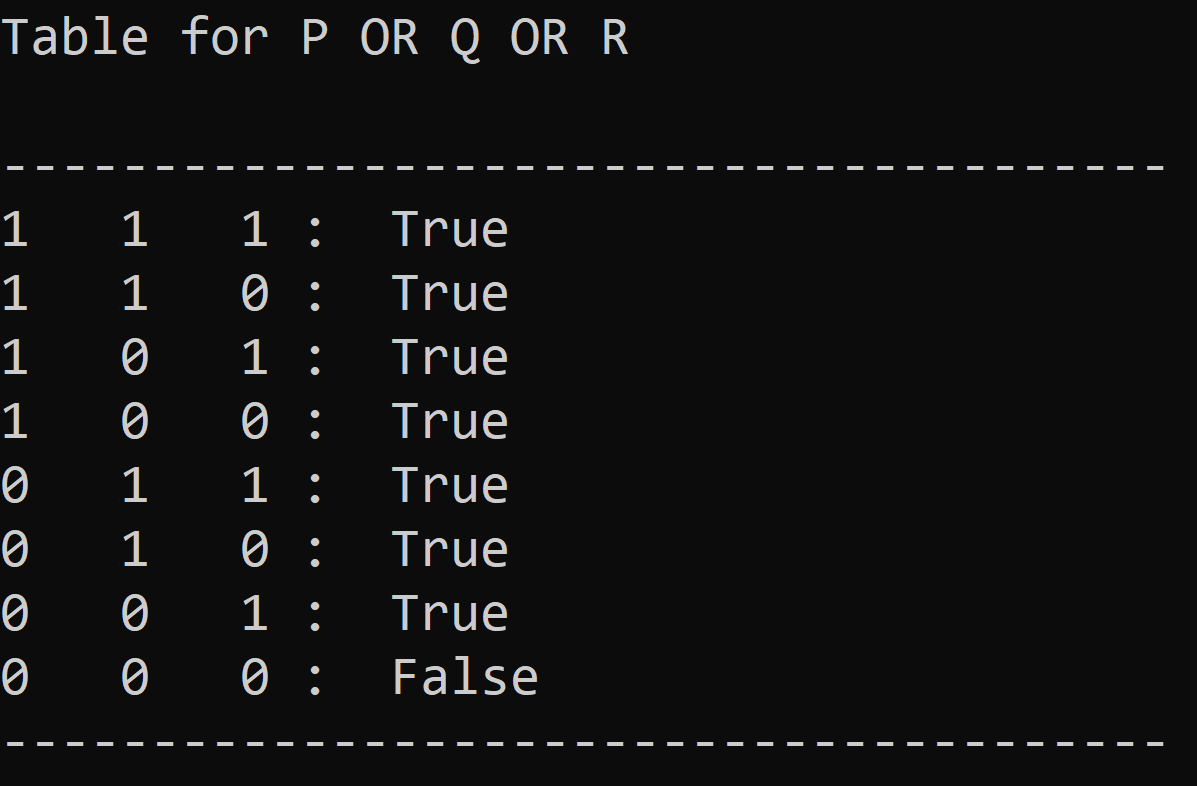
**Output:**

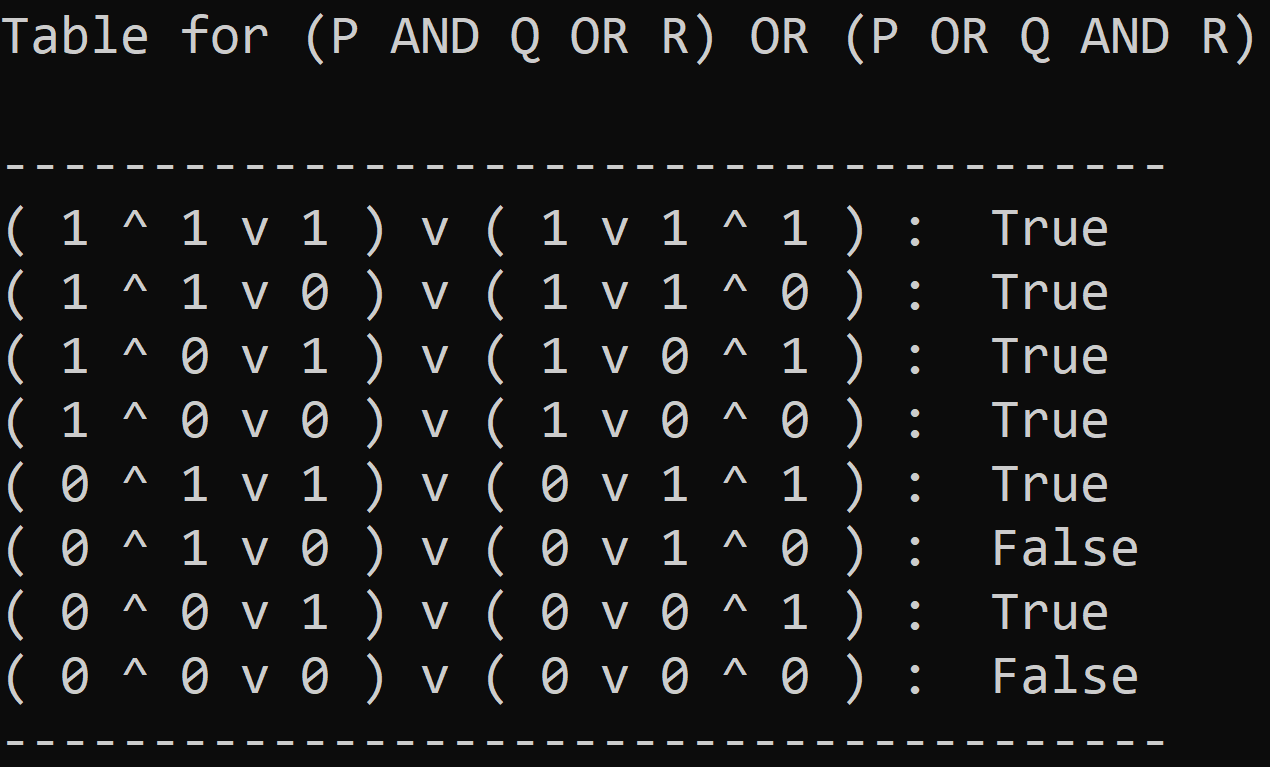


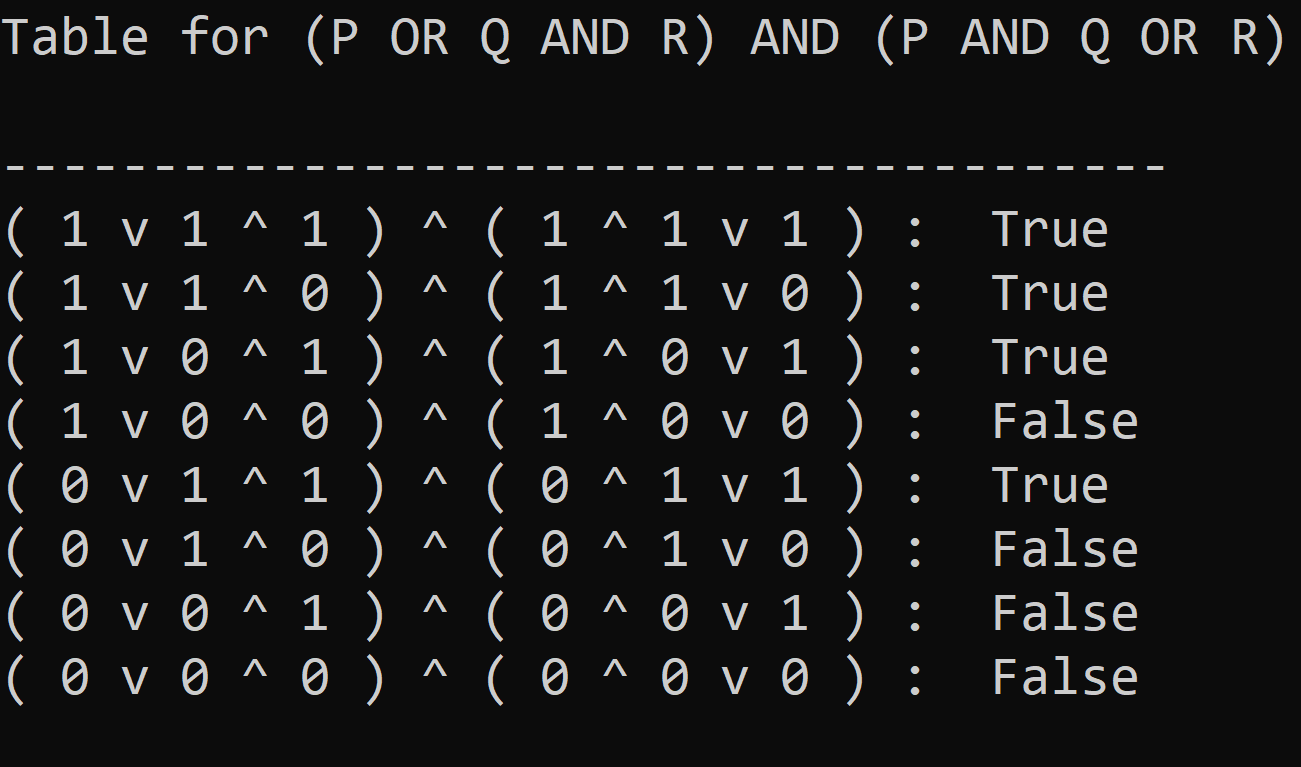












**Experiment No: 05**

**Program:** Write a Program to Solve 8 Queens Problem.

**Code:**

boardcount = 0

def IsBoardOk (board, row, col) :

   #if queen X takes position to the left of column col on the same row.

   for c in range(col) :

       if (board[row][c] == 'X') :

           return False

   #if queen X takes position on the upper left diagonal

   for r, c in zip(range(row-1, -1, -1), range(col-1, -1, -1)) :

       if (board[r][c] == 'X') :

           return False

   #if queen X takes position on the lower left diagonal

   for r, c in zip(range(row+1, len(board), 1), range(col-1, -1, -1)) :

      if (board[r][c] == 'X') :

          return False

   return True

def DisplayBoard (board) :

    for row in board :

        print(row)

def PlaceNQueens (board, col) :

    # If all the columns have a queen 'Q', a solution has been found.

    global boardcount

    if (col >= len(board)) :

        boardcount += 1

        print("Chessboard " + str(boardcount))

        print("--------------------------")

        DisplayBoard(board)

        print("--------------------------\n")

    else :

        # Else try placing the queen on each row of the column and check if the chessboard remains OK.

        for row in range(len(board)) :

            board[row][col] = 'X'

            if (IsBoardOk(board, row, col) == True) :

                # Chess board was OK, hence try placing the queen 'Q' in the next column.

                PlaceNQueens(board, col + 1)

            board[row][col] = '.'; # As previously placed queen was not valid, restore '.'

def main() :

   board = []

   N = int(input("Enter No. of queens : "))

   for i in range(N) :

       row = ["."] \* N

       board.append(row)

   # Start placing the queen 'X' from the 0'th column.

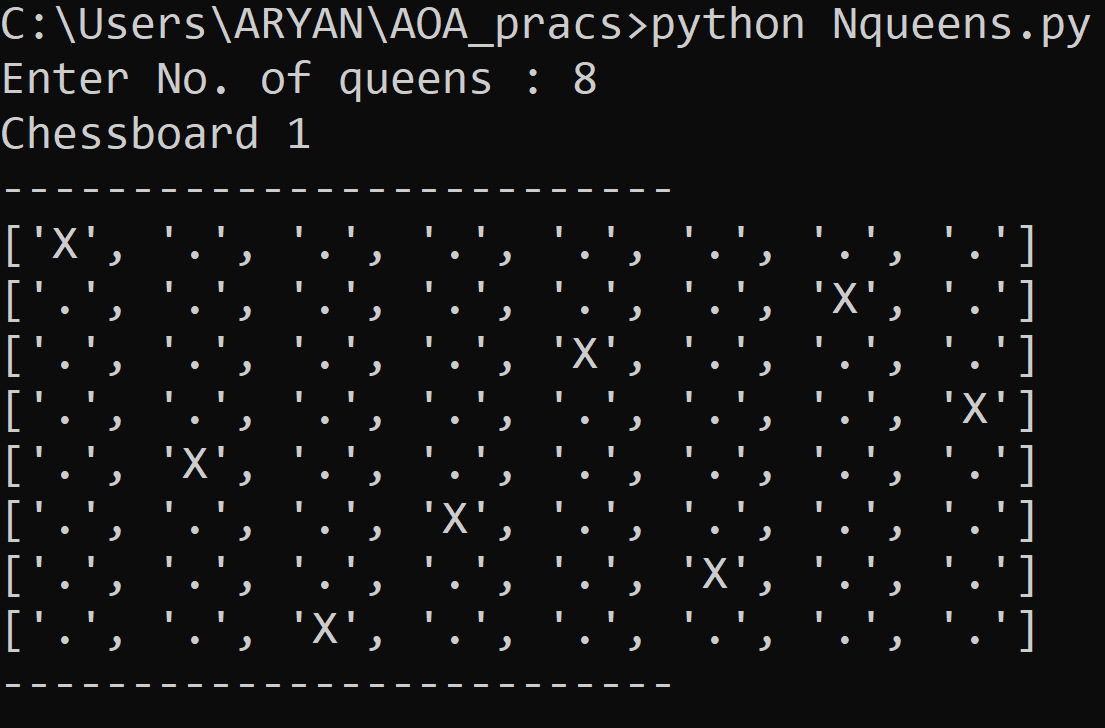
   PlaceNQueens(board, 0)

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

    main()

**Output:**

**Output:**



**Experiment No: 06**

**Program:** Write a program to list the name of enemies of America the names of criminals using Forward Chaining from the following statements :

1. It is a crime for an American to sell weapons to the enemy of America.
2. Country Nono is a enemy of America.
3. Nono has some Missiles.
4. All the missiles were sold to Nono by Colonel.
5. Missile is weapon.
6. Colonel is an American

**Code:**

# Import libraries

import aima3.utils

import aima3.logic

# The main entry point for this module

def forward\_chaining():

    # Create an array to hold clauses

    clauses = []

    # Add first-order logic clauses (rules and fact)

    clauses.append(aima3.utils.expr("(American(x) & Weapon(y) & Sells(x, y, z) & Hostile(z)) ==> Criminal(x)"))

    clauses.append(aima3.utils.expr("Enemy(Nono, America)"))

    clauses.append(aima3.utils.expr("Owns(Nono, M1)"))

    clauses.append(aima3.utils.expr("Missile(M1)"))

    clauses.append(aima3.utils.expr("(Missile(x) & Owns(Nono, x)) ==> Sells(Colonel, x, Nono)"))

    clauses.append(aima3.utils.expr("American(Colonel)"))

    clauses.append(aima3.utils.expr("Missile(x) ==> Weapon(x)"))

    # Create a first-order logic knowledge base (KB) with clauses

    KB = aima3.logic.FolKB(clauses)

    # Add rules and facts with tell

    KB.tell(aima3.utils.expr("Enemy(x, America) ==> Hostile(x)"))

    # Get information from the knowledge base with ask

    hostile = aima3.logic.fol\_fc\_ask(KB, aima3.utils.expr('Hostile(x)'))

    criminal = aima3.logic.fol\_fc\_ask(KB, aima3.utils.expr('Criminal(x)'))

    # Print answers

    print('enemy?')

    print(list(hostile))

    print('\nCriminal?')

    print(list(criminal))

    print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": forward\_chaining()

**Output:**

