Write a program to Implement TIC-TAC-TOE game using Python

Program :

```
import random
finalBoard = ["-"]*10
#player moves (pm)
pm = []
#computer moves (cm)
cm = []

winPattern =
[[1,2,3],[4,5,6],[7,8,9],[1,4,7],[2,5,8],[3,6,9],[3,5,7],[1,5,9]]

def showBoard():
    k = 1
    for i in range(1,10):
        print(finalBoard[i],end=" ")
        if(i%3==0):
            print()


def playerTurn(symbol):
    pos = int(input())
    while(pos not in pm and pos not in cm):
        print("Enter a valid position : (1-9) ")
        finalBoard[pos] = symbol
        pm.append(pos)
#cs -> computer symbol
#ps -> player symbol
def computerTurn(cs,ps):
    for i in range(1,10):
        board = finalBoard[:]
        if i not in cm and i not in pm:
            makeMove(board, cs, i)
            if isWinner(board,cs):
                return i
    for i in range(1,10):
        board = finalBoard[:]
        if i not in pm and i not in cm:
            makeMove(board, ps, i)
            if(isWinner(board,ps)):
                return i
```

```
    move = chooseRandomMoveFromList([1,3,7,9])
    if move != None:
        return move
    if finalBoard[5]=="-":
        return 5
    return chooseRandomMoveFromList([2,4,6,8])


def isWinner(board,symbol):
    for pattern in winPattern:
        if(board[pattern[0]]==symbol and board[pattern[1]]==symbol and
board[pattern[2]]==symbol):
            return True
    return False


def isTie(cs,ps):
    if isWinner(finalBoard,ps)==False and
isWinner(finalBoard,cs)==False :
       return True
    return False


def makeMove(board,letter,pos):
    board[pos] = letter

def chooseRandomMoveFromList(movesList):
    possibleMoves = []
    for i in movesList:
        if i not in pm and i not in cm:
            possibleMoves.append(i)
    if(len(possibleMoves)!=0):
        return random.choice(possibleMoves)
    else:
        return None

def start():
    #computer starts the game everytime
    plays = 0
    flag = False
    print("Enter Your Symbol : ")
    playerSymbol = input()
    if(playerSymbol == "X" or playerSymbol == "x"):
        computerSymbol = "O"
        playerSymbol = "X"
```

```
    else:
        playerSymbol = "O"
        computerSymbol = "X"
    print("\nYour symbol is : ",playerSymbol)
    #flag  represents whether the game is over or not
    while(plays <= 8 and flag != True):
        if(flag != True):
            print("Player Turn : \n")
            print("Enter your position (1-9) ")
            playerTurn(playerSymbol)
            showBoard()
            flag = isWinner(finalBoard, playerSymbol)

        if(flag != True):
            print("Computer Turn : \n")
            pos = computerTurn(computerSymbol,playerSymbol)
            cm.append(pos)
            finalBoard[pos] = computerSymbol
            showBoard()
            flag = isWinner(finalBoard,computerSymbol)

        plays = plays + 2

    if(flag != True):
        print("Computer Turn :\n")
        computerTurn(computerSymbol,playerSymbol)
        if(isWinner(finalBoard,computerSymbol)):
            print("Computer Won the game ")
        elif(isWinner(finalBoard,playerSymbol)):
            print("You won the game ")
        else:
            print("TIE GAME")
    print("Final Board : ")
    showBoard()


start()
```

OUTPUT :

```
Enter Your Symbol :
o

Your symbol is :  O
Player Turn :

Enter a valid position : (1-9)
1
O - -
- - -
- - -
Computer Turn :

O - X
- - -
- - -
Player Turn :

Enter a valid position : (1-9)
2
O O X
- - -
- - -
Computer Turn :

O O X
- - -
- - X
```

```
Player Turn :

Enter a valid position : (1-9)
6
O O X
- - O
- - X
Computer Turn :

O O X
- - O
X - X
Player Turn :

Enter a valid position : (1-9)
8
O O X
- - O
X O X
Computer Turn :

O O X
- X O
X O X
Final Board :
O O X
- X O
X O X
```

Write a program to Implement Water Jug problem

Program :

```
from collections import defaultdict
jug1, jug2, aim1,aim2 = 4, 3, 4,1
visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
    if (amt1 == aim1 and amt2 == aim2) or (amt2 == aim1 and amt1 ==
aim2):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))
    else:return False
print("Steps:")
waterJugSolver(0, 0)
```

OUTPUT :

Steps:

0 0

4 0

4 3

0 3

3 0

3 3

4 2

0 2

2 0

2 3

4 1

Write a Program to Implement Breadth First Search Traversal (BFS)

Program :

```
from collections import defaultdict

class Graph:

    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):
        visited = [False] * (len(self.graph))
        queue = []
        queue.append(s)
        visited[s] = True

        while queue:
            s = queue.pop(0)
            print (s, end = " ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                visited[i] = True

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
" (starting from vertex 2)")
g.BFS(2)
```

OUTPUT :

2 0 3 1

Write a Program to Implement Depth First Search Traversal (DFS)

Program :

```
from collections import defaultdict

class Graph:

    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        visited[v]= True
        print (v)
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    def DFS(self):
        V = len(self.graph)
        visited =[False]*(V)
        for i in range(V):
            if visited[i] == False:
                self.DFSUtil(i, visited)

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Depth First Traversal")
g.DFS()
```

```
OUTPUT :

0 1 2 3
```

Write a Program to Implement Eight Puzzle Problem :

Program :

```
def countMisMatch(state):
    count = 0
    finalState  = [5,3,6,7,0,2,4,1,8]
    for i in range(9):
        if(state[i]!=0 and state[i]!=finalState[i]):
            count = count + 1
    return count

def printMatrix(state):
    for i in range(9):
        if(i%3 == 0):
            print()
        print(state[i],end=" ")


def solvePuzzle(state):
    level = 0
    h = 1
    while(h>0):
        level = level + 1
        #index_0 represents the position of element 0 which represents
empty space
        index_0 = state.index(0)
        if(index_0 == 0):
            arr = [1,3]
            state,h = move(arr,index_0,state)
        elif(index_0 == 1):
            arr = [0,2,4]
            state,h = move(arr,index_0,state)
        elif(index_0 == 2):
            arr = [1,5]
            state,h = move(arr,index_0,state)
        elif(index_0 == 3):
            arr = [0,4,6]
            state,h = move(arr,index_0,state)
        elif(index_0 == 4):
            arr = [1,3,5,7]
            state,h = move(arr,index_0,state)
        elif(index_0 == 5):
            arr = [2,4,8]
            state,h = move(arr,index_0,state)
```

```
        elif(index_0 == 6):
            arr = [3,4]
            state,h = move(arr,index_0,state)
        elif(index_0 == 7):
            arr = [4,6,8]
            state,h = move(arr,index_0,state)
        elif(index_0 == 8):
            arr = [5,7]
            state,h = move(arr,index_0,state)

        print("\n\nLevel ::  ",level)
        print("Hueristic value (h)  ",h)
        printMatrix(state)

def move(arr,pos,state):
    store_state = state[:]
    store_h = 99999
    for i in range(len(arr)):
        dup_state = state[:]
        temp = dup_state[pos]
        dup_state[pos] = dup_state[arr[i]]
        dup_state[arr[i]] = temp
        temp_h = countMisMatch(dup_state)
        if(temp_h < store_h):
            store_h = temp_h
            store_state = dup_state
    return store_state,store_h


initial_state = [3,7,6,5,1,2,4,0,8]
solvePuzzle(initial_state)
```

OUTPUT :

```
Level ::   1
Hueristic value (h)   3

3 7 6
5 0 2
4 1 8
Level ::   2
Hueristic value (h)   3

3 0 6
5 7 2
4 1 8
Level ::   3
Hueristic value (h)   2

0 3 6
5 7 2
4 1 8
Level ::   4
Hueristic value (h)   1

5 3 6
0 7 2
4 1 8
Level ::   5
Hueristic value (h)   0

5 3 6
7 0 2
4 1 8
```

Write a Program to Implement Towers of Hanoi Problem .

Program :

```
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n == 0:
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk", n, "from rod", from_rod, "to rod", to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)


N = 3
TowerOfHanoi(N, 'A', 'C', 'B')
```

OUTPUT :

Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C

Write a program to implement Missionaries and Cannibals Problem

Program :

```
good = {(0, 0), (3, 0), (0, 3), (3, 1), (3, 2), (2, 2), (1, 1), (0,
2), (0, 1)}
count = 0

def printsolution(ans, d):
    print(f"Solution - {count}:")
    print("(M, C, B)")
    cur = ans
    while d[cur] != cur:
        print(cur)
        cur = d[cur]
    print(cur)

def solve(v, d):
    global count
    if v[0] == v[1] == 0:
        count += 1
        printsolution(v, d)
        return
    pos = [(-1, 0), (-1, -1), (-2, 0), (0, -2), (0, -1)]
    a, b, c = v[0], v[1], v[2]
    if c == 0:
        mul = -1
    else:
        mul = 1
    for i in pos:
        x, y = i
        na = a+x*mul
        nb = b+y*mul
        t = (na, nb, c ^ 1)
        if (na, nb) in good and t not in d:
            d[t] = v
            solve(t, d)
            del d[t]

s = (3, 3, 1)
d = {s: s}
solve(s, d)
```

OUTPUT :

```
Solution - 1:
(M, C, B)
(0, 0, 0)
(1, 1, 1)
(0, 1, 0)
(0, 3, 1)
(0, 2, 0)
(2, 2, 1)
(1, 1, 0)
(3, 1, 1)
(3, 0, 0)
(3, 2, 1)
(2, 2, 0)
(3, 3, 1)
Solution - 2:
(M, C, B)
(0, 0, 0)
(0, 2, 1)
(0, 1, 0)
(0, 3, 1)
(0, 2, 0)
(2, 2, 1)
(1, 1, 0)
(3, 1, 1)
(3, 0, 0)
(3, 2, 1)
(2, 2, 0)
(3, 3, 1)
```

```
Solution - 3:
(M, C, B)
(0, 0, 0)
(1, 1, 1)
(0, 1, 0)
(0, 3, 1)
(0, 2, 0)
(2, 2, 1)
(1, 1, 0)
(3, 1, 1)
(3, 0, 0)
(3, 2, 1)
(3, 1, 0)
(3, 3, 1)
Solution - 4:
(M, C, B)
(0, 0, 0)
(0, 2, 1)
(0, 1, 0)
(0, 3, 1)
(0, 2, 0)
(2, 2, 1)
(1, 1, 0)
(3, 1, 1)
(3, 0, 0)
(3, 2, 1)
(3, 1, 0)
(3, 3, 1)
```

Write a program to implement Monkey Banana Problem

Program :

```
def solve(banana,box,height,monkey,hold):
    if monkey==banana and height==1:
        ans.append("Monkey took banana")
        return True
    if (banana,box,height,monkey,hold) in d:
        return False
    found=0
    d[(banana,box,height,monkey,hold)]=1
    options={1:"Move to box", 2:"Move to banana", 3:"Climb onto the
box", 4:"Hold box to move"}
    for option in options:
        if option==1 and hold==0 and height==0 and
solve(banana,box,height,box,hold):
            ans.append(options[option])
            found=1
            break
        elif option==2 and height==0 and ((hold==1 and
solve(banana,banana,height,banana,hold)) or (hold==0 and
solve(banana,box,height,banana,hold))):
            ans.append(options[option])
            found=1
            break
        elif option==3 and height==0 and monkey==box and
solve(banana,box,height+1,monkey,0):
            ans.append(options[option])
            found=1
            break
        elif option==4 and height==0 and monkey==box and
solve(banana,box,height,monkey,1):
            ans.append(options[option])
            found=1
            break
    return found

n=int(input("Enter the size of the world: "))
world=[[0]*n for i in range(n)]
x,y=map(int,input("Enter tree position: ").split())
world[x][y]=1
x,y=map(int,input("Enter box position: ").split())
world[x][y]=-1
x,y=map(int,input("Enter monkey position: ").split())
```

```
for i in range(n):
    for j in range(n):
        if world[i][j]==1:
            print(f"Monkey found the banana tree at ({i},{j})")
            banana=(i,j)
        if world[i][j]==-1:
            print(f"Box found at ({i},{j})")
            box=(i,j)
d={}
ans=[]
solve(banana,box,0,(x,y),0)
ans.reverse()
print(*ans,sep="\n")
```

OUTPUT :

```
Enter the size of the world: 5
Enter tree position: 2 3
Enter box position: 1 2
Enter monkey position: 0 0
Box found at (1,2)
Monkey found the banana tree at (2,3)
Move to box
Hold box to move
Move to banana
Climb onto the box
Monkey took banana
```

Write a program to implement N-Queens Problem

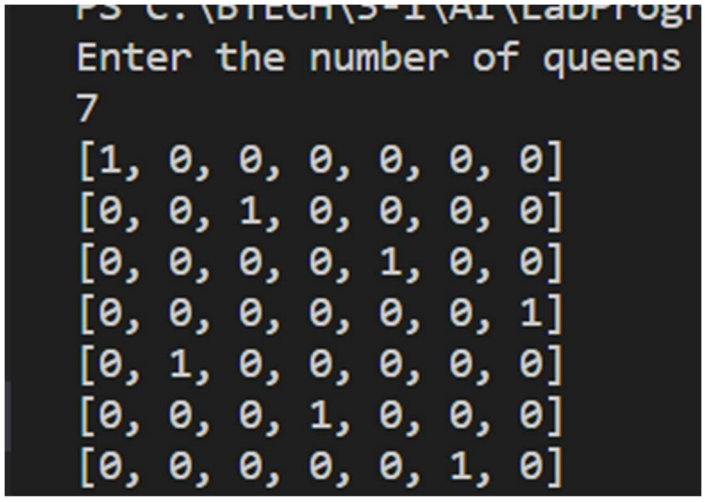
Program :

```
def is_attack(i, j):
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(is_attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                #recursion
                #wether we can put the next queen with this arrangment
or not
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0

    return False
#Number of queens
print ("Enter the number of queens")
N = int(input())
board = [[0]*N for _ in range(N)]
N_queen(N)
for i in board:
    print (i)
```

OUTPUT :


Enter the number of queens
7
[1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0]