```
import random
def TicTacToe():
    board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
    end = False
    MagicSquare = [8, 3, 4, 1, 5, 9, 6, 7, 2]
    print("Welcome to tic-tac-toe game:")
    print("game rules ")
    print("player token :X")
    print("oponent token:O")
    print(" player who gets 3 consecutive tokens first ,wins the game")
    print("game starts Now")

    def PrintBoard():
        print()
        print('', board[0], "|", board[1], "|", board[2])
        print("---|---|---")
        print('', board[3], "|", board[4], "|", board[5])
        print("---|---|---")
        print('', board[6], "|", board[7], "|", board[8])
        print()

    def isBoardFull(board):
        for i in range(1, 10):
            if isSpaceFree(board, i):
                return False
        return True

    def GetNumber():
        while True:
            number = input()
            try:
                number = int(number)
                if number in range(1, 10):
                    return number
                else:
                    print("\nNumber not on board")
            except ValueError:
                print("\nThat's not a number. Try again")
                continue

    def getBoardCopy(board):
        dupeBoard = []
        for i in board:
            dupeBoard.append(i)
        return dupeBoard
```

```
def isSpaceFree(board1, move):
    return board1[move - 1] == ' '

def makeMove(board1, letter, move):
    board1[move - 1] = letter

def chooseRandomMoveFromList(board, movesList):
    possibleMoves = []
    for i in movesList:
        if isSpaceFree(board, i):
            possibleMoves.append(i)
    if len(possibleMoves) != 0:
        return random.choice(possibleMoves)
    else:
        return None

def computerchoice():
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, 'O', i)
            if CheckWin(copy, 'O'):
                return i
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, 'X', i)
            if CheckWin(copy, 'X'):
                return i
    if isSpaceFree(board, 5):
        return 5
    move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move is not None:
        return move
    return chooseRandomMoveFromList(board, [2, 4, 6, 8])

def Turn(player):
    placing_index = GetNumber() - 1
    if board[placing_index] == "X" or board[placing_index] == "O":
        print("\nBox already occupied. Try another one")
        Turn(player)
    else:
        board[placing_index] = player
```

```
def Turn1(move):
    board[move - 1] = 'O'

def CheckWin(board1, player):
    for x in range(9):
        for y in range(9):
            for z in range(9):
                if x != y and y != z and z != x:
                    if board1[x] == player and board1[y] == player and board1[z] == player:
                        if MagicSquare[x] + MagicSquare[y] + MagicSquare[z] == 15:
                            return True

def isBoardFull(board):
    count = 0
    for a in range(9):
        if board[a] == "X" or board[a] == "O":
            count += 1
    if count == 9:
        print("The game ends in a Tie\n")
        return True

while True:
    PrintBoard()
    end = CheckWin(board, "O")
    if end:
        print("Computer wins the game")
        break
    else:
        if isBoardFull(board):
            break
    print("Choose a box player X")
    Turn("X")
    PrintBoard()
    end = CheckWin(board, "X")
    if end:
        print("You win the game")
        break
    else:
        if isBoardFull(board):
            break
    move = computerchoice()
    Turn1(move)

TicTacToe()
```

```
from collections import defaultdict
import math;
jug1=int(input("enter the jug1 value"))
jug2=int(input("enter the jug2 value"))
aim=int(input("enter the aim"))
visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        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
def check():
    if (jug1<=aim) and (jug2<=aim):
        print("Not Possible")
        return True
    elif (jug1/2==jug2 or jug2/2==jug1) and (jug1!=aim and jug2!=aim):
        print("Not Possible")
        return True
    elif(aim%(math.gcd(jug1,jug2))!=0):
        print("Not Possible")
        return True
result=check();
if result!=True:
    print("Steps: ")
    waterJugSolver(0, 0)
```

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph=defaultdict(list);
        self.bfs=""
        self.found=False
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self,root,search):
        visited=[]
        queue=[]
        self.bfs=""
        visited.append(root)
        queue.append(root)
        while queue:
            m=queue.pop(0)
            self.bfs=self.bfs+m+""
            if(m==search):
                self.found=True
                return
            for neighbour in self.graph[m]:
                if neighbour not in visited:
                    visited.append(neighbour)
                    queue.append(neighbour)
g=Graph()
n=int(input("enter no.of nodes\n"))
root=input("enter root node\n")
search=input("enter goal node\n")
print("enter vertices of tree\n")
for i in range(0,n-1):
    s=input()
    x=s.split(",")
    g.addEdge(x[0],x[1])
g.BFS(root,search)
if(g.found):
    print("following is the breadth-first search\n")
    print(g.bfs)
else:
    print("given search element is not found in tree\n")
```

```
from collections import defaultdict;
class Graph:
    def __init__(self):
        self.graph=defaultdict(list);
        self.dfs="";
        self.found=False;
        self.flag=False;
    def addEdge(self,u,v):
        self.graph[u].append(v);
    def DFSutil(self,root,search,visited):
        visited.add(root)
        self.dfs=self.dfs+root+" ";
        if(root==search):
            self.found=True;
            return False;
        for neighbour in self.graph[root]:
            if neighbour not in visited:
                if self.DFSutil(neighbour,search,visited)==False:
                    return False;
    def DFS(self,root,search):
        visited=set();
        self.DFSutil(root,search,visited)
g=Graph();
n=int(input("enter the no.of nodes"));
root=input("enter root node")
search=input("enter search element")
print("enter the verices of tree")
for i in range(0,n-1):
    s=input();
    x=s.split(",")
    g.addEdge(x[0],x[1])
g.DFS(root,search);
if(g.found):
    print("Following is the Depth-First Search")
    print(g.dfs);
else:
     print("given element not found in tree")
```

```
import heapq

# Define the size of the puzzle (3x3)
PUZZLE_SIZE = 3

# Define the possible moves (up, down, left, right) along with their corresponding directions
MOVES = [(0, 1, 'Right'), (0, -1, 'Left'), (1, 0, 'Down'), (-1, 0, 'Up')]

# Function to calculate the Manhattan distance heuristic
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(PUZZLE_SIZE):
        for j in range(PUZZLE_SIZE):
            if state[i][j] != 0:
                target_x, target_y = divmod(goal_state[i][j] - 1, PUZZLE_SIZE)
                distance += abs(i - target_x) + abs(j - target_y)
    return distance

# Function to check if a state is valid
def is_valid(x, y):
    return 0 <= x < PUZZLE_SIZE and 0 <= y < PUZZLE_SIZE

# Function to perform A* search to solve the puzzle and record the path
def solve_puzzle(start_state, goal_state):
    open_set = [(manhattan_distance(start_state, goal_state), start_state, '', 0)]
    visited = set()
    came_from = {}  # Dictionary to store the parent state for each state
    move_counter = 0  # Move counter

    while open_set:
        _, current_state, direction, _ = heapq.heappop(open_set)

        if tuple(map(tuple, current_state)) == tuple(map(tuple, goal_state)):
            # Backtrack from goal state to start state to construct the path
            path = [(current_state, direction, move_counter)]
            while tuple(map(tuple, current_state)) != tuple(map(tuple, start_state)):
                current_state, direction = came_from[tuple(map(tuple, current_state))]
                path.append((current_state, direction, move_counter))
            path.reverse()  # Reverse the path to get the correct order
            return path

        visited.add(tuple(map(tuple, current_state)))
        x, y = None, None

        # Find the position of the empty tile
```

```
        for i in range(PUZZLE_SIZE):
            for j in range(PUZZLE_SIZE):
                if current_state[i][j] == 0:
                    x, y = i, j

        for dx, dy, new_direction in MOVES:
            new_x, new_y = x + dx, y + dy
            if is_valid(new_x, new_y):
                new_state = [list(row) for row in current_state]
                new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[x][y]
                if tuple(map(tuple, new_state)) not in visited:
                    priority = manhattan_distance(new_state, goal_state)
                    heapq.heappush(open_set, (priority, new_state, new_direction, 0))
                    # Store the parent state and direction for the current state
                    came_from[tuple(map(tuple, new_state))] = (current_state, new_direction)
                    move_counter += 1

    return None

if __name__ == "__main__":
    print("Enter the starting state of the 8-puzzle:")
    start_state = []
    for _ in range(PUZZLE_SIZE):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state of the 8-puzzle:")
    goal_state = []
    for _ in range(PUZZLE_SIZE):
        row = list(map(int, input().split()))
        goal_state.append(row)

    solution_path = solve_puzzle(start_state, goal_state)

    if solution_path:
        print("\nSolution found! Path to reach the goal state:")
        for move_number, (state, direction, _) in enumerate(solution_path):
            print("Move number:", move_number)
            print("Direction:", direction)
            for row in state:
                print(" ".join(map(str, row)))
    else:
        print("\nNo solution found.")
```

```
n=int(input("enter the disks"))
a=[i for i in range(1,n+1)]
b=[]
c=[]
def display(n,from_rod,to_rod):
        if from_rod=='A' and to_rod=='B':
            b.append(n);
            a.remove(n);
        elif from_rod=='B' and to_rod=='C':
            c.append(n);
            b.remove(n);
        elif from_rod=='A' and to_rod=='C':
            c.append(n);
            a.remove(n);
        elif from_rod=='B' and to_rod=='A':
            a.append(n);
            b.remove(n);
        elif from_rod=='C' and to_rod=='B':
            b.append(n);
            c.remove(n);
        elif from_rod=='C' and to_rod=='A':
            a.append(n);
            c.remove(n);
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print("Move disk 1 from rod",from_rod,"to rod",to_rod)
        display(1,from_rod,to_rod);
        print('A:',a,'B:',b,'C:',c);
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk",n,"from rod",from_rod,"to rod",to_rod)
    display(n,from_rod,to_rod)
    print('A:',a,'B:',b,'C:',c);
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)
    #n =int(input("enter the no.of disks"))

TowerOfHanoi(n, 'A', 'C', 'B')
```

```
import itertools

def tsp(graph, start_node):
    # Generate all possible permutations of nodes
    nodes = list(graph.keys())
    nodes.remove(start_node)
    permutations = list(itertools.permutations(nodes))

    # Initialize variables for best path and cost
    best_path = None
    best_cost = float('inf')

    # Initialize variables for other paths and costs
    other_paths = []
    other_costs = []

    # Iterate through all permutations and calculate cost
    for perm in permutations:
        current_path = [start_node] + list(perm) + [start_node]
        current_cost = 0

        for i in range(len(current_path) - 1):
            current_cost += graph[current_path[i]][current_path[i+1]]

        # Update best path and cost if current path is better
        if current_cost < best_cost:
            best_path = current_path
            best_cost = current_cost

        # Add current path and cost to other paths if it's not the best path
        elif current_cost != best_cost:
            other_paths.append(current_path)
            other_costs.append(current_cost)

    return best_path, best_cost, other_paths, other_costs

# Function to get user input for graph details
def get_graph_details():
    graph = {}
    num_nodes = int(input("Enter the total number of nodes: "))

    for i in range(num_nodes):
        node_name = input(f"Enter the name of node {i+1}: ")
        num_adjacent_nodes = int(input(f"Enter the number of adjacent nodes for
{node_name}: "))
```

```
        adj_nodes = {}
        for j in range(num_adjacent_nodes):
            adj_node_input = input(f"Enter the name of adjacent node {j+1} and its weight 
(separated by space): ")
            adj_node_data = adj_node_input.split()

            if len(adj_node_data) != 2:
                print("Invalid input. Please enter the adjacent node and its weight separated by a 
space.")
                return get_graph_details()

            adj_node_name, weight = adj_node_data
            adj_nodes[adj_node_name] = int(weight)
        graph[node_name] = adj_nodes

    return graph

# Function to get user input for starting node
def get_starting_node():
    starting_node = input("Enter the starting node: ")
    return starting_node


# Main function to execute the program
def main():
    graph = get_graph_details()
    start_node = get_starting_node()
    best_path, best_cost, other_paths, other_costs = tsp(graph, start_node)
    print("Best Path:", best_path)
    print("Total Cost:", best_cost)

    print("\nOther Paths:")
    for i in range(len(other_paths)):
        print(f"Path {i+1}: {other_paths[i]}")
        print(f"Total Cost: {other_costs[i]}")
        print()


# Execute the main function
main()
```

```
def is_safe(board, row, col, N):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper-left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check upper-right diagonal
    for i, j in zip(range(row, -1, -1), range(col, N)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens_util(board, row, N):
    if row == N:
        return True

    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1
            if solve_n_queens_util(board, row + 1, N):
                return True
            board[row][col] = 0

    return False

def solve_n_queens(N):
    board = [[0 for _ in range(N)] for _ in range(N)]

    if not solve_n_queens_util(board, 0, N):
        print("Solution does not exist")
        return

    for row in board:
        print(" ".join(["Q" if cell == 1 else "." for cell in row]))

if __name__ == "__main__":
    N = int(input("Enter the number of queens (N): "))
    solve_n_queens(N)
```

```
from collections import deque
def generatenextstates(state, m, n):
    m_left, c_left, boat, m_right, c_right = state
    moves = [(0, 1), (0, 2), (1,0), (2, 0), (1, 1)]
    possible_states = []

    for i in moves:
        if boat == 1:
            move_str = str(i[0]) + "M " + str(i[1]) + "C move from left to right"
            new_state = (m_left - i[0], c_left - i[1], 0, m_right + i[0], c_right + i[1])
        else:
            move_str = str(i[0]) + "M " + str(i[1]) + "C  move from right to left"
            new_state = (m_left + i[0], c_left + i[1], 1, m_right - i[0], c_right - i[1])

        if checkvalidstate(new_state, m, n):
            possible_states.append((new_state, move_str))

    return possible_states
def checkvalidstate(state, m, n):
    m_left, c_left, boat, m_right, c_right = state

    if (
        0 <= m_left <= m and
        0 <= c_left <= n and
        0 <= m_right <= m and
        0 <= c_right <= n and
        (m_left == 0 or m_left >= c_left) and
        (m_right == 0 or m_right >= c_right)):
        return True

    return False
def bfs(m, n):
    initial_state = (m, n, 1, 0, 0)
    visited = set()
    queue = deque()
    queue.append((initial_state, []))

    while queue:
        current_state, path = queue.popleft()
        visited.add(current_state)

        if checkfinalstate(current_state, m):
            return path
```

```
        for next_state, move_description in generatenextstates(current_state, m, n):
            if next_state not in visited:
                queue.append((next_state, path + [(next_state, move_description)]))
def checkfinalstate(state, m):
    return state == (0, 0, 0, m, m)
def  path(s):
    print("Initial state:")
    print("Boat positioned:at left")
    print("left side of river:",m,"M",n,"C")
    print("right side of river:",0,"M",0,"C")
    for i in range(len(s)):
        state, move_description = s[i]
        m_left, c_left, boat, m_right, c_right = state
        if boat == 1:
            boat_position = "at left bank"
        else:
            boat_position="at right bank"
        print("Step", i + 1,":")
        print(move_description)
        print(" Left side of river:", m_left, "M", c_left, "C" )
        print("Right side of river:", m_right, "M", c_right, "C ")
        print(" Boat positioned:", boat_position)
m = int(input("Enter the number of missionaries (m): "))
n = int(input("Enter the number of cannibals (n): "))
solution = bfs(m, n)
if solution:
    print("Solution found!")
    path(solution)
else:
    print("No solution found")
```