Prog1

```
def isValid(state):
    if (state[0] > 3 or state[1] > 3 or state[1]
< 0 or state[0] < 0 or
    (state[0] != 0 and state[0] < state[1]) or
    (3-state[0] != 0 and 3-state[0] < (3-
state[1]))):
        return False
    return True


def nextState(M, C, B):
    global ns
    print(f"########################## Next state
{ns} start ##########################")
    moves = [[1, 1], [1, 0], [0, 1], [0, 2], [2,
0]]
    valid_states = []
    for each in moves:
        print("loop start")
        if B == 1:
            print("B=1")
            next_state = [M+each[0], C+each[1],
0]
            print(f"next_state {next_state}")
        else:
            print("B!=1")
            next_state = [M-each[0], C-each[1],
1]
            print(f"next_state {next_state}")
```

```
        if isValid(next_state):
            print("isValid")
            valid_states.append(next_state)
        print("Loop end")
    print(f"######################### next state
{ns} end #########################")
    ns+=1
    return valid_states



def findSolutions(M, C, B, visited_paths,
solutions):
    global fs
    print(f"######################### findSolutions
{fs} start #########################")
    if [M, C, B] == [0, 0, 1]:
        print("found")
        solutions.append(visited_paths+[[0, 0,
1]])
        return True
    elif [M, C, B] in visited_paths:
        print(f"duplicate {[M,C,B]}")
        return False
    else:
        print(f"state found {[M,C,B]}")
        visited_paths.append([M, C, B])
        for current_state in nextState(M, C, B):
            print("loop to movgen from
current_state")
```

```
            print(f"current_state[0],
current_state[1], current_state[2],
visited_paths[:], solutions {current_state[0],
current_state[1], current_state[2],
visited_paths[:], solutions}")
            findSolutions(current_state[0],
current_state[1],
                          current_state[2],
visited_paths[:], solutions)
    print(f"########################## findSolutions
{fs} end ##########################")
    fs+=1


def mainProgram():
    solutions = []
    findSolutions(3, 3, 0, [], solutions)
    for soln in solutions:
        print("Solutions found: ")
        for i in range(len(soln)):
            print(soln[i], end="")
            if i != len(soln)-1:
                print(" -> ", end="")
        print("\n")

fs = 1
ns=1
mainProgram()
```

Prog2

```
def moveGen(node, A, B):
    res = []
    x, y = node
    dx, dy = A - x, B - y
    if x < A:
        res.append([A, y])
    if y < B:
        res.append([x, B])
    if x > 0:
        res.append([0, y])
    if y > 0:
        res.append([x, 0])
    if x > 0 and x + y <= B:
        res.append([0, x+y])
    if x > 0 and x+y > B:
        res.append([x-dy, B])
    if y > 0 and x+y <= A:
        res.append([x+y, 0])
    if y > 0 and x+y > A:
        res.append([A, y-dx])
    return res

def removeSeen(children, openlist, closedlist):
    open_heads_list = [i[0] for i in openlist]
    closed_heads_list = [i[0] for i in
closedlist]
    return [i for i in children if i not in
open_heads_list + closed_heads_list]

def makePairs(childrenlist, parent):
```

```
    return [[child, parent] for child in
childrenlist]

def findLink(node, closedlist):
    return list(filter(lambda x:x[0] == node,
closedlist))[0]

def reconstructPath(nodepair, closedlist):
    node, parent = nodepair
    res = [node]
    while parent is not None:
        node, parent = findLink(parent,
closedlist)
        res.append(node)
    res.reverse()
    return res

def mainProgram():
    A = int(input("Enter the capacity of jug 1:
"))
    B = int(input("Enter the capacity of jug 2:
"))
    start = [0, 0]
    goal = [0, 0]
    goal[0] = int(input("Enter the goal quantity
of jug 1: "))
    goal[1] = int(input("Enter the goal quantity
of jug 2: "))
    open = [[start, None]]
    closed = []
    solnExists = False
```

```
    while open:
        nodepair = open.pop(0)
        node = nodepair[0]
        if node == goal:
            if not solnExists:
                print("\nSolution found!")
                solnExists = True
            print("Goal state can be reached
through path: ", end="")
            print(reconstructPath(nodepair,
closed))
            print()
        else:
            closed = [nodepair] + closed
            children = moveGen(node, A, B)
            noLoops = removeSeen(children, open,
closed)
            new = makePairs(noLoops, node)
            open = new + open
    if not solnExists:
        print("No solutions found!")

mainProgram()
```

Prog3

```
def moveGen(node):
    print("$$$$$$$$$$$$$$$$$ movegen start
$$$$$$$$$$$$$$$$$$$$$$$$$")
    print(f"node {node}")
```

```
    res = []
    empty_pos = []
    for i in range(3):
        for j in range(3):
            if node[i][j] == -1:
                empty_pos = [i, j]
                break
    print(f"empty_pos {empty_pos}")
    neighbours = []
    for i in [1, -1]:
        neighbours.append([empty_pos[0]+i,
empty_pos[1]])
        neighbours.append([empty_pos[0],
empty_pos[1]+i])
    print(f"neighbours {neighbours}")
    neighbours = list(filter(lambda x: x[0] in
range(0,3) and x[1] in range(0,3), neighbours))
    print(f"neighbours {neighbours}")
    print("********* for loop start
***************")
    for neighbour in neighbours:
        print(f"neighbour {neighbour}")
        move = []
        print(f"move {move}")
        for i in node:
            move.append(i[:])
            print(f"inner loop move {move}")
        # swap neighbour and empty pos
        move[neighbour[0]][neighbour[1]],
move[empty_pos[0]][empty_pos[1]] =
```

```
move[empty_pos[0]][empty_pos[1]],
move[neighbour[0]][neighbour[1]]
        print(f"move[neighbour[0]][neighbour[1]],
move[empty_pos[0]][empty_pos[1]]
{move[neighbour[0]][neighbour[1]],
move[empty_pos[0]][empty_pos[1]]}")
        res.append(move)
    print("********* for loop end
***************")
    print(f"res {res}")
    print("$$$$$$$$$$$$$$$$$ movegen end
$$$$$$$$$$$$$$$$$$$$$$$$$")
    return res

def removeSeen(children, openlist, closedlist):
    print("Removing children")
    open_heads_list = [i[0] for i in openlist]
    closed_heads_list = [i[0] for i in
closedlist]
    print(f"open {open_heads_list}")
    print(f"closed {closed_heads_list}")
    return [i for i in children if i not in
open_heads_list+closed_heads_list]

def makePairs(childrenlist, parent):
    # print("make pairs")
    # for child in childrenlist:
    #     print(child)
    return [[child, h(child), parent] for child
in childrenlist]
```

```
def findLink(node, closedlist):
    return list(filter(lambda x: x[0] == node,
closedlist))[0]

def reconstructPath(nodepair, closedlist):
    node = nodepair[0]
    parent = nodepair[-1]
    res = [node]
    while parent is not None:
        node = findLink(parent, closedlist)[0]
        parent = findLink(parent, closedlist)[-1]
        res.append(node)
    res.reverse()
    return res

def h(board):
    print("h value")
    res = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] == goal[i][j]:
                res += 1
    print(f"board {board} res {res}")
    return res

def show_board(board):
    print("********************")
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] != -1:
                print(board[i][j],end="\t")
```

```
            else:
                print("X",end="\t")
        print()
    print("********************")

goal = [[1,2,3],[8,-1,4],[7,6,5]]
def mainProgram():
    print("Enter a 3x3 initial board:")
    start = [[1,2,3],[7,8,4],[6,-1,5]]
    # for i in range(3):
    #     tempList = []
    #     for j in range(3):
    #         tempList.append(int(input("Enter
val: ")))
    #     start.append(tempList)

    print("Enter a 3x3 goal board:")
    # for i in range(3):
    #     tempList = []
    #     for j in range(3):
    #         tempList.append(int(input("Enter
val: ")))
    #     goal.append(tempList)

    solnExists = False
    open = [[start, h(start), None]]
    closed = []
    while open:
        nodepair = open.pop(0)
        node = nodepair[0]
        if node == goal:
```

```
            if not solnExists:
                print("\nSolution found!")
                solnExists = True
            print("Goal state can be reached
through path: ")
            path =
reconstructPath(nodepair,closed)
            step = 0
            for board in path:
                if step == 0:
                    print("\nInitially")
                elif step == len(path)-1:
                    print("\nFinally, Step ",
step)
                else:
                    print("\nStep ", step)
                step += 1
                show_board(board)
            print()
            break
        else:
            closed = [nodepair]+closed
            children = moveGen(node)
            noLoops = removeSeen(children, open,
closed)
            new = makePairs(noLoops, node)
            open = sorted(new+open, key=lambda
x:x[1], reverse=True)
    if not solnExists:
        print("No solution found!")
```

```
mainProgram()
```

Prog4

```
def h(input):
    i=0 # to exit
    count = 0
    # print(f"$$$$$$$$$$$$$$$$$$ h start
$$$$$$$$")
    # print(f"input {input} \n count {count}")
    for q_row in range(len(input)):
        q_col = input[q_row]
        # print(f"q_col {q_col}")
        for q2_row in range(len(input)):
            q2_col = input[q2_row]
            # print(f"q2_col {q2_col}")
            if q2_row == q_row:
                # print(f"q2_row == q_row
{q2_row} == {q_row}")
                continue
            if q2_row - q2_col == q_row - q_col:
                # print(f"q2_row - q2_col ==
q_row - q_col {q2_row} - {q2_col} == {q_row} -
{q_col}")
                count += 1
            if q2_row + q2_col == q_row + q_col:
                # print(f"q2_row + q2_col ==
q_row + q_col {q2_row} + {q2_col} == {q_row} +
{q_col}")
                count += 1
```

```
        # if i == 2:
            # exit()
        # i+=1
    # print(f"$$$$$$$$$$$$$$$$$$ h end
$$$$$$$$")
    # print(f"count {count}")

    return count

def moveGen(input, open, closed, N):
    res = []
    # print("$$$$$$$$$$ moveGen start
$$$$$$$$$$$$")
    for i in range(1, N):
        child = input[:]
        # print(f"child {child}")
        child[0], child[i] = child[i], child[0]
        # print(f"child[0] {child[0]}  child[{i}]
{child[i]}")
        if child not in open and child not in
closed:
            # print(f"append child {child}")
            res.append(child)
    # print("$$$$$$$$$$ moveGen end
$$$$$$$$$$$$")
    return res

def isSoln(board, N):
    diag_collns = h(board) > 0
    two_q_same_col = len(set(board)) < N
    return not (diag_collns or two_q_same_col)
```

```
def show_board(board, N):
    final_board = [['-' for i in range(N)] for j
in range(N)]
    for i in range(len(board)):
        target = board[i]
        final_board[i][target] = 'Q'
    # print("*******"*N)
    for i in range(len(final_board)):
        for j in range(len(final_board)):
            print(final_board[i][j],end="\t")
        print()
    # print("*******"*N)

def mainProgram():
    N = int(input("Enter the no. of queens: "))
    start = [i for i in range(N)]
    open, closed = [start], []
    print("initial board:")
    show_board(start,N)
    solnExists = False
    i=0
    #
print("**********************************************
**********************************************
******")
    while open:
        # print(f"################ iteration {i}
start ##################")
        board = open.pop(0)
        # print(f"board {board}")
```

```
        if isSoln(board,N):
            print("solution found")
            print("\nfinal board:")
            show_board(board,N)
            solnExists = True
            break
        else:
            closed = [board]+closed
            # print(f"closed {closed}")
            children = moveGen(board, open,
closed, N)
            # print(f"children {children}")
            open = sorted(children+open,
key=lambda x: h(x))
            # print(f"open {open}")
        # print(f"################ iteration {i}
end ##################")
        i += 1
    if not solnExists:
        print("No Solution found!")
    print("**********************************************
**********************************************
**********")

mainProgram()
```

Prog5

```
knowledge_base={
    'flu':['muscle pain','cough','fever','runny
nose','sneezing','sore throat'],
```

```
    'tuberculosis':['chest
pain','cough','fever','fatigue'],
    'coronavirus':['loss of taste','loss of
smell','sore throat','cough','fever','shortness
of breath']
}
def inference(symptoms):
    probabilty={}
    for disease in knowledge_base.keys():
        count=0
        for symptom in knowledge_base[disease]:
            if symptom in symptoms:
                count+=1
        probabilty[disease]=count/len(knowledge_b
ase[disease])

    maxprobability=0
    for disease in probabilty.keys():
        if probabilty[disease]>maxprobability:
            maxprobability=probabilty[disease]

    diseases=''
    for disease in probabilty.keys():
        if probabilty[disease]==maxprobability:
            diseases+=disease+', '

    diseases=list(diseases)
    diseases[-2]='.'
    diseases=''.join(diseases)

    if maxprobability==1:
```

```
        print('You are having '+diseases)
    elif maxprobability>0:
        print('You may have '+diseases)
    else:
        print('You are not having any disease')

def askquestions():
    symptoms=[]
    questions=[]
    for disease in knowledge_base.keys():
        questions+=knowledge_base[disease]
    print(questions)
    questions=list(set(questions))
    print('Please answer the following questions:
')
    for question in questions:
        answer=input(f'Do you have {question} ?
[yes/no] : ' )
        if answer=='yes':
            symptoms.append(question)
    print('')
    return symptoms

symptoms=askquestions()
inference(symptoms)
```

Prog6

```
def empty_cells(board):
    count=0
    for i in range(3):
```

```
        for j in range(3):
            if board[i][j]!='x' or
board[i][j]!='o':
                count+=1
    return count

def return_index(board,position):
    for i in range(3):
        for j in range(3):
            if board[i][j]==position:
                return [i,j]
    return [-1,-1]

def displayboard(board):
    for i in range(3):
        print(' ',end='')
        for j in range(3):
            print(' ',board[i][j],end=' ')
            if j!=2:
                print('|',end='')
        print('')
        if i!=2:
            for j in range(18):
                print('-',end='')
            print('')


def evaluate(board):
    # print("$$$$$$$$$$$$$$$$$$ evaluate
$$$$$$$$$$$$$$$$$$$$")
```

```
    # print(f"first condition {[board[i][j]!='x'
and board[i][j]!='o' for i in range(3) for j in
range(3) ]}")
    if all([board[i][j]!='x' and board[i][j]!='o'
for i in range(3) for j in range(3) ]):
        return 0

    if ['x','x','x'] in board:
        return 10

    if ['o','o','o'] in board:
        return -10

    # print("$$$$$$$$$$$$$$$$ col start
$$$$$$$$$$$$")
    for col in range(3):
        if board[0][col]==board[1][col] and
board[1][col]==board[2][col]:
            #
print(f"board[0][{col}]==board[1][{col}] and
board[1][{col}]==board[2][{col}]  {board[0][col]=
=board[1][col] and
board[1][col]==board[2][col]}")
            if board[0][col]=='x':
                # print(f"board[0][{col}]=='x'
{board[0][col]=='x'}")
                return 10
            elif board[0][col]=='o':
                # print(f"board[0][{col}]=='o'
{board[0][col]=='o'}")
                return -10
```

```
    # print("$$$$$$$$$$$$$$$$ col end
$$$$$$$$$$$$")
    if (board[0][0]==board[1][1] and
board[1][1]==board[2][2]) or
(board[0][2]==board[1][1] and
board[1][1]==board[2][0]):
        if board[1][1]=='x':
            return 10
        elif board[1][1]=='o':
            return -10

    return 0

def game_over(board):
    score=evaluate(board)
    if score==10:
        print('Game Over!')
        print('')
        displayboard(board)
        print('')
        print('X Won!')
        return True
    elif score==-10:
        print('Game Over!')
        print('')
        displayboard(board)
        print('')
        print('O Won!')
        return True
```

```
    for i in range(3):
        for j in range(3):
            if board[i][j]!='x' and
board[i][j]!='o':
                return False
    return True

def minimax(board,depth,isMax):
    score=evaluate(board)
    if score==10 or score==-10:
        return score

    if game_over(board):
        return 0

    maxPlayer,minPlayer='x','o'
    if isMax:
        best_score=-1000
        for i in range(3):
            for j in range(3):
                if board[i][j]!='x' and
board[i][j]!='o':
                    original_value=board[i][j]
                    board[i][j]=maxPlayer
                    best_score=max(best_score,min
imax(board,depth+1,not isMax))
                    board[i][j]=original_value
        return best_score
    else:
        best_score=1000
        for i in range(3):
```

```
            for j in range(3):
                if board[i][j]!='x' and
board[i][j]!='o':
                    original_value=board[i][j]
                    board[i][j]=minPlayer
                    best_score=min(best_score,min
imax(board,depth+1,not isMax))
                    board[i][j]=original_value
        return best_score


def findBestMove(board,isMax):
    best=-1000 if isMax else 1000
    best_move=[-1,-1]
    for i in range(3):
        for j in range(3):
            if board[i][j]!='x' and
board[i][j]!='o':
                original_value=board[i][j]
                board[i][j]='x' if isMax else 'o'
                move_value=minimax(board,9-
empty_cells(board),not isMax)
                if (isMax and move_value>best) or
(not isMax and move_value<best):
                    best=move_value
                    best_move=[i,j]
                board[i][j]=original_value
    return best_move


game_board=[]
```

```
count=0
for i in range(3):
    row=[]
    for j in range(3):
        row.append(count+1)
        count+=1
    game_board.append(row)

player=input('Do you want to play x or o ? : ')
computer='x' if player=='o' else 'o'
isMax=True if player=='o' else False
turn='user' if player=='x' else 'computer'
print('')
displayboard(game_board)
print('')

while not game_over(game_board):
    if turn=='user':
        position=int(input(f'Enter position to
enter {player}: '))
        print('')
        positions=return_index(game_board,positio
n)
        if(positions==[-1,-1]):
            continue
        game_board[positions[0]][positions[1]]=pl
ayer

        turn='computer'
    elif turn=='computer':
        positions=findBestMove(game_board,isMax)
```

```
        game_board[positions[0]][positions[1]]=co
mputer
        turn='user'
    if turn=='user':
        print('Computer:')
    else:
        print('User:')
    displayboard(game_board)
    print('')

if(evaluate(game_board) not in [10,-10]):
    print('Game Over!')
    print('')
    displayboard(game_board)
    print('')
    print('Tie Game!')
```

Prog7

```
edges = {}
weights = {}
heuristic = {}
f = {}
g = {}
open = []
closed = []


def fsort(openlist):
```

```
    for p in range(len(openlist)):
        for i in range(len(openlist)-1):
            current = openlist[i]
            next = openlist[i+1]
            if f[current[0]] > f[next[0]]:
                temp = openlist[i+1]
                openlist[i+1] = openlist[i]
                openlist[i] = temp
    return openlist


def findparent(node):
    for s in closed:
        if s[0] == node:
            return s[1]
    return None


def ReconstructPath(st):
    node = st[0]
    parent = st[1]
    path = [node]
    while parent is not None:
        node = parent
        parent = findparent(node)
        path.append(node)
    path.reverse()
    return path


def removenode(li, node):
```

```
    for s in li:
        if s[0] == node:
            li.remove(s)
    return li


def checkifpresent(li, node):
    for s in li:
        if s[0] == node:
            return True
    return False


def propagateImprovement(node):
    neighbours = edges[node]
    for n in neighbours:
        if checkifpresent(open, n):
            edge = n+node
            new_g = g[node]+weights[edge]
            if new_g < g[n]:
                g[n] = new_g
                f[n] = g[n]+heuristic[n]
                open = removenode(open, n)
                open.append([n, node, g[n],
heuristic[n]])
        if checkifpresent(closed, n):
            edge = n+node
            new_g = g[node]+weights[edge]
            if new_g < g[n]:
                g[n] = new_g
                f[n] = g[n]+heuristic[n]
```

```
                closed = removenode(closed, n)
                closed.append([n, node, g[n],
heuristic[n]])
                propagateImprovement(n)


n = int(input('Enter number of nodes: '))
print('Enter Node Names:')
nodes = []

for i in range(n):
    nodes.append(input(f'Node {i+1}: ').upper())
    heuristic[nodes[i]] = float(input('Enter
Heuristic Value: '))
    print('')
    edges[nodes[i]] = []

print(f"nodes {nodes}")
print(f"edges {edges}")

s = d = 'start'
count = 0
print('Enter Edges: ')
print('Note: Enter edge as end,end to stop
entering more edges\nEnter Node names in Capital
Letters')
print('If any Edge cost is infinity, please enter
the value as 99999')
print('')
while s.lower() != 'end' and d.lower() != 'end':
    print('Edge ', count+1, ':')
```

```
    s = input('Enter source node: ')
    d = input('Enter destination node: ')
    if s != 'end' and d != 'end':
        c = float(input('Enter cost of edge: '))
        edges[s].append(d)
        edges[d].append(s)
        weights[s+d] = weights[d+s] = c
    print('')
    count += 1

print('')

start = input('Enter Start Node: ')
goal = input('Enter Goal Node: ')
f[start] = heuristic[start]
g[start] = 0
state = [start, None, 0, heuristic[start]]
open.append(state)
print(f"weight {weights}")
print(f"edge {edges}")
i=1
while len(open) != 0:
    print(f"$$$$$$$$$$$$$$$$$$$$$$$$$$$$
iteration {i} start
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$")
    current_state = open.pop(0)
    print(f"current state: {current_state}")
    closed.append(current_state)
    if current_state[0] == goal:
        print('Solution Found!')
```

```
        print('Path: ',
ReconstructPath(current_state))
        print('Total Cost: ', current_state[2])
        break
    neighbours = edges[current_state[0]]
    print(f"neighbours {neighbours}")
    j=1
    for neighbour in neighbours:
        print(f"###################### loop {j}
start ##############################")
        if checkifpresent(open, neighbour) ==
False and checkifpresent(closed, neighbour) ==
False:
            print("open and closed false")
            edge = neighbour + current_state[0]
            print(f"edge {edge}")
            open.append([neighbour,
current_state[0], current_state[2]+weights[edge],
heuristic[neighbour]])
            print(f"[neighbour, current_state[0],
current_state[2]+weights[edge],
heuristic[neighbour]]  {[neighbour,
current_state[0], current_state[2]+weights[edge],
heuristic[neighbour]]}")
            g[neighbour] =
current_state[2]+weights[edge]
            print(f"g[neighbour] {g[neighbour]}")
            f[neighbour] =
g[neighbour]+heuristic[neighbour]
            print(f"f[neighbour] {f[neighbour]}")
        elif checkifpresent(open, neighbour):
```

```
            print("open true")
            edge = neighbour+current_state[0]
            print(f"edge {edge}")
            new_g =
current_state[2]+weights[edge]
            print(f"new_g {new_g}")
            if new_g < g[neighbour]:
                g[neighbour] = new_g
                print(f"g[neighbour]
{g[neighbour]}")
                f[neighbour] =
g[neighbour]+heuristic[neighbour]
                print(f"f[neighbour]
{f[neighbour]}")
                open = removenode(open,
neighbour)
                print(f"open {open}")
                open.append([neighbour,
current_state[0],
                             g[neighbour],
heuristic[neighbour]])
                print(f"open {open}")
        elif checkifpresent(closed, neighbour):
            print("closed true")
            edge = neighbour+current_state[0]
            print(f"edge {edge}")
            new_g =
current_state[2]+weights[edge]
            print(f"new_g {new_g}")
            if new_g < g[neighbour]:
                g[neighbour] = new_g
```

```
                print(f"g[neighbour]
{g[neighbour]}")
                f[neighbour] =
g[neighbour]+heuristic[neighbour]
                print(f"f[neighbour]
{f[neighbour]}")
                closed = removenode(closed,
neighbour)
                print(f"closed {closed}")
                closed.append([neighbour,
current_state[0],
                               g[neighbour],
heuristic[neighbour]])
                print(f"closed {closed}")
                propagateImprovement(neighbour)
        print(f"f {f}")
        print(f"g {g}")
        print(f"###################### loop {j} end
##############################")
        j+=1
    open = fsort(open)
    print(f"$$$$$$$$$$$$$$$$$$$$$$$$$$
iteration {i} end
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$")
    i+=1
```

Prog8

```
import itertools

def findInputs(ex):
    inputs=[]
```

```
    for c in ex:
        if c!='+' and c!='*' and c!='-' and
c!='(' and c!=')' and c!=' ' and c not in inputs:
            inputs.append(c)
    print(f"inputs {inputs}")
    return inputs

def evaluate(ex,row):
    input_no=0
    replaced=[]
    for c in ex:
        print(f"c {c}")
        if c!='+' and c!='*' and c!='-' and
c!='(' and c!=')' and c!=' ' and c not in
replaced:
            ex=ex.replace(c,str(row[input_no]))
            input_no+=1
            replaced.append(c)
        print(f"ex {ex}")
        print(f"input {input_no}")
    print(f"replaced {replaced}")
    ex=ex.replace('+','&')
    ex=ex.replace('*','|')
    ex=ex.replace('-','~')
    print(f"ex {ex}")
    result=eval(ex)
    return result

def compute_results(truthtable,ex):
    outputs=[]
    for row in truth_table:
```

```
        outputs.append(evaluate(ex,row))

    print(f"outputs {outputs}")
    return outputs

n=int(input('Enter number of inputs: '))
print('Rules: ')
print('1) To enter "AND" use "+"')
print('2) To enter "OR" use "*"')
print('3) To enter "NOT" use "-"')
logical_expression=input('Enter the logical
expression: ')

truth_table =
list(itertools.product([0,1],repeat=n))
print(truth_table)
table_outputs=compute_results(truth_table,logical
_expression)

print('')
print('Truth Table:')
print('')
for c in findInputs(logical_expression):
    print(c,end='  ')
print('Output')

for i in range(len(truth_table)):
    for val in truth_table[i]:
        print(val,end='  ')
    print('  ',table_outputs[i])
print('')
```

```
if table_outputs.count(1)>=1:
    print('The entered propositional logic
expression is Satisfiable')
else:
    print('The entered propositional logic
expression is Not Satisfiable')
```