B.M.S. COLLEGE OF ENGINEERING
BENGALURU Autonomous Institute,Affiliated to VTU

Lab Record

Artificial Intelligence

(22CS5PCAIN)

Bachelor of
Technology in
Computer Science and Engineering

Submitted by:

Shashank H S
1BM21CS198

Department of Computer Science and
Engineering B.M.S. College of
Engineering
Bull Temple Road, Basavanagudi, Bangalore
560 019 Mar-June 2021

B.M.S. COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

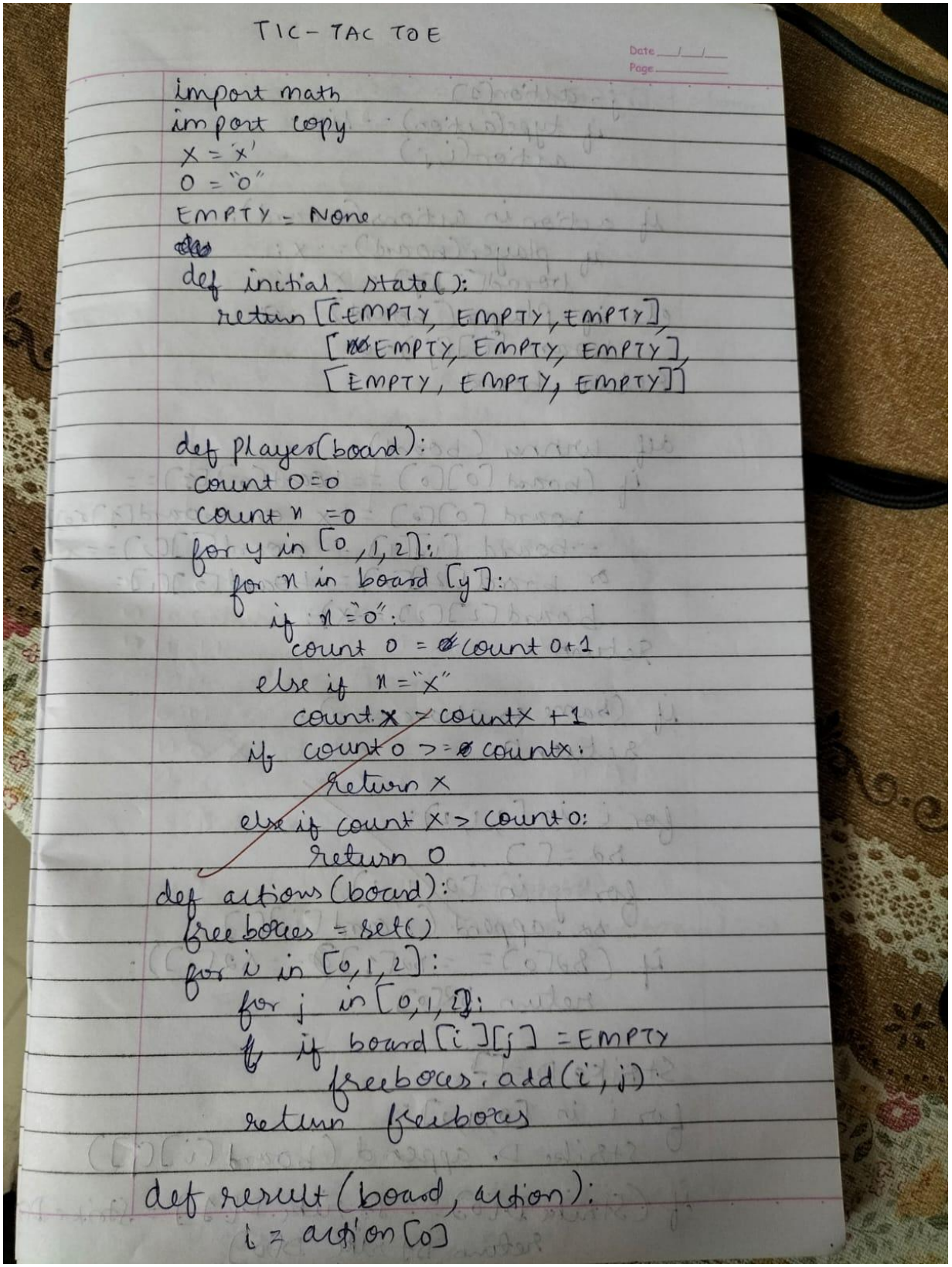
This is to certify that the Artificial Intelligence **(22CS5PCAIN) laboratory** has been carried out by **Shashank H S(1BM21CS198)** during the 5th Semester September-January 2021.

Signature of the Faculty In charge:

Dr. Pallavi G B
Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of Contents

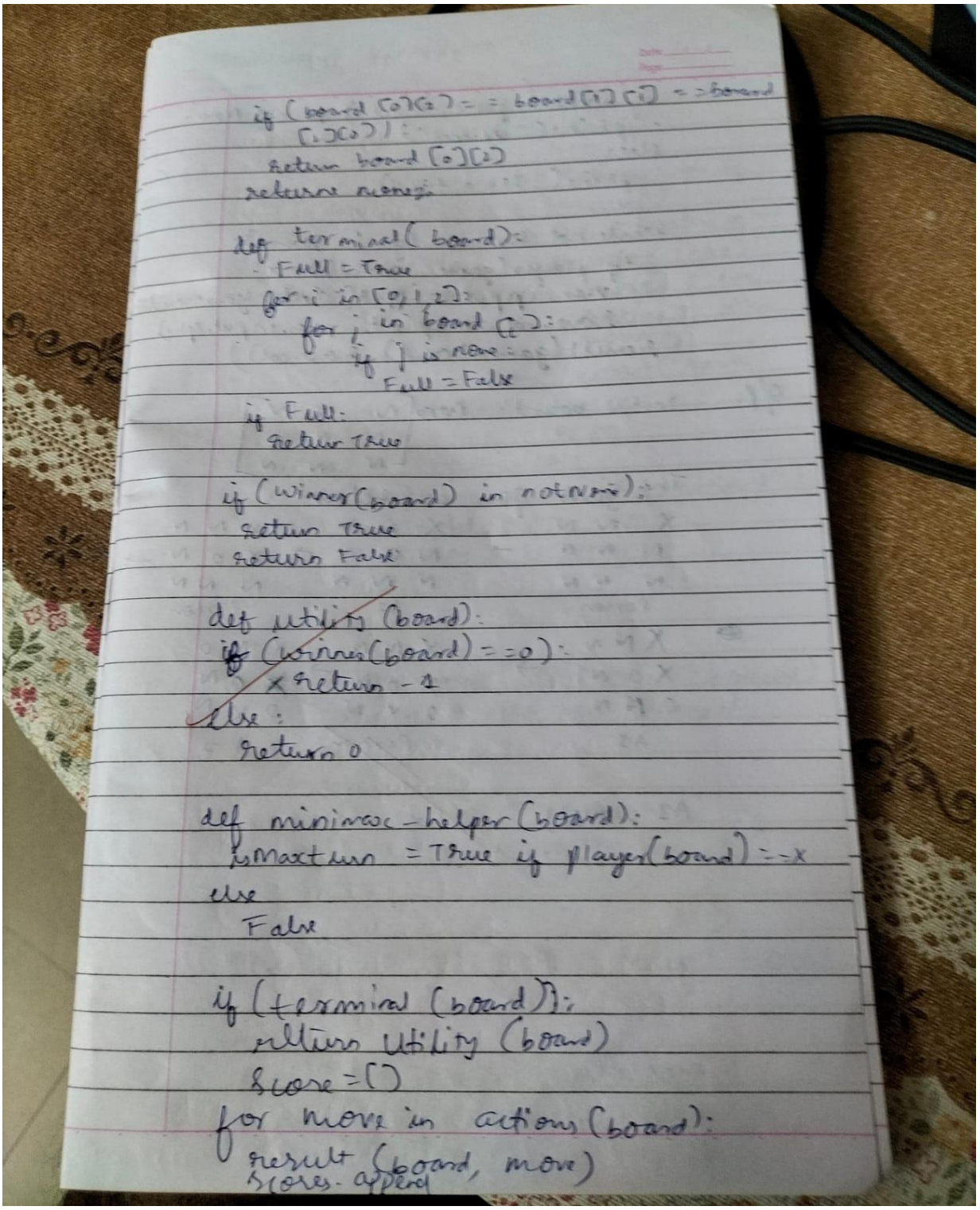
Lab-Program-1



TIC-TAC TOE

```
import math
import copy
X = 'x'
O = "O"
EMPTY = None

def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    countO=0
    countX =0
    for y in [0,1,2]:
        for x in board[y]:
            if x="O":
                countO = countO+1
            else if x="X"
                countX = countX +1
    if countO >=countX:
        return X
    else if countX > countO:
        return O
def actions(board):
    freeboxes = set()
    for i in [0,1,2]:
        for j in [0,1,2]:
            if board[i][j] =EMPTY
                freeboxes.add(i,j)
    return freeboxes

def result(board, action):
    i = action[0]
```

```
    j = action[1]
    if type(action) == list:
        action(i,j)

if action in actions(board):
    if player(board) == x:
        board[i][j] = x
else if player(board) == o:
    board[i][j] = o
Return board

def winner(board):
    if (board[0][0] == board[0][1] ==
        board[0][2] == x or board[1][0]
        == board[1][1] == board[1][2] == x
        or board[2][0] == board[2][1] ==
        board[2][2] == x):
    Return x

if (same as above == o):
    return 0

for i in [0,1,2]:
    s2 = []
    for j in [0,1,2]:
        s2.append(board[j][i])
    if (s2[0] == s2[1] == s2[2]):
        return s2[0]

StrikeD = []
for i in [0,1,2]:
    strikeD.append(board[i][j])
if (strikeD[0] == strikeD[1] = strikeD
        return strikeD[0]
```



```
if (board[0][2] == board[1][1] == board[2][0]):
    return board[0][2]
return none

def terminal(board):
    Full = True
    for i in [0, 1, 2]:
        for j in board[i]:
            if j is none:
                Full = False
    if Full:
        return True

if (winner(board) in notNone):
    return True
return False

def utility(board):
    if (winner(board) == 0):
        return 1
    else:
        return 0

def minimax-helper(board):
    isMaxTurn = True if player(board) == X
    else
        False

if (terminal(board)):
    return Utility(board)
    scores = []
for move in actions(board):
    result(board, move)
    scores.append
```

```
if winner(game board) is not None:
    print("Winner")
else:
    print("It's a tie!")

while not terminal(game_board):
    if player(game_board) == X:
        user_input = input("Enter your move")
        row, col = map(int, user_input.split)
        result(game_board), (row, col))
```

o/p:

Initial board:

N	N	N
N	N	N
N	N	N

The game goes as below:

X	N	N	X	N	N	X	N	N
N	N	N	→	N	O	N	→	X	O	N	→
N	N	N	N	N	W	N	N	N
Person			AI			Person		

X	N	N	X	N	N	X	N	O
X	O	N	→	X	O	N	→	X	O	N
O	N	N	O	X	N	O	X	N
AI			Person			AI		

AI WINS!!

Implement Tic-Tac-Toe Game

Objective: The objective of tic-tac-toe is that players have to position their marks so that they make a continuous line of three cells horizontally, vertically or diagonally**.**

Code:

```
board = [' ' for x in
range(10)] def
insertLetter(letter, pos):
    board[pos] = letter



def spaceIsFree(pos):
    return board[pos]
    == ' '



def
    printBoard(boar
    d): print('  |     |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' +
    board[3]) print('     |     |')
    print('_____')
    print('  |  |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' +
    board[6]) print('     |     |')
    print('_____')
    print('  |  |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
```

```
    print('   |   |')



def isWinner(bo, le):
    return (bo[7] == le and bo[8] == le and bo[9] == le) or (bo[4] == le and bo[5]
== le and bo[6] == le) or (
                bo[1] == le and bo[2] == le and bo[3] == le) or (bo[1] == le and bo[4]
== le and bo[7] == le) or (
                    bo[2] == le and bo[5] == le and bo[8] == le)
                    or ( bo[3] == le and bo[6] == le and bo[9] ==
                    le) or (
                    bo[1] == le and bo[5] == le and bo[9] == le) or (bo[3] == le and
bo[5] == le and bo[7] == le)



def
    playerMove()
    : run = True
    while run:
        move = input('Please select a position to place an \'X\' (1-9):
        ') try:
            move = int(move)
            if move > 0 and move <
                10: if
                spaceIsFree(move):
                    run = False
                    insertLetter('X',
                    move)
                else:
                    print('Sorry, this space is occupied!')
            else:
```

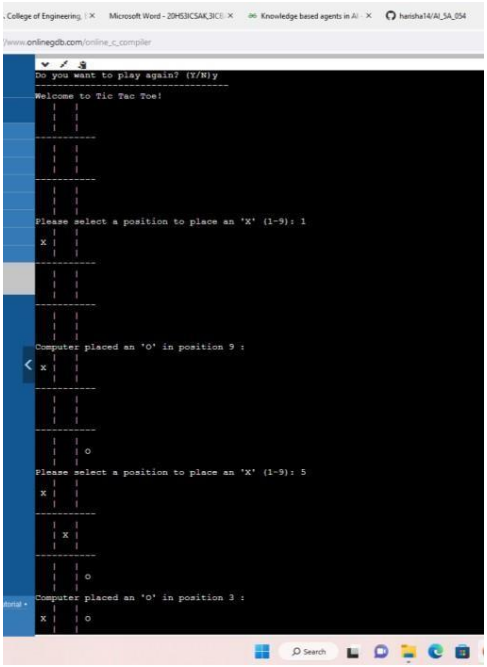
```
print('Please type a number!')


  def compMove():
    possibleMoves = [x for x, letter in enumerate(board) if letter == ' ' and x != 0]
    move = 0

    for let in ['O', 'X']:
      for i in
        possibleMoves:
        boardCopy =
        board[:]
        boardCopy[i] = let
        if isWinner(boardCopy,
          let): move = i
          return move

    cornersOpen = []
    for i in
      possibleMoves:
      if i in [1, 3, 7, 9]:
        cornersOpen.append(i)

    if len(cornersOpen) > 0:
      move =
      selectRandom(cornersOpen)
      return move

    if 5 in
      possibleMoves:
      move = 5
      return move
```

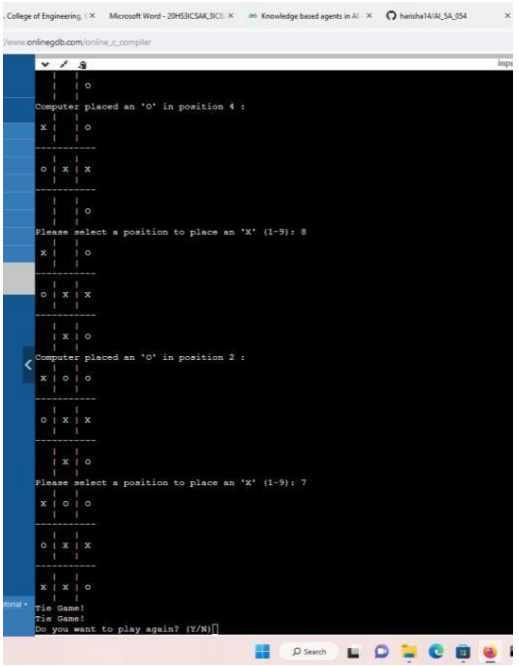
```
    edgesOpen = []
    for i in
        possibleMoves:
        if i in [2, 4, 6, 8]:
            edgesOpen.append(i)

    if len(edgesOpen) > 0:
        move =

    selectRandom(edgesOpen) return

    move


def
    selectRandom(li)
    : import random
    ln = len(li)
    r = random.randrange(0,
    ln) return li[r]


def
    isBoardFull(board)
    : if board.count('
    ') > 1:
        return
    False else:
        return True
```

```
while not
    (isBoardFull(board)): if
    not (isWinner(board, 'O')):
        playerMove()
        printBoard(board)
    else:
        print('Sorry, O\'s won this
        time!') break

    if not (isWinner(board,
        'X')): move =
        compMove()
        if move == 0:
            print('Tie
            Game!')
        else:
            insertLetter('O', move)
            print('Computer placed an \'O\' in position', move, ':')
            printBoard(board)
    else:
        print('X\'s won this time! Good
        Job!') break

if
    isBoardFull(boa
    rd): print('Tie
    Game!')
```

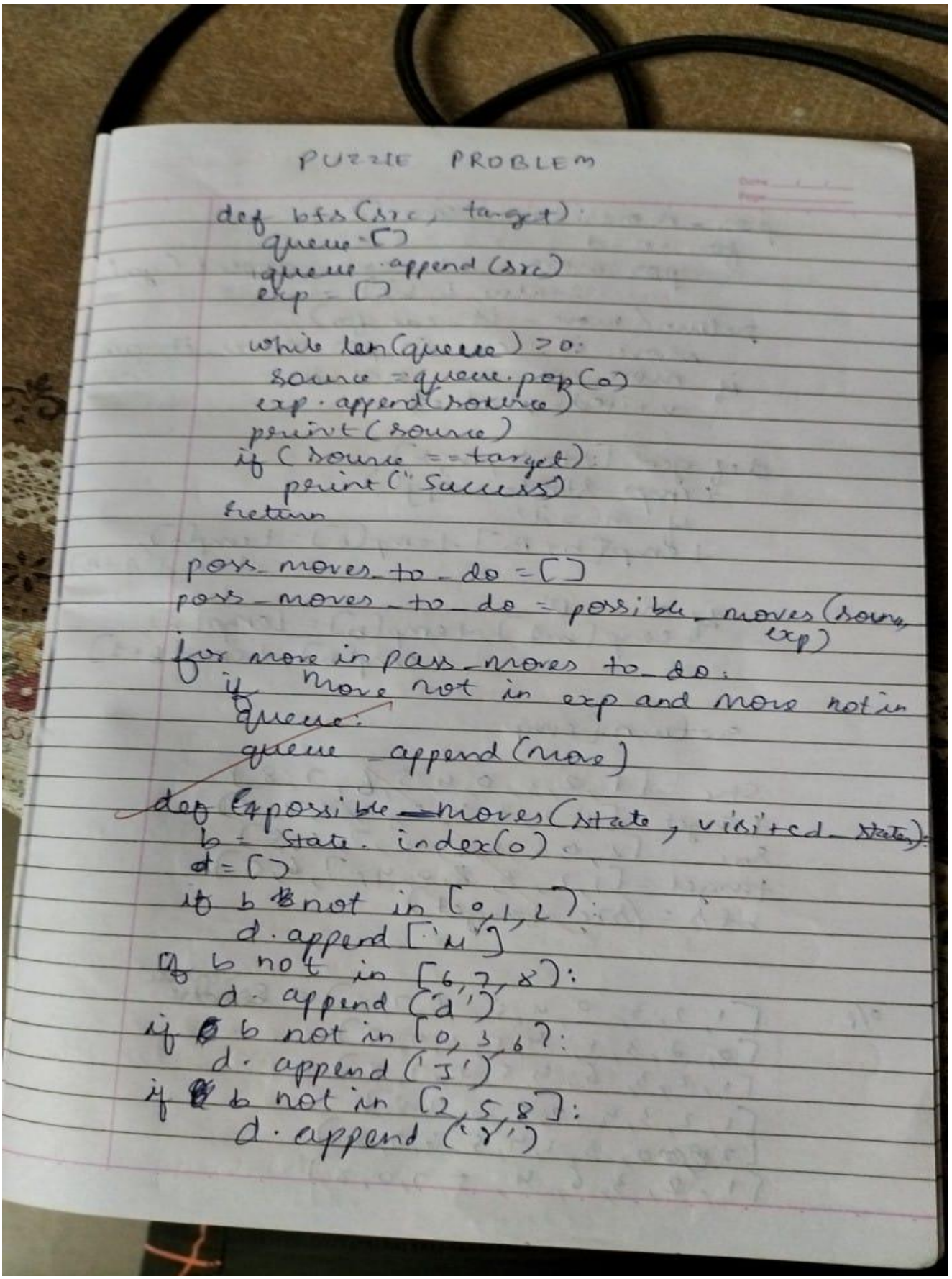
```
board = [' ' for x in range(10)] print('___   ') main()
        else:
            break;
```

Output:





Lab-Program-2



PUZZLE PROBLEM

```
def bfs(src, target):
    queue=[]
    queue.append(src)
    exp = []

    while len(queue)>0:
     source = queue.pop(0)
    exp.append(source)
    print(source)
    if (source == target):
      print("Success")
    return

poss_moves_to_do =[]
poss_moves_to_do = possible_moves(source,
                                  exp)
for move in poss_moves_to_do:
    if move not in exp and move not in
    queue:
    queue_append(move)

def possible_moves(state, visited_states):
    b = state.index(0)
    d=[]
    if b not in [0,1,2]:
      d.append['u']
    if b not in [6,7,8]:
      d.append('d')
    if b not in [0,3,6]:
      d.append('l')
    if b not in [2,5,8]:
      d.append('r')
```

```
pos_moves_it_can = []
for in d
    pos_move_it_can.append(ge
        state, i, b))
return(move_it_can for)
    move it_can in pos_moves_it_ca
if move_it_can not in
    visited_states
```

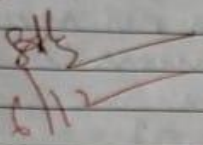
```
def gen [state, m, b]
    temp = state.copy()
    if m == 'd'
    temp[b+B] temp[b] = temp[b],
                        temp[A+
    if m == 'U'
    temp[b-1], temp[b] = temp[b]
                        temp[b], temp[b+1]

return temp
```

```
src = [1, 2, 3, 0, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5, 0, 6, 7, 8]
src = [2, 0, 3, 1, 8, 4, 7, 6, 5]
target = [1, 2, 3, 8, 0, 4, 7, 6, 5]
bfs = (src, target)
```

0/1:

```
[1, 2, 3, 0, 4, 5, 6, 7, 8] <- source
[0, 2, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 0, 7, 8]
[1, 2, 3, 4, 0, 5, 6, 7, 8]
[2, 0, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 7, 0, 8]
```

[1, 0, 3, 4, 2, 5, 6, 7, 8]
[1, 2, 3, 4, 7, 5, 6, 0, 8]
[1, 2, 3, 4, 5, 0, 6, 7, 8] ← ~~0~~ target
success



Solve 8 puzzle problem.

Objective: The objective of 8-puzzle problem is to reach the end state from the start state by considering all possible movements of the tiles without any heuristic.

Code:

```
import numpy as np
import os
class Node:
    def init (self, node_no, data, parent, act, cost):
        self.data = data self.parent
        = parent self.act = act
        self.node_no = node_no
        self.cost = cost
def get_initial():
    print("Please enter number from 0-8, no number should be
repeated or be out of this range")
    initial_state = np.zeros(9)
    for i in range(9):
        states = int(input("Enter the " + str(i + 1)
+ " number: "))
        if states < 0 or states > 8:
            print("Please only enter states which are [0-8], run
code again")
            exit(0)
        else:
            initial_state[i] = np.array(states)
    return np.reshape(initial_state, (3, 3))
def find_index(puzzle):
    i, j = np.where(puzzle == 0) i = int(i)
    j = int(j)
    return i, j
def move_left(data):
```

```
        i, j = find_index(data)
        if j == 0:
                return None
        else:
                temp_arr = np.copy(data) temp
                = temp_arr[i, j - 1] temp_arr[i, j]
                = temp temp_arr[i, j - 1] = 0
                return temp_arr
def move_right(data):
        i, j = find_index(data)
        if j == 2:
                return None
        else:
                temp_arr = np.copy(data) temp
                = temp_arr[i, j + 1] temp_arr[i, j]
                = temp temp_arr[i, j + 1] = 0
                return temp_arr
def move_up(data):
        i, j = find_index(data)
        if i == 0:
                return None
        else:
                temp_arr = np.copy(data) temp
                = temp_arr[i - 1, j] temp_arr[i, j]
                = temp temp_arr[i - 1, j] = 0
                return temp_arr
def move_down(data):
        i, j = find_index(data)
        if i == 2:
                return None
        else:
                temp_arr = np.copy(data) temp
                = temp_arr[i + 1, j] temp_arr[i, j]
                = temp temp_arr[i + 1, j] = 0
                return temp_arr
```

```
def move_tile(action, data):
        if action == 'up':
                return move_up(data)
        if action == 'down':
                return move_down(data)
        if action == 'left':
                return move_left(data)
        if action == 'right':
                return move_right(data)
        else:
                return None
def print_states(list_final): # To print the final states on the console
        print("printing final solution")
        for l in list_final:
                print("Move : " + str(l.act) + "\n" + "Result
: " + "\n" + str(l.data) + "\t" + "node number:" +
str(l.node_no))


def write_path(path_formed): # To write the final path in the text file
        if os.path.exists("Path_file.txt"):
                os.remove("Path_file.txt")

        f = open("Path_file.txt", "a")
        for node in path_formed:
                if node.parent is not None: f.write(str(node.node_no) + "\t" +
str(node.parent.node_no) + "\t" + str(node.cost) +
"\n")
        f.close()
def write_node_explored(explored): # To write all the nodes
explored by the program
        if os.path.exists("Nodes.txt"):
                os.remove("Nodes.txt")

        f = open("Nodes.txt", "a")
        for element in explored:
```

```
            f.write('[')
            for i in range(len(element)):
                for j in range(len(element)): f.write(str(element[j][i]) + "
                    ")
            f.write(']')
            f.write("\n")
        f.close()
def write_node_info(visited): # To write all the info about the
nodes explored by the program
        if os.path.exists("Node_info.txt"):
            os.remove("Node_info.txt")

        f = open("Node_info.txt", "a")
        for n in visited:
            if n.parent is not None: f.write(str(n.node_no) +
                "\t" +
str(n.parent.node_no) + "\t" + str(n.cost) + "\n") f.close()
def path(node): # To find the path from the goal node to the starting
node
        p = [] # Empty list
        p.append(node) parent_node =
        node.parent
        while parent_node is not None:
            p.append(parent_node) parent_node =
            parent_node.parent
        return list(reversed(p))
def path(node): # To find the path from the goal node to the starting
node
        p = [] # Empty list
        p.append(node) parent_node =
        node.parent
        while parent_node is not None:
            p.append(parent_node) parent_node =
            parent_node.parent
        return list(reversed(p))
def path(node): # To find the path from the goal node to the starting
node
        p = [] # Empty list
```

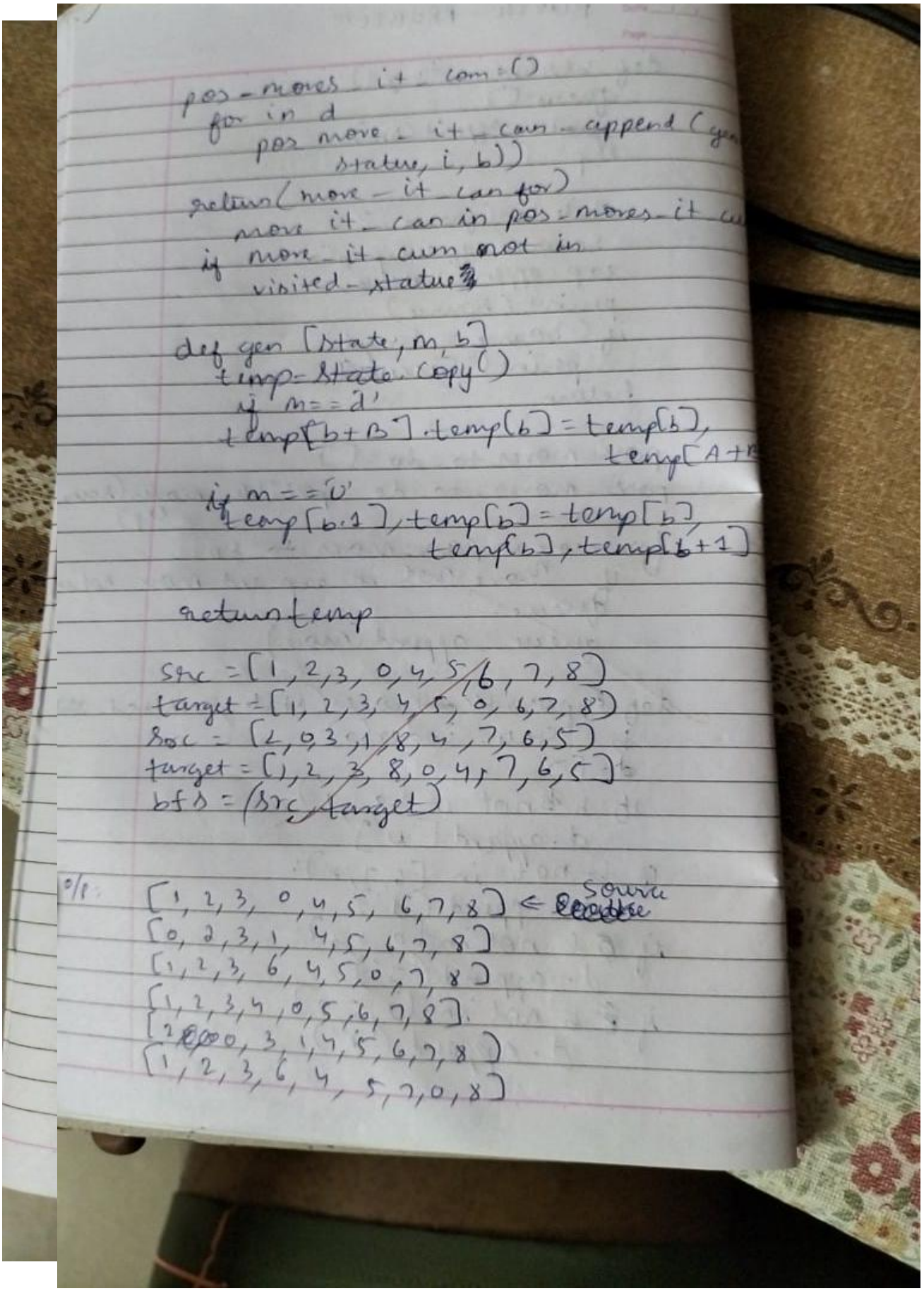
```
        p.append(node) parent_node =
        node.parent
        while parent_node is not None:
                p.append(parent_node) parent_node =
                parent_node.parent
        return list(reversed(p))
def check_correct_input(l): array =
        np.reshape(l, 9) for i in
        range(9):
                counter_appear = 0 f =
                array[i]
                for j in range(9):
                        if f == array[j]:
                                counter_appear += 1
                if counter_appear >= 2:
                        print("invalid input, same number entered
2 times")
                        exit(0)
def check_solvable(g):
        arr = np.reshape(g, 9)
        counter_states = 0 for i in
        range(9):
                if not arr[i] == 0: check_elem
                        = arr[i]
                        for x in range(i + 1, 9):
                                if check_elem < arr[x] or arr[x] ==
0:
                                        continue
                                else:
                                        counter_states += 1
        if counter_states % 2 == 0:
                print("The puzzle is solvable, generating
path")
        else:
                print("The puzzle is insolvable, still creating nodes")
k = get_initial() check_correct_input(k)
```

```
check_solvable(k)

root = Node(0, k, None, None, 0)

# BFS implementation call
goal, s, v = exploring_nodes(root)

if goal is None and s is None and v is None: print("Goal State could
        not be reached, Sorry")
  else:
        # Print and write the final output
        print_states(path(goal)) write_path(path(goal))
        write_node_explored(s) write_node_info(v)
```

Output:

```
Please enter number from 0-8, no number should be repeated or be out of this range
Enter the 1 number: 1
Enter the 2 number: 3
Enter the 3 number: 2
Enter the 4 number: 5
Enter the 5 number: 4
Enter the 6 number: 6
Enter the 7 number: 0
Enter the 8 number: 7
Enter the 9 number: 8
The puzzle is solvable, generating path
Exploring Nodes
Goal_reached
printing final solution
Move : None
Result :
[[1. 3. 2.]
 [5. 4. 6.]
 [0. 7. 8.]]    node number:0
Move : up
Result :
[[1. 3. 2.]
 [0. 4. 6.]
 [5. 7. 8.]]    node number:1
Move : right
Result :
[[1. 3. 2.]
 [4. 0. 6.]
 [5. 7. 8.]]    node number:5
```

Lab-Program-3



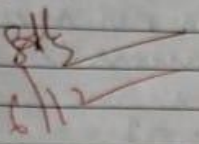
```
pos_moves_it_can = []
for i in d
    pos_moves_it_can.append(gen(state, i, b))
return(moves_it_can for moves_it_can in pos_moves_it_can if moves_it_can not in visited_states)

def gen[state, m, b]
    temp = state.copy()
    if m == 'd'
        temp[b+3], temp[b] = temp[b], temp[A+B]
    if m == 'u'
        temp[b-1], temp[b] = temp[b], temp[b], temp[b+1]
return temp

src = [1, 2, 3, 0, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5, 0, 6, 7, 8]
src = [2, 0, 3, 1, 8, 4, 7, 6, 5]
target = [1, 2, 3, 8, 0, 4, 7, 6, 5]
bfs = (src, target)
```

o/p:

```
[1, 2, 3, 0, 4, 5, 6, 7, 8] <- source
[0, 2, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 0, 7, 8]
[1, 2, 3, 4, 0, 5, 6, 7, 8]
[2, 0, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 7, 0, 8]
```

[1, 0, 3, 4, 2, 5, 6, 7, 8]
[1, 2, 3, 4, 7, 5, 6, 0, 8]
[1, 2, 3, 4, 5, 0, 6, 7, 8] ← ~~target~~ target

success



2 Implement Iterative deepening search algorithm.

Objective: IDDFS combines depth first search's space efficiency and breadth first search's completeness. It improves depth definition, heuristic and score of searching nodes so as to improve efficiency.

Code:

```
import copy
inp=[[1,2,3],[4,-1,5],[6,7,8]]
out=[[1,2,3],[6,4,5],[-1,7,8]]
def move(temp,
 movement): if
 movement=="up":
  for i in range(3):
   for j in range(3):
    if(temp[i][j]==
    -1): if i!=0:
      temp[i][j]=temp[i-
      1][j] temp[i-1][j]=-
      1
     return temp
 if
  movement=="dow
  n": for i in range(3):
   for j in range(3):
    if(temp[i][j]==
    -1): if i!=2:
      temp[i][j]=temp[i+1]
      [j] temp[i+1][j]=-1
     return temp
 if
  movement=="lef
  t": for i in
  range(3):
   for j in range(3):
    if(temp[i][j]==
    -1): if j!=0:
      temp[i][j]=temp[i][j
      -1] temp[i][j-1]=-1
     return temp
```

```
  if movement=="right":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==
        -1):
          if j!=2:
            temp[i][j]=temp[i][j
            +1] temp[i][j+1]=-1
          return
temp def ids():
  global
  inp
  global
  out
  global
  flag
  for limit in range(100):
    print('LIMIT ->
    '+str(limit)) stack=[]
    inpx=[inp,"none
    "]
    stack.append(in
    px) level=0
    while(True):
      if len(stack)==0:
        break
      puzzle=stack.pop
      (0) if
      level<=limit:
        print(str(puzzle[1])+" --> "+str(puzzle[0]))
        if(puzzle[0]==out):
          print("Found")
          print('Path
          cost='+str(level))
          flag=True
          ret
        urn
        else:
          level=level+1
          if(puzzle[1]!="down"):
```

```
        if(puzzle[1]!="right"):
          temp=copy.deepcopy(puzzle[0]
          ) left=move(temp, "left")
          if(left!=puzzle[0]):
          leftx=[left,"left"]
            stack.insert(0, leftx)
        if(puzzle[1]!="up"):
        temp=copy.deepcopy(puzzle[0])
        down=move(temp, "down")
        if(down!=puzzle[0]):
            downx=[down,"down"]
            stack.insert(0, downx)
        if(puzzle[1]!="left"):
          temp=copy.deepcopy(puzzle[0])
       right=move(temp,
          "right")
          if(right!=puzzle[0]):
          rightx=[right,"right"]
          stack.insert(0, rightx)
print('~~~~~~~~~~~~ IDS
~~~~~~~~~~~~') ids()


import copy
inp=[[1,2,3],[4,-1,5],[6,7,8]]
out=[[1,2,3],[6,4,5],[-1,7,8]]
def move(temp,
 movement): if
 movement=="up":
   for i in range(3):
     for j in
       range(3):
       if(temp[i][j]=
       =-1): if i!=0:
          temp[i][j]=temp[i
          -1][j] temp[i-
          1][j]=-1
        return temp
 if
   movement=="do
   wn": for i in
   range(3):
     for j in
       range(3):
       if(temp[i][j]=
```

=

```
            temp[i+1][j]
          =-1 return
          temp
  if movement=="left":
    for i in range(3):
      for j in
        range(3):
        if(temp[i][j]=
        =-1):
          if j!=0:
            temp[i][j]=temp[i]
            [j-1] temp[i][j-
            1]=-1
          return temp
  if movement=="right":
    for i in range(3):
      for j in
        range(3):
        if(temp[i][j]=
        =-1):
          if j!=2:
            temp[i][j]=temp[i]
            [j+1]
            temp[i][j+1]=-1
          return
temp def
ids():
  global
  inp
  global
  out
  global
  flag
  for limit in range(100):
    print('LIMIT ->
    '+str(limit)) stack=[]
    inpx=[inp,"no
    ne"]
    stack.append(i
    npx) level=0
    while(True):
      if len(stack)==0:
        break
      puzzle=stack.po
      p(0) if
      level<=limit:
        print(str(puzzle[1])+" -->
```

```
"+str(puzzle[0])) if(puzzle[0]==out):
  print("Found")
  print('Path
  cost='+str(level))
  flag=True
  re
tur
n
els
e:
  level=level+1
  if(puzzle[1]!="down"):
  temp=copy.deepcopy(puzzle[
  0])
```

```
            up=move(temp,
            "up")
            if(up!=puzzle[0]):
              upx=[up,"up"]
              stack.insert(0,
              upx)
          if(puzzle[1]!="right"):
            temp=copy.deepcopy(puzzl
            e[0]) left=move(temp, "left")
            if(left!=puzzle[0]):
            leftx=[left,"left"]
              stack.insert(0, leftx)
          if(puzzle[1]!="up"):
          temp=copy.deepcopy(puzzle[
          0]) down=move(temp,
          "down") if(down!=puzzle[0]):
              downx=[down,"do
              wn"]
              stack.insert(0,
              downx)
          if(puzzle[1]!="left"):
            temp=copy.deepcopy(puzzl
            e[0])
          right=move(temp,
            "right")
            if(right!=puzzle[0]):
            rightx=[right,"right
            "] stack.insert(0,
            rightx)
print('~~~~~~~~~~~~~ IDS
~~~~~~~~~~~~~') ids()
```

Output:

```
#Test 1
src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]

depth = 1
iddfs(src, target, depth)
```

```
False
```

```
#Test 2
src = [3,5,2,8,7,6,4,1,-1]
target = [-1,3,7,8,1,5,4,6,2]

depth = 1
iddfs(src, target, depth)
```

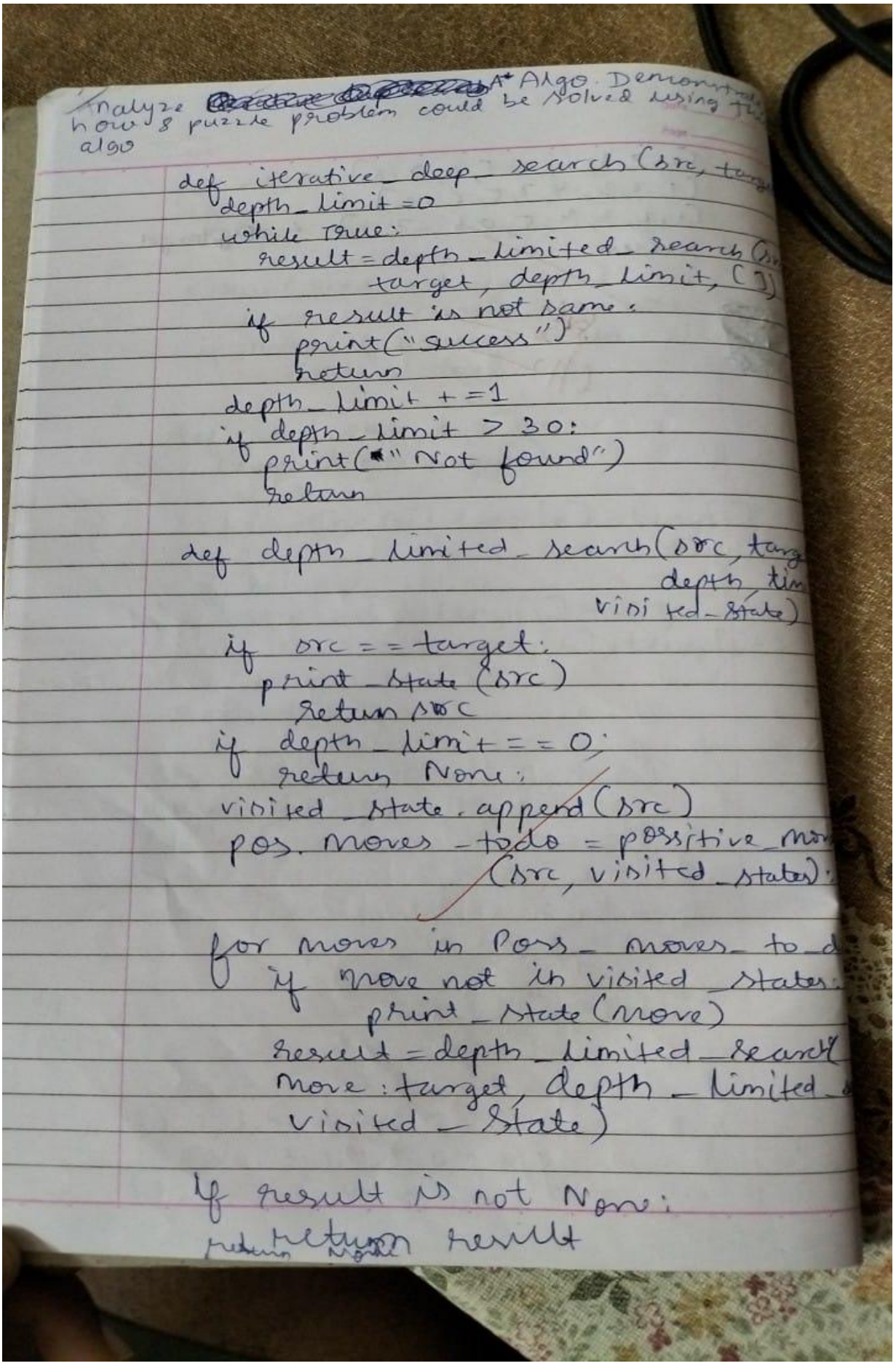
```
False
```

```
# Test 2
src = [1,2,3,-1,4,5,6,7,8]
target=[1,2,3,6,4,5,-1,7,8]

depth = 1
iddfs(src, target, depth)
```

```
src = [1, 2, 3, 4, 5, 6, 7, 8, -1]
target = [-1, 1, 2, 3, 4, 5, 6, 7, 8]

for i in range(1, 100):
    val = iddfs(src,target,i)
    print(i, val)
    if val == True:
        break
```

```
1 False
2 False
3 False
4 False
5 False
6 False
7 False
8 False
9 False
10 False
11 False
12 False
13 False
14 False
15 False
16 False
17 False
18 False
19 False
20 False
21 False
22 False
23 False
24 False
```

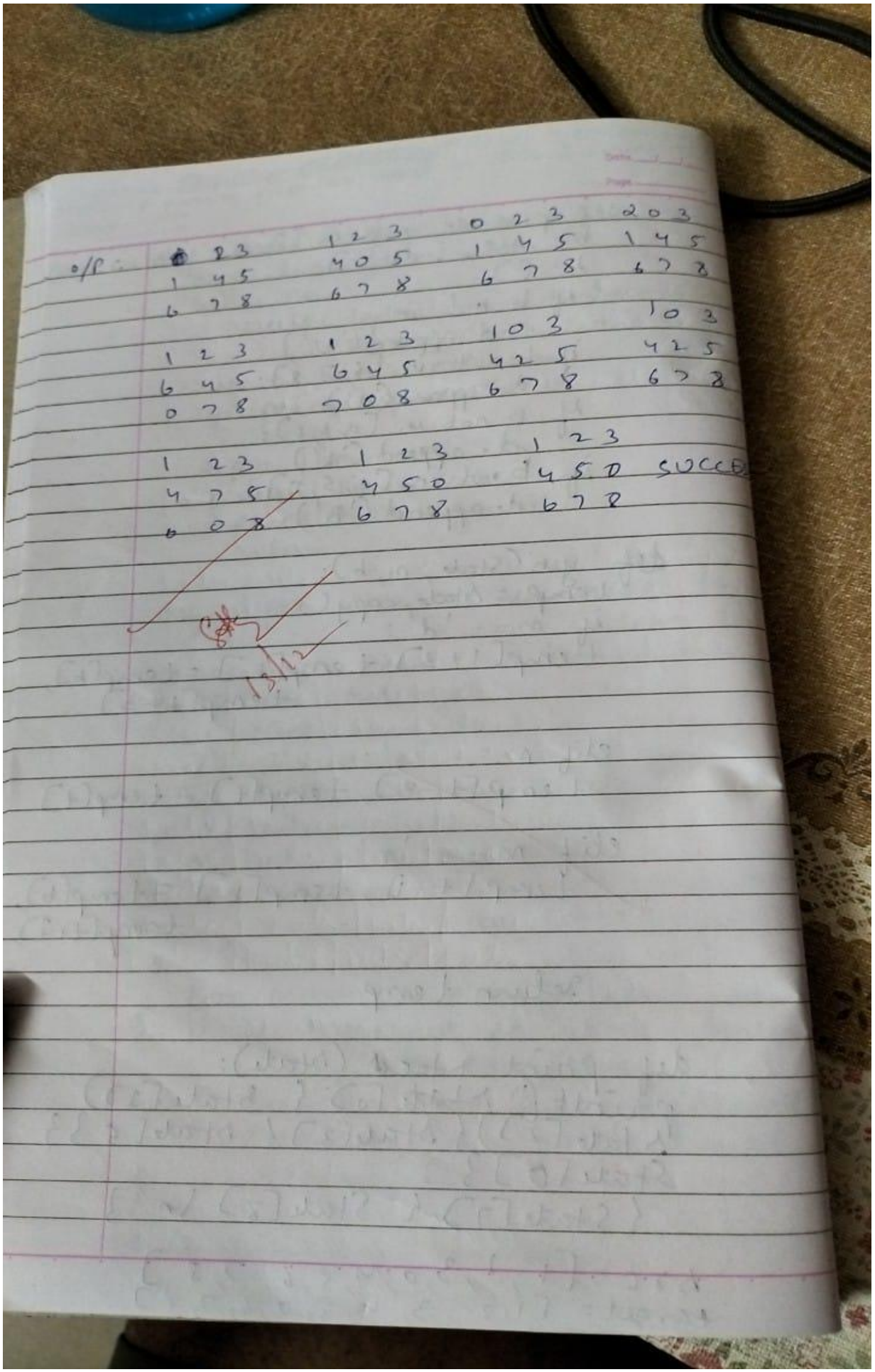
Lab
Program 4



Analyze how 8 puzzle problem could be solved using A* Algo. Demonstrate this algo

```
def iterative_deep_search(src, target):
    depth_limit = 0
    while True:
        result = depth_limited_search(src,
                 target, depth_limit, [])
        if result is not None:
            print("success")
            return
        depth_limit += 1
        if depth_limit > 30:
            print("Not found")
            return

def depth_limited_search(src, target,
                         depth_limit,
                         visited_state):
    if src == target:
        print_state(src)
        return src
    if depth_limit == 0:
        return None
    visited_state.append(src)
    pos_moves_todo = possitive_moves
                     (src, visited_states)

    for moves in Poss_moves_to_do:
        if move not in visited_states:
            print_state(move)
            result = depth_limited_search(
            move, target, depth_limited_
            visited_state)

    If result is not None:
        return result
```

```
def possible_moves(state, visited_states):
    b = state.index(0)
    d = []
    if b not in
        d.append('U')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]
        d.append('R')

def gen(state, m, b):
    temp = state.copy()
    if m == 'd':
    temp[i+3] temp[i] = temp[b],
                        temp[b+3]

    elif m == 'u':
        temp[t-3], temp[t] = temp[t]

    elif m == 'r':
        temp[t+1], temp[b] = temp[b],
                             temp[t+1],

    return temp

def print_stack(state):
    print(" state[0] { state[1])
    state[2]){ state[3] { state[5]}}
    State[6]}
      { State[7] { State[8] \n"}

src = [1, 2, 3, 0, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5, 0, 6, 7, 8]
```


o/p:
1 2 3
4 0 5
6 7 8
1 2 3
1 4 5
6 7 8
0 2 3
1 4 5
6 7 8
2 0 3
1 4 5
6 7 8
1 2 3
6 4 5
0 7 8
1 2 3
6 4 5
7 0 8
1 0 3
4 2 5
6 7 8
1 0 3
4 2 5
6 7 8
1 2 3
4 7 5
6 0 8
1 2 3
4 5 0
6 7 8
1 2 3
4 5 D
6 7 8
SUCCE
13/12

Implement A* search algorithm.

Objective: The a* algorithm takes into account both the cost to go to goal from present state as well the cost already taken to reach the present state. In 8 puzzle problem, both depth and number of misplaced tiles are considered to take decision about the next state that has to be visited.

Code:

```
def print_b(src):
    state =
    src.copy()
    state[state.index(-1)]
    = ' ' print(
f"""
{state[0]} {state[1]} {state[2]}
{state[3]} {state[4]} {state[5]}
{state[6]} {state[7]}
 {state[8]} “”””
    )
def h(state,
    target): count
    = 0
    i = 0
    for j in state:
        if state[i] !=
            target[i]: count
            = count+1
    return count
def astar(state,
    target): states =
    [src]
    g = 0
    visited_states
    = [] while
    len(states):
        print(f"Level:
        {g}") moves = []
        for state in states:
            visited_states.append(st
            ate) print_b(state)
            if state == target:
```

```
            print("Succes
            s") return
        moves += [move for move in
            possible_moves( state, visited_states) if
            move not in moves]
    costs = [g + h(move, target) for move in
    moves] states = [moves[i]
            for i in range(len(moves)) if costs[i] ==
    min(costs)] g += 1
print("Fail")
def possible_moves(state,
    visited_state): b = state.index(-1)
    d = []
    if b - 3 in
        range(9):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b + 3 in
        range(9):
        d.append('d')
    pos_moves
    = [] for m in
    d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if move not in
visited_state] def gen(state, m, b):
    temp =
    state.copy() if m
    == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b
    - 3] if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b
    - 1] if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b
    + 1] if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b
    + 3] return temp
```

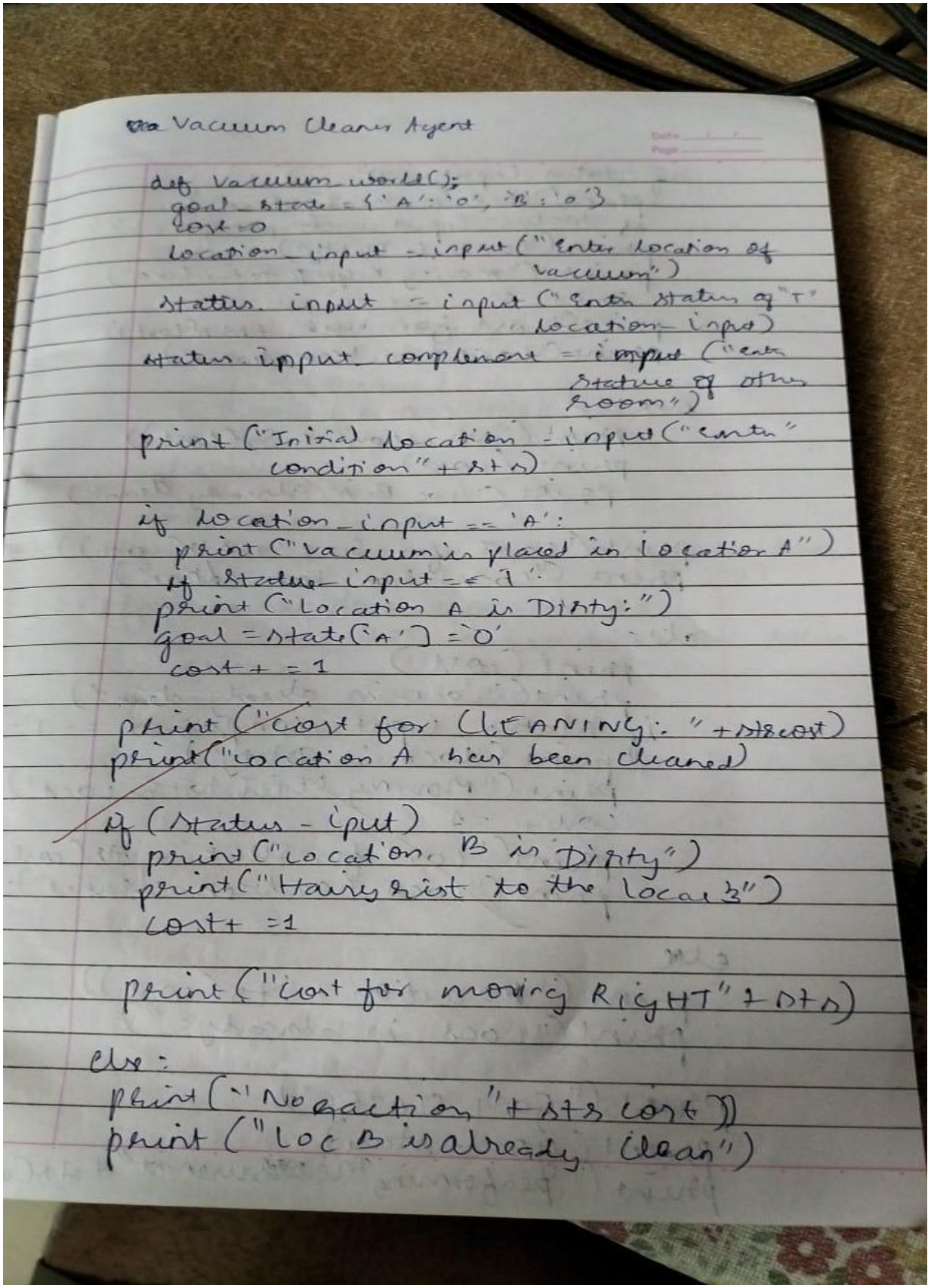
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5,6, 7, 8,-1]

astar(src, target)

Output:

```
Enter the start state matrix

1 0 1 0
1 0 0 1
1 1 1 1
Enter the goal state matrix

1 1 0 1
1 0 0 1
1 1 1 0
|
  |
 \'/

1 0 1 0
1 0 0 1
1 1 1 1
```

Lab Program 5



Vacuum Cleaner Agent

```
def Vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter location of
                            vacuum")
    status_input = input("Enter status of" "
                          location_input)
    status_input_complement = input("enter
                              status of other
                              room")
    print("Initial location condition" + str(s))

    if location_input == 'A':
        print("Vacuum is placed in location A")
        if status_input == '1':
            print("Location A is Dirty:")
            goal_state['A'] = '0'
            cost += 1

    print("Cost for CLEANING: " + str(cost))
    print("Location A has been cleaned")

    if (status_input_complement == '1'):
        print("Location B is Dirty")
        print("Moving right to the location B")
        cost += 1

    print("Cost for moving RIGHT" + str(cost))

    else:
        print("No action" + str(cost))
        print("Loc B is already clean")
```

```
if status_input == '0':
    print("LOCA is already clean");
    if status_input = complement == '1':
    print ("LOCB is DIRTY")
    print ("MOVING RIGHT to the LOCB")
     cost += 1
    print("cost for SUCK" +str(cost))
    print ("LOCB has been cleaned")

else:
    print ("No action" F Str(cost))
    print (cost)
     print ("Loc B is already clean")

print ("COST for SUCK" + str(cost))
print ("LOCA has been clean")

else:
    print (cost)
    print("LOCB is already clean")
    if status_input - complement == '1':
    print ("LOCA is DIRTY")
    print ("MOVING LEFT to the LOCA")
    cost + =1
    print("COST for SUCK" + str(cost))
    print ("LOCA has been cleaned").

else
    print ("No action" + str(cost))
    print("LOCA is already:")

Print ("GOAL STATE:")
print(goal-state)
print("performing Measurement" + str(cost))
```

Implement vacuum cleaner agent.

Objective: The objective of the vacuum cleaner agent is to clean the whole of two rooms by performing any of the actions – move right, move left or suck. Vacuum cleaner agent is a goal based agent.

Code:

```
def vacuum_world():

    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input+ " : ")
    status_input_complement = input("Enter status of other room :
    ")
    print("Initial Location Condition {A : " + str(status_input_complement) +
", B : " + str(status_input) + " }" )

    if location_input == 'A':
        print("Vacuum is placed in Location
        A") if status_input == '1':
            print("Location A is
            Dirty.") goal_state['A'] =
            '0'
            cost += 1                    #cost for suck
            print("Cost for CLEANING A " +
            str(cost)) print("Location A has been
            Cleaned.")

            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to the Location B.
                ") cost += 1
                print("COST for moving RIGHT " +
                str(cost)) goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK " +
                str(cost)) print("Location B has
                been Cleaned. ")
```

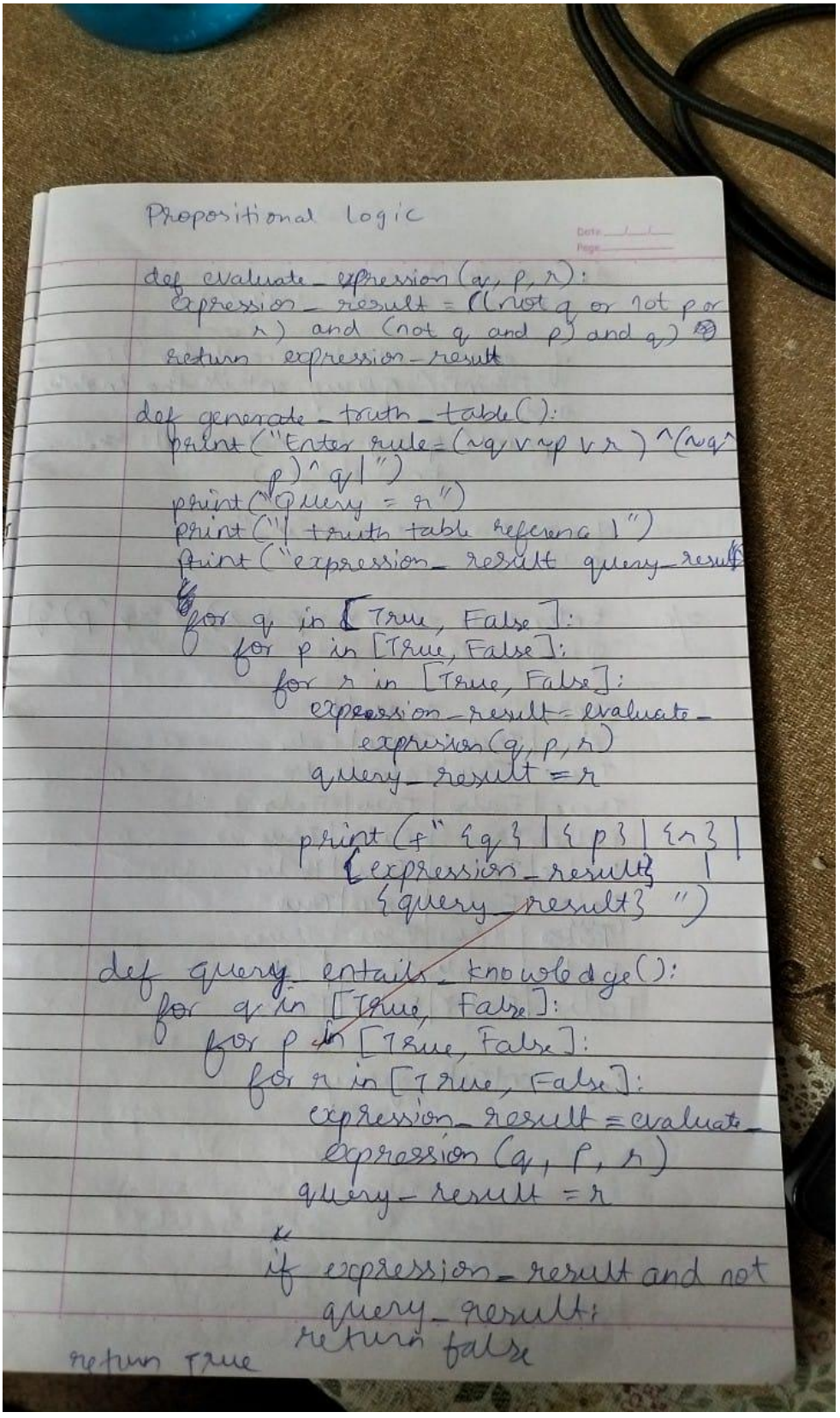
```
        else:
            print("No action" + str(cost))
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean
        ") if status_input_complement ==
        '1':
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B.
            ") cost += 1
            print("COST for moving RIGHT " +
            str(cost)) goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned.
            ")
        else:
            print("No action " +
            str(cost)) print(cost)
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location
    B") if status_input == '1':
        print("Location B is
        Dirty.") goal_state['B'] =
        '0'
        cost += 1
        print("COST for CLEANING " +
        str(cost)) print("Location B has been
        Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A.
            ") cost += 1
            print("COST for moving LEFT " +
            str(cost)) goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK " + str(cost))
```

```
            print("Location A has been Cleaned.")

        else:
            print(cost)
            print("Location B is already clean.")

            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A.
                ") cost += 1
                print("COST for moving LEFT " +
                str(cost)) goal_state['A'] = '0'
                cost += 1
                print("Cost for SUCK " + str(cost))
                print("Location A has been Cleaned.
                ")
            else:
                print("No action " + str(cost))
                print("Location A is already
                clean.")

    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output:

```
Enter Location of Vacuum: A
Enter status of A : 0
Enter status of other room : 1
Initial Location Condition {A : 1, B : 0 }
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Lab Program 6



Propositional Logic

```
def evaluate_expression(q, p, r):
    expression_result = ((not q or not p or
        r) and (not q and p) and q)
    return expression_result

def generate_truth_table():
    print("Enter rule=(~q v ~p v r) ^ (~q^
        p) ^ q|")
    print("query = r")
    print("| truth table referena |")
    print("expression_result query_result
    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_
                    expression(q, p, r)
                query_result = r

                print(f" {q} | {p} | {r} |
                    {expression_result} |
                    {query_result} ")

def query_entails_knowledge():
    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_
                    expression(q, p, r)
                query_result = r

                if expression_result and not
                    query_result:
                    return false
return True
```

```
def main():
    generate_truth_table()

    if query_entails_knowledge():
        print("query entails the know
    else:
        print("query does not entail the
                knowledge")

if __name__ == "__main__":
    main()
```

o/p Enter rule = (~q v ~p v r) ^ (~q ^ p)
Query = r

expression	result
True	True	True	False
True	True	False	False
True	False	True	False
False	True	False	True
False	True	True	False
False	False	True	True
True	True	True	True
False	False	False	True
False	False	False	False

Logic entails query

Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not.

Objective: The objective of this program is to see if the given query entails a knowledge base. A query is said to entail a knowledge base if the query is true for all the models where knowledge base is true.

Code:

```
combinations=[(True,True,
True),(True,True,False),(True,False,True),(True,False, False),(False,True,
True),(False,True, False),(False, False,True),(False,False, False)]
variable={'p':0,'q':1, 'r':2}
kb="
q="
priority={'~':3,'v':1,'^':
2} def input_rules():
    global kb, q
    kb = (input("Enter rule: "))
    q = input("Enter the Query:
") def entailment():
    global kb, q
    print("*10+"Truth Table
    Reference"+"*10) print('kb','alpha')
    print('*'*10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb),
        comb) f =
        evaluatePostfix(toPostfix(q), comb)
        print(s, f)
        print('-
        '*10) if s
        and not f:
            return
    False return
    True
def isOperand(c):
    return c.isalpha() and

c!='v' def

isLeftParanthesis(c):
```

```
    return c == '('

def
    isRightParanthesis(c)
    : return c == ')'

def isEmpty(stack):
    return len(stack)
    == 0

def
    peek(stack):
    return
    stack[-1]

def hasLessOrEqualPriority(c1,
    c2): try:
        return
    priority[c1]<=priority[c2]
    except KeyError:
        return False
def
toPostfix(infix):
    stack = []
    postfix =
    '' for c in
    infix:
        if
            isOperand
            (c):
            postfix +=
            c
        else:
            if
                isLeftParanthesis
                (c):
                stack.append(c)
            elif
                isRightParanthesis(c)
                : operator =
                stack.pop()
```

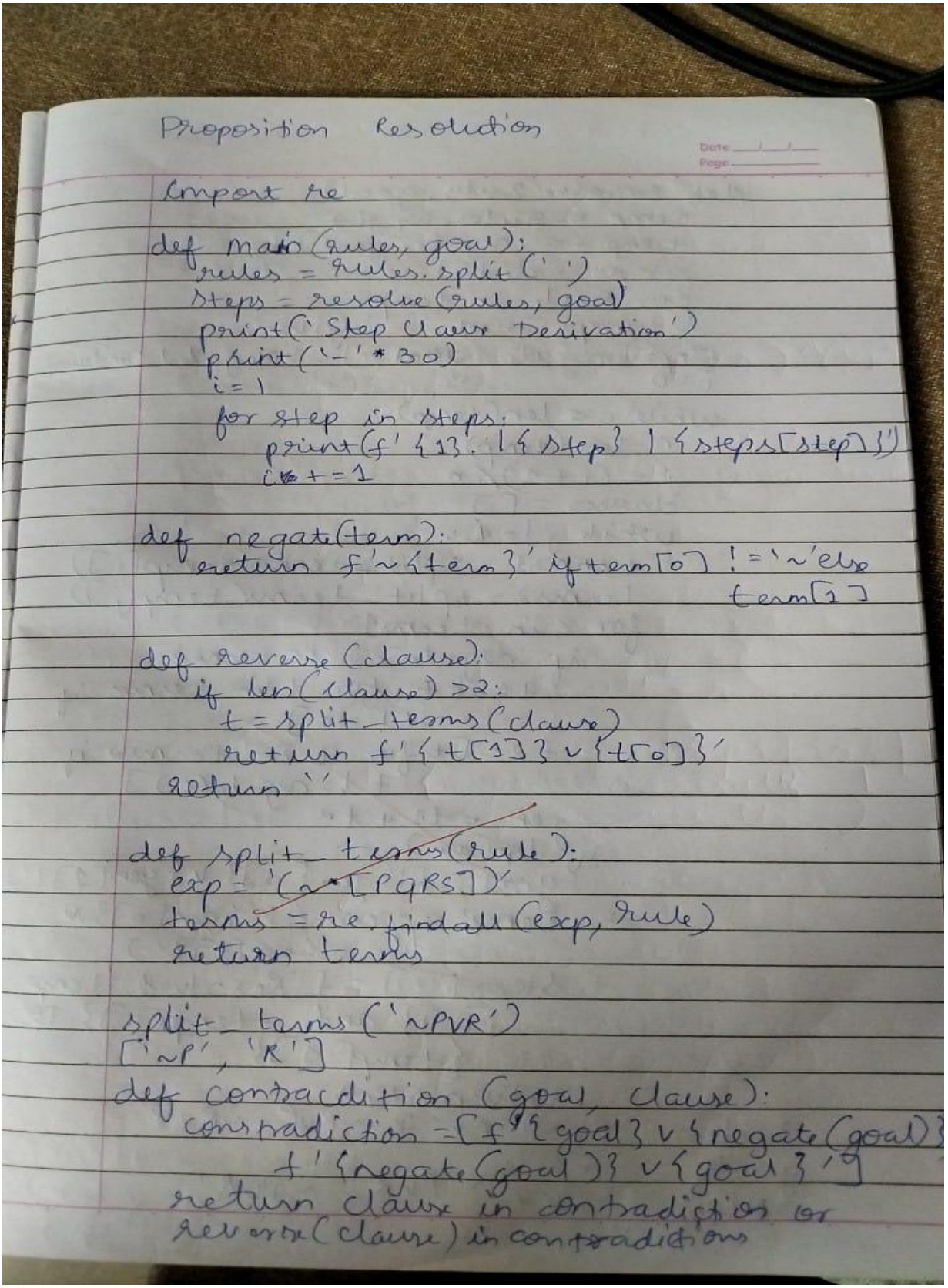
```
            while not
                isLeftParanthesis(operator):
                postfix += operator
                operator = stack.pop()
        else:
            while (not isEmpty(stack)) and
hasLessOrEqualPriority(c, peek(stack)):
                postfix +=
            stack.pop()
            stack.append(c)
    while (not
        isEmpty(stack)):
        postfix += stack.pop()
```

```
    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not
            val1)
        else:
            val1 =
            stack.pop()
            val2 =
            stack.pop()
            stack.append(_eval(i,val2,val1))
    return stack.pop()
def _eval(i, val1,
    val2): if i == '^':
        return val2 and
    val1 return val2 or
    val1
#Test 1
input_rules()
ans =
entailment() if
ans:
    print("Knowledge Base entails
query") else:
    print("Knowledge Base does not entail
query") #Test 2
input_rules()
ans =
entailment() if
ans:
    print("Knowledge Base entails
query") else:
    print("Knowledge Base does not entail query")
```

Output:

```
Enter rule: (~qv~pvr)^(~q^p)^q
Enter the Query: r
Truth Table Reference
kb alpha
**********
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
Knowledge Base entails query
```

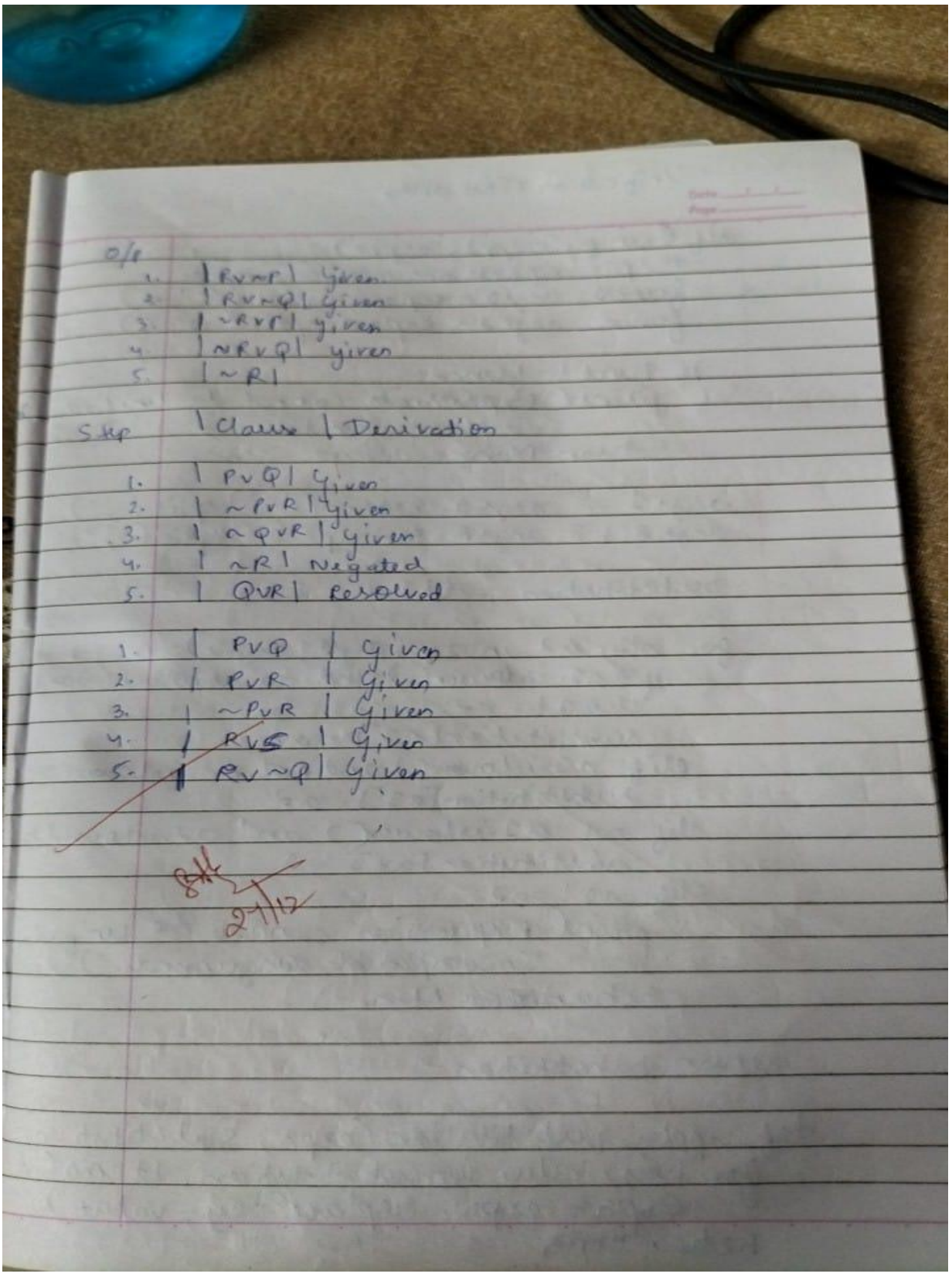
Lab Program 7



Proposition Resolution

```
import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('Step Clause Derivation')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f'{i}. | {step} | {steps[step]}')
        i += 1

def negate(term):
    return f'~{term}' if term[0] != '~' else
                                    term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]} v {t[0]}'
    return ''

def split_terms(rule):
    exp = '(~[PQRS])'
    terms = re.findall(exp, rule)
    return terms

split_terms('~PvR')
['~P', 'R']
def contradiction(goal, clause):
    contradiction = [f'{goal} v {negate(goal)}',
                     f'{negate(goal)} v {goal}']
    return clause in contradictions or
    reverse(clause) in contradictions
```

```
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if
                          t != c]
                    t2 = [t for t in terms2 if
                          t != negate(c)]
                    gen = t1 + t2
                    if len(gen)
                    good, f'{gen[0]} v {gen[1]}'
                    temp.append(f'{gen[0]} v
                                {gen[1]}')
                    steps[''] = f"Resolved {temp
                      [i]} and {temp[j]} to
                        {temp[-1]}
                    return steps
rules = 'RvP  Rv~Q  ~RvP  ~RvQ'
goals = 'R'
rules = 'PvQ  ~PvR  ~QvR'
```



O/P

1. | PvP | Given
2. | RV~R | Given
3. | ~PvP | Given
4. | ~PvQ | Given
5. | ~R |

Step	Clause	Derivation
1.	PvQ	Given
2.	~PvR	Given
3.	~QvR	Given
4.	~R	Negated
5.	QvR	Resolved

1.	PvQ	Given
2.	PvR	Given
3.	~PvR	Given
4.	RvS	Given
5.	Rv~Q	Given

24/12

Create a knowledgebase using prepositional logic and prove the given query using resolution

Objective: The resolution takes two clauses and produces a new clause which includes all the literals except the two complementary literals if exists. The knowledge base is conjuncted with the not of the give query and then resolution is applied.

Code:

```
def
   disjunctify(clauses)
   : disjuncts = []
   for clause in clauses:
      disjuncts.append(tuple(clause.split('v')))
   return disjuncts

def getResolvant(ci, cj, di,
   dj): resolvant = list(ci) +
   list(cj)
   resolvant.remove(di)
   resolvant.remove(dj)
   return tuple(resolvant)


def resolve(ci,
   cj): for di in
   ci:
      for dj in cj:
         if di == '~' + dj or dj == '~' + di:
            return getResolvant(ci, cj, di,
            dj)

def checkResolution(clauses, query):
   clauses += [query if query.startswith('~') else '~' + query]
   proposition = '^'.join(['(' + clause + ')' for clause in clauses])
   print(f'Trying to prove {proposition} by contradiction ... ')

   clauses =
   disjunctify(clauses)
   resolved = False
```

```
    new = set()

    while not
        resolved: n =
        len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1,
        n)] for (ci, cj) in pairs:
            resolvant = resolve(ci,
            cj) if not resolvant:
                resolved =
                True break
            new =
        new.union(set(resolvents)) if
        new.issubset(set(clauses)):
            break
        for clause in new:
            if clause not in clauses:
                clauses.append(clau
                se)

    if resolved:
        print('Knowledge Base entails the query, proved by resolution')
    else:
        print("Knowledge Base doesn't entail the query, no empty set produced
after resolution")
clauses = input('Enter the clauses
').split() query = input('Enter the query:
') checkResolution(clauses, query)
```

Output:

```
#Test1
TELL(['implies', 'p', 'q'])
TELL(['implies', 'r', 's'])
ASK(['implies',['or','p','r'], ['or', 'q', 's']])
```

```
True
```

```
CLEAR()
```

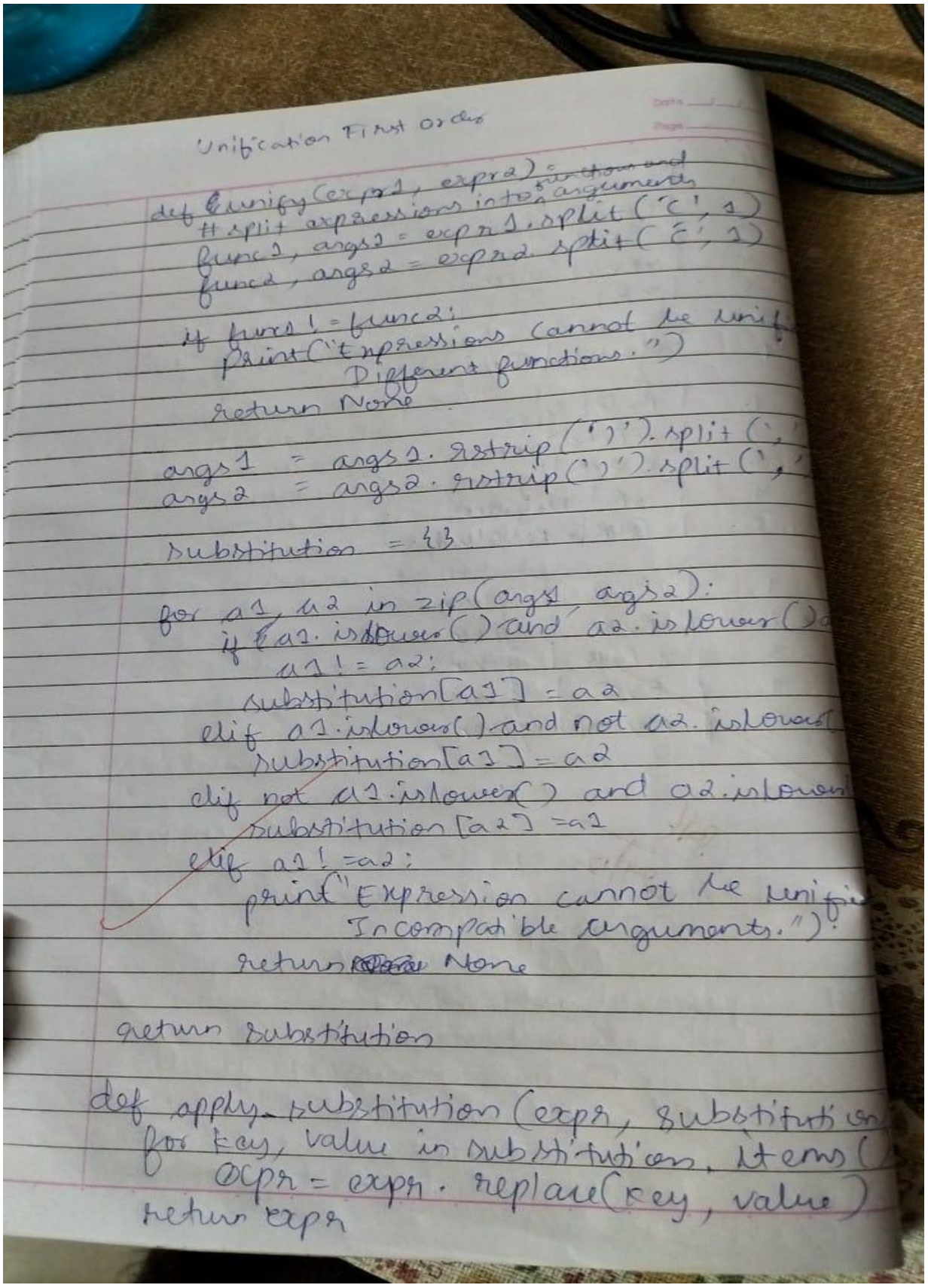
```
#Test2
TELL('p')
TELL(['implies',['and','p','q'],'r'])
TELL(['implies',['or','s','t'],'q'])
TELL('t')
ASK('r')
```

```
True
```

```
CLEAR()
```

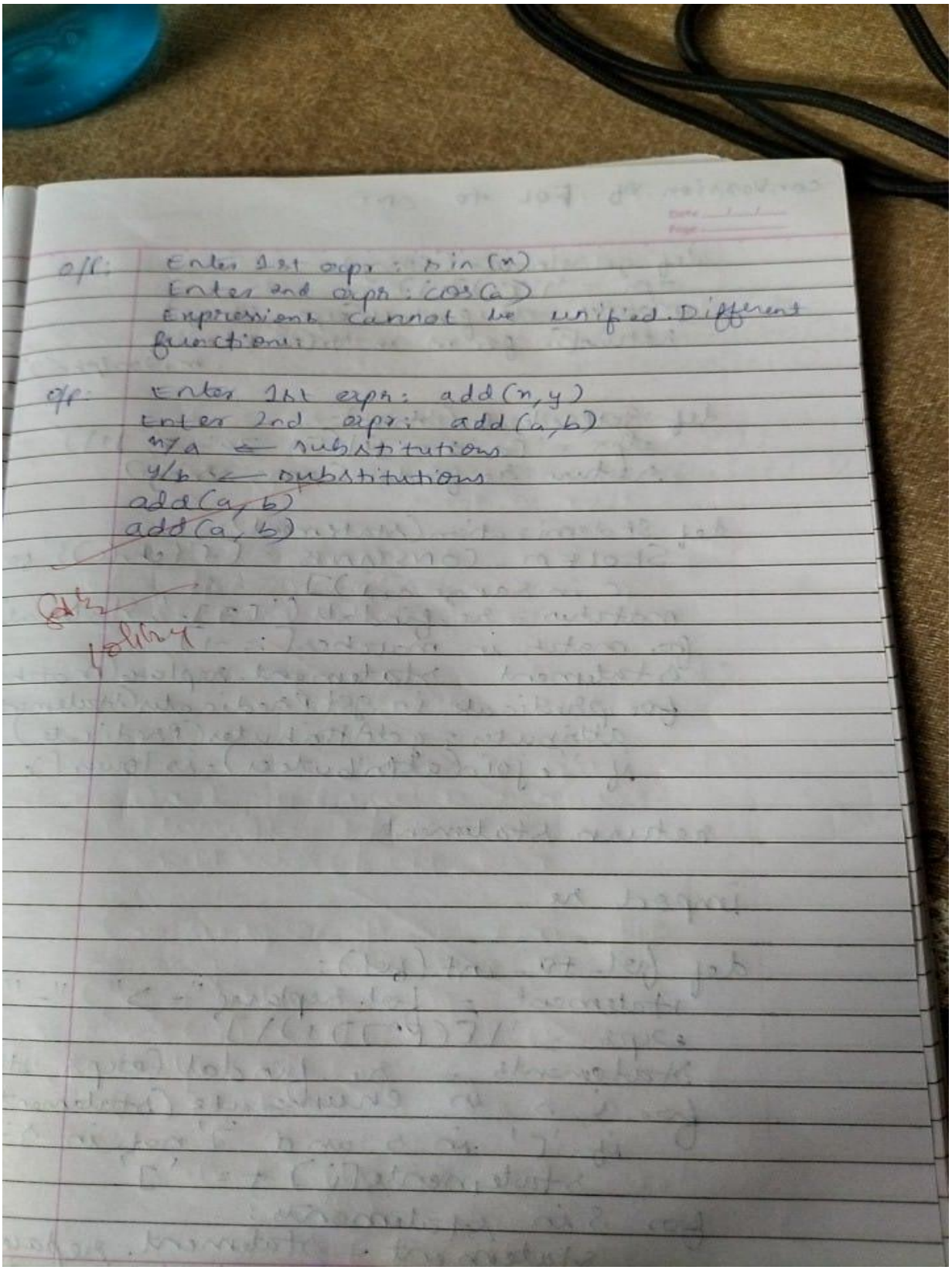
```
#Test3
TELL('a')
TELL('b')
TELL('c')
TELL('d')
ASK(['or', 'a', 'b', 'c', 'd'])
```

Lab Program 8



Unification First Order

```
def Unify(expr1, expr2):
    # split expressions into function and arguments
    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

    if func1 != func2:
        print("Expressions cannot be unified. Different functions.")
        return None

    args1 = args1.rstrip(')').split(',')
    args2 = args2.rstrip(')').split(',')

    substitution = {}

    for a1, a2 in zip(args1, args2):
        if a1.islower() and a2.islower():
            a1 != a2:
            substitution[a1] = a2
        elif a1.islower() and not a2.islower():
            substitution[a1] = a2
        elif not a1.islower() and a2.islower():
            substitution[a2] = a1
        elif a1 != a2:
            print("Expression cannot be unified. Incompatible arguments.")
            return None

    return substitution

def apply_substitution(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
    return expr
```



o/p: Enter 1st expr: sin(x)
Enter 2nd expr: cos(a)
Expressions cannot be unified. Different functions

o/p: Enter 1st expr: add(x,y)
Enter 2nd expr: add(a,b)
x/a ← substitutions
y/b ← substitutions
add(a,b)
add(a,b)

Implement unification in first order logic

Objective: Unification can find substitutions that make different logical expressions identical. Unify takes two sentences and make a unifier for the two if a unification exist.

Code:

```
import re
def getAttributes(expression):
    expression =
    expression.split("(")[1:] expression
    = "(".join(expression) expression =
    expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def
    getInitialPredicate(expression)
    : return
    expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in
    enumerate(attributes):
        if val == old:
            attributes[index] =
            new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
```

```
        new, old = substitution
        exp = replaceAttributes(exp, old,
    new) return exp

def checkOccurs(var,
    exp): if exp.find(var)
    == -1:
        return
    False
    return True
def getFirstPart(expression):
    attributes =
    getAttributes(expression) return
    attributes[0]
def getRemainingPart(expression):
    predicate =
    getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1,
    exp2): if exp1
    == exp2:
        return []
    if isConstant(exp1) and
        isConstant(exp2): if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be
            unified") return []

    if isConstant(exp1):
        return [(exp1,
        exp2)]

    if isConstant(exp2):
        return [(exp2,
        exp1)]

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
```

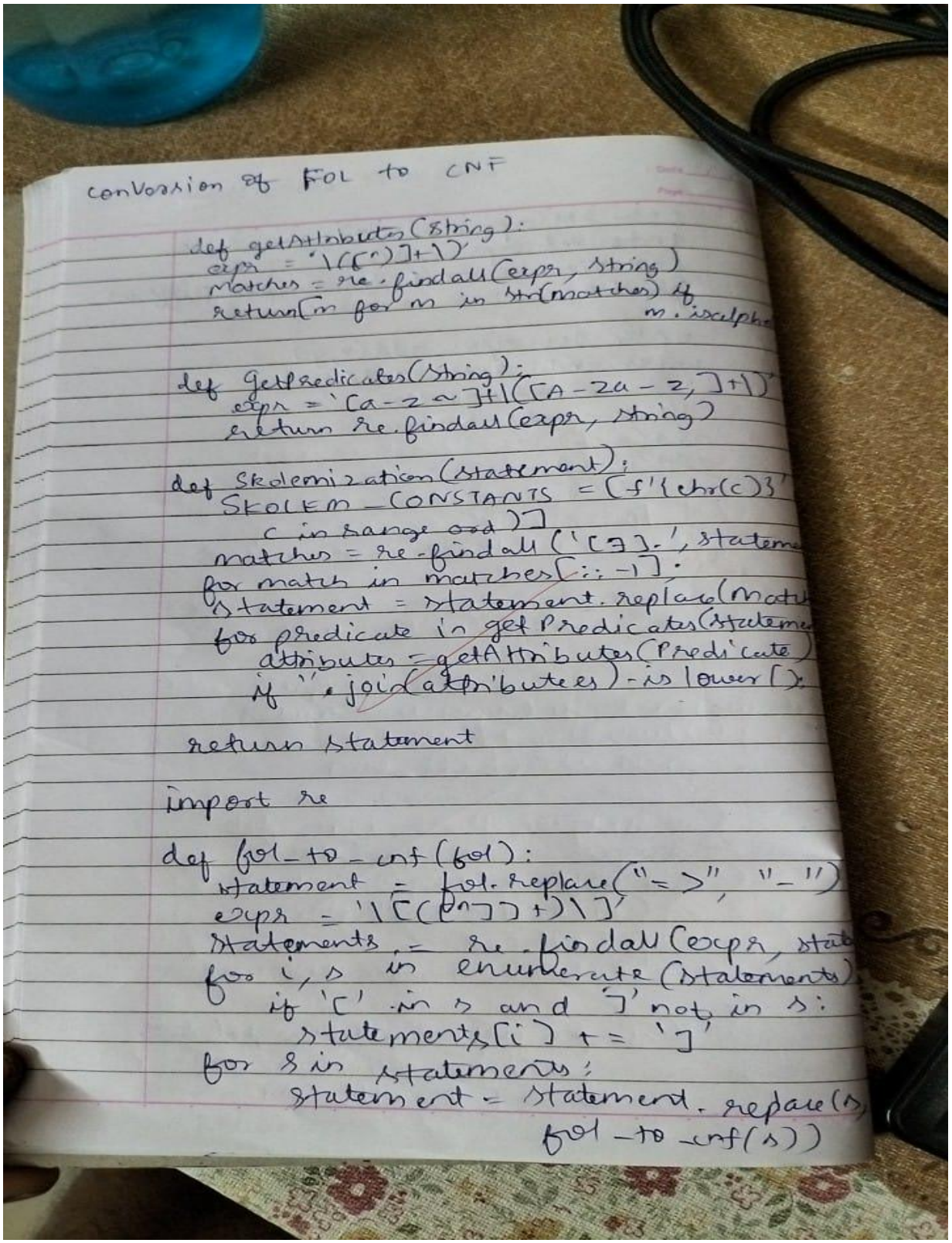
```
print("Cannot be unified as the predicates do not match!")
return []

    attributeCount1=
    len(getAttributes(exp1))
    attributeCount2=
    len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2}
  do not match. Cannot be unified")
        return []

    head1 =
    getFirstPart(exp1)
    head2 =
    getFirstPart(exp2)
    initialSubstitution = unify(head1,
    head2) if not initialSubstitution:
        return []
    if attributeCount1 == 1:
        return
        initialSubstitution

    tail1 =
    getRemainingPart(exp1)
    tail2 =
    getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1,
        initialSubstitution) tail2 =
        apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1,
    tail2) if not remainingSubstitution:
        return []
    return initialSubstitution +


  remainingSubstitution if__name___== "__main
```

```
print("The substitutions are:")
print([' / '.join(substitution) for substitution in substitutions])
```

Output:

```
Enter the first expression
king(x)
Enter the second expression
king(john)
The substitutions are:
['john / x']
```

Lab Program 9



Conversion of FOL to CNF

```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return[m for m in str(matches) if
                                m.isalph

def getPredicates(string):
    expr = '[a-zA-Z]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def Skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}'
        c in range(ord
    matches = re.findall('[∀].', statem
    for match in matches[::-1]:
        statement = statement.replace(matc
        for predicate in getPredicates(statem
            attributes = getAttributes(Predicate
            if ''.join(attributes).islower():

    return statement

import re

def fol_to_cnf(fol):
    statement = fol.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, stat
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s,
                                fol_to_cnf(s))
```

while '-' in statement:
    i = statement.index('-')
    br = statement.index('[') if '[' in
    new_statement = '~' + statement[br:i]+
                    '|' + statement[i+1:]

return Skolemization(statement)
print(fol_to_cnf("bird(x)⇒ ~fly(x)"))
print(fol_to_cnf("∃x(bird(x)⇒ ~fly(x))"))

o/p: ~bird(x)|~fly(x)
[bird(A)|~fly(A)]

Sk 2
17/11/24

Convert given first order logic statement into Conjunctive Normal Form (CNF).

Objective: FOL logic is converted to CNF makes implementing resolution theorem easier.

Code:

```
import re

def
    getAttributes(string)
    : expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
```

```
    expr = '[a-z~]+\([A-Za-
    z,]+\)' return
    re.findall(expr, string)

def DeMorgan(sentence):
    string =
    ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string =
    string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate,
    f'~{predicate}') s = list(string)
    for i, c in
        enumerate(string): if c
        == 'V':
            s[i] =
        '^' elif c
        == '^':
            s[i] = 'V'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if flag else string
```

```
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
     matches = re.findall('[A∃].', statement)

    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[^]]+\]]',
        statement) for s in statements:
            statement = statement.replace(s, s[1:-
        1]) for predicate in
        getPredicates(statement):
            attributes =
            getAttributes(predicate) if
            ''.join(attributes).islower():
                statement =
statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not
                a.islower()][0] statement =
                statement.replace(aU,
f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')
    return statement

def fol_to_cnf(fol):

    statement = fol.replace("<=>",
    "_") while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^['+
statement[i+1:] + '=>' + statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>",
    "-") expr = '\[([^]]+)\]'
    statements = re.findall(expr,
    statement) for i, s in
    enumerate(statements):
        if '[' in s and ']' not
            in s: statements[i]
            += ']'
```

```
for s in statements:
```

```
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' +
        statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~A' in statement:
        i = statement.index('~A')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = 'Ǝ',
statement[i+2], '~'
        statement = ''.join(statement)
    while '~Ǝ' in statement:
        i = statement.index('~Ǝ')
        s = list(statement)
        s[i], s[i+1], s[i+2] = 'A', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[A','[~A')
    statement = statement.replace('~[Ǝ','[~Ǝ')
    expr = '(~[AVƎ].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr,
    statement) for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement

def main():
    print("Enter
    FOL:") fol =
    input()
    print("The CNF form of the given FOL is:
    ") print(Skolemization(fol_to_cnf(fol)))
```

main()

Output:

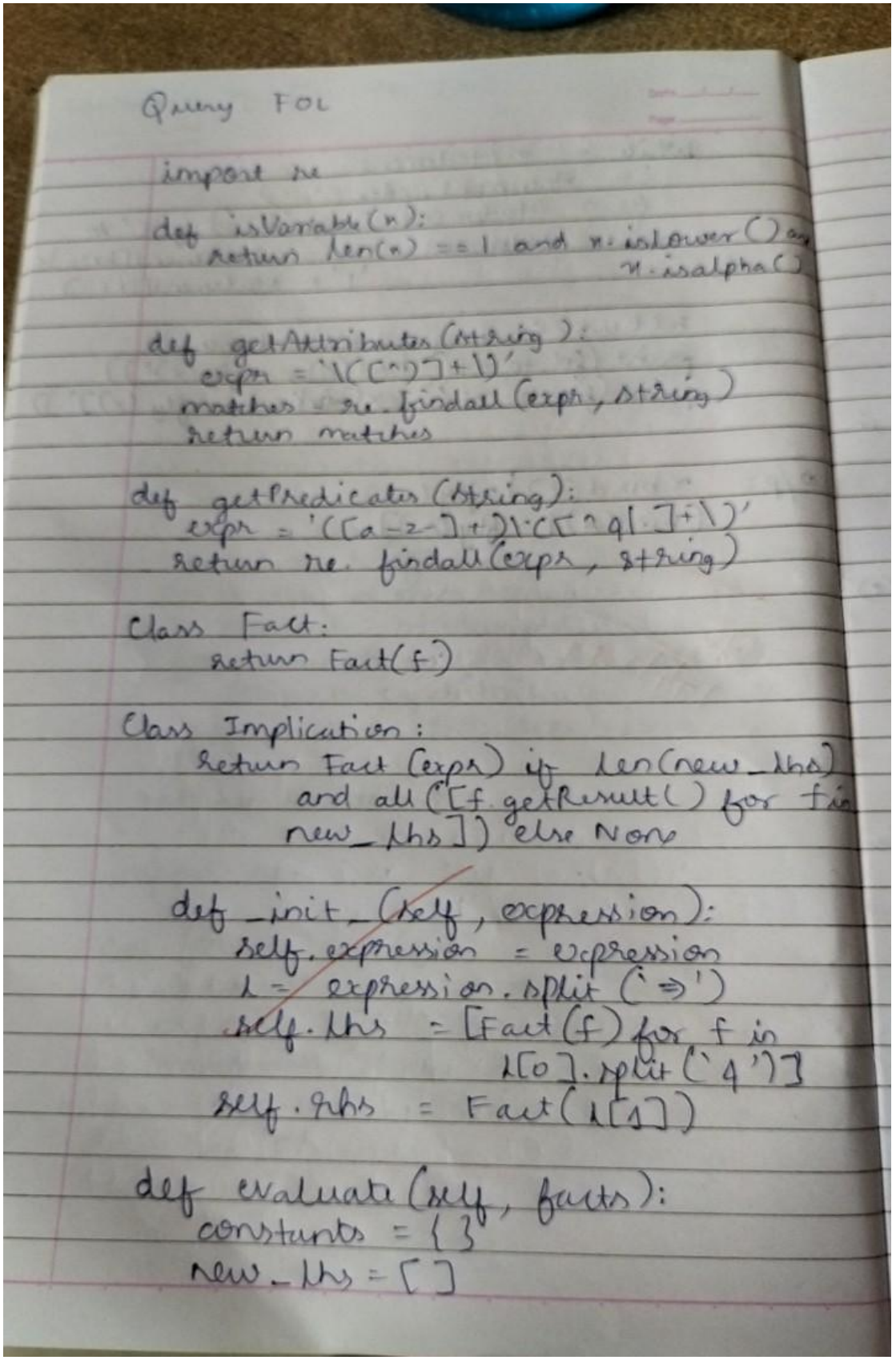
```
main()
```

```
Enter FOL:
∀x food(x) => likes(John, x)
The CNF form of the given FOL is:
~ food(A) V likes(John, A)
```

```
main()
```

```
Enter FOL:
∀x[∃z[loves(x,z)]]
The CNF form of the given FOL is:
[loves(x,B(x))]
```

Lab Program 10



Query FOL

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and
                           x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z]+)\([^&|]+\)'
    return re.findall(expr, string)

Class Fact:
    return Fact(f)

Class Implication:
    Return Fact(expr) if len(new_lhs)
        and all([f.getResult() for fa
        new_lhs]) else None

    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in
                    l[0].split('&')]
        self.rhs = Fact(l[1])

def evaluate(self, facts):
    constants = {}
    new_lhs = []
```

```
        else:
            self.facts.add(Fact(e))

        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in
                     self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).
                                    predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts:")
        for i, f in enumerate(set([f.expressio
                                   for f in self.facts])):
            print(f'\t{i+1}. {f}')

kb = KB()
kb.tell('king(x) & greedy(x) => evil(x)')
kb.tell('king(John)')
kb.tell('Greedy(John)')
kb.tell('king(Richard)')
kb.query('evil(x)')
```

```
for fact in facts:
    for val in self.lhs
        if val.predicate == fact.predicate,
            for i,v in enumerate(val.
                                 getVariables()):

                if v:
                    constants[v] = fact.getConstant
                                   ()[i]

        new_lhs.append(fact)

predicate, attributes = getPredicates(self.rhs.
                        expression)[0], str(getAttributes
                        (self.rhs.expression)[0])

for key in constants:
    if constants[key]:
        attributes = attributes.replace(key,
                     constants[key])

expr = f'{predicate}({attributes})'

return Fact(expr) if len(new_lhs) and
    all([f.getResult() for f in new_lhs]
    ) else None

class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '⇒' in e:
            self.implications.add(Implications(e))
```

O/P. Querying evil(x):
1. evil(John)

3.1.1.24.

Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.

Objective: A forward-chaining algorithm will begin with facts that are known. It will proceed to trigger all the inference rules whose premises are satisfied and then add the new data derived from them to the known facts, repeating the process till the goal is achieved or the problem is solved.

Code:

```
import re
def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def
    getAttributes(string)
    : expr = '\([^)]+\)'
    matches = re.findall(expr,
    string) return matches

def getPredicates(string):
    expr = '([a-
    z~]+)\([^&|]+\)'
    return re.findall(expr,

string) class Fact:

    def__init__(self,
        expression):
        self.expression =
        expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate =
        getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]
```

```
    def
        getResult(self)
        : return
        self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self,
        constants): c =
        constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p
for p in self.params])})"
        return

Fact(f) class

Implication:

    def_init_(self,  expression):
        self.expression             =
        expression          l       =
        expression.split('=>')
        self.lhs    =    [Fact(f)    for    f    in
        l[0].split('&')] self.rhs = Fact(l[1])

    def evaluate(self,
        facts): constants =
        {} new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in
                        enumerate(val.getVariables()): if v:
                            constants[v] =
                    fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = :
```

```
attributes = attributes.replace(key, constants[key]) expr =
f'{predicate}{attributes}'
return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else
None

  class KB:

    def__init__(self):
      self.facts = set()
      self.implications =
      set()

    def tell(self,
      e): if '=>'
      in e:
        self.implications.add(Implication(
      e)) else:
        self.facts.add(Fact(
      e)) for i in
      self.implications:
        res =
        i.evaluate(self.facts) if
        res:
          self.facts.add(res)

    def ask(self, e):
      facts = set([f.expression for f in
      self.facts]) i = 1
      print(f'Querying
      {e}:') for f in facts:
        if Fact(f).predicate == Fact(e).predicate:
          print(f'\t{i}. {f}')
          i += 1

    def display(self):
      print("All facts:
      ")
      for i, f in enumerate(set([f.expression for f in self.facts])):
        print(f'\t{i+1}. {f}')
```

```
print("Enter the number of FOL expressions present in
KB:") n = int(input())
print("Enter the
expressions:") for i in
range(n):
    fact =
    input()
    kb.tell(fac
    t)
print("Enter the
query:") query =
input() kb.ask(query)
kb.display()
```

Output:

```
Querying criminal(x):
1. criminal(West)
All facts:
    1. american(West)
    2. sells(West,M1,Nono)
    3. owns(Nono,M1)
    4. missile(M1)
    5. enemy(Nono,America)
    6. weapon(M1)
    7. hostile(Nono)
    8. criminal(West)
Querying evil(x):
    1. evil(John)
```