**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:*

# Snigdha S Kashyap

1BM21CS215

Department of Computer Science and Engineering B.M.S. College of Engineering

Bull Temple Road, Basavanagudi, Bangalore 560 Nov 2023 - Feb 2024

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 Snigdha S Kashyap (1BM21CS215) during the 5th Semester Nov 2023 – Feb 2024.

Signature of the Faculty In charge:

Sneha S Bagalkot Assistant Professor

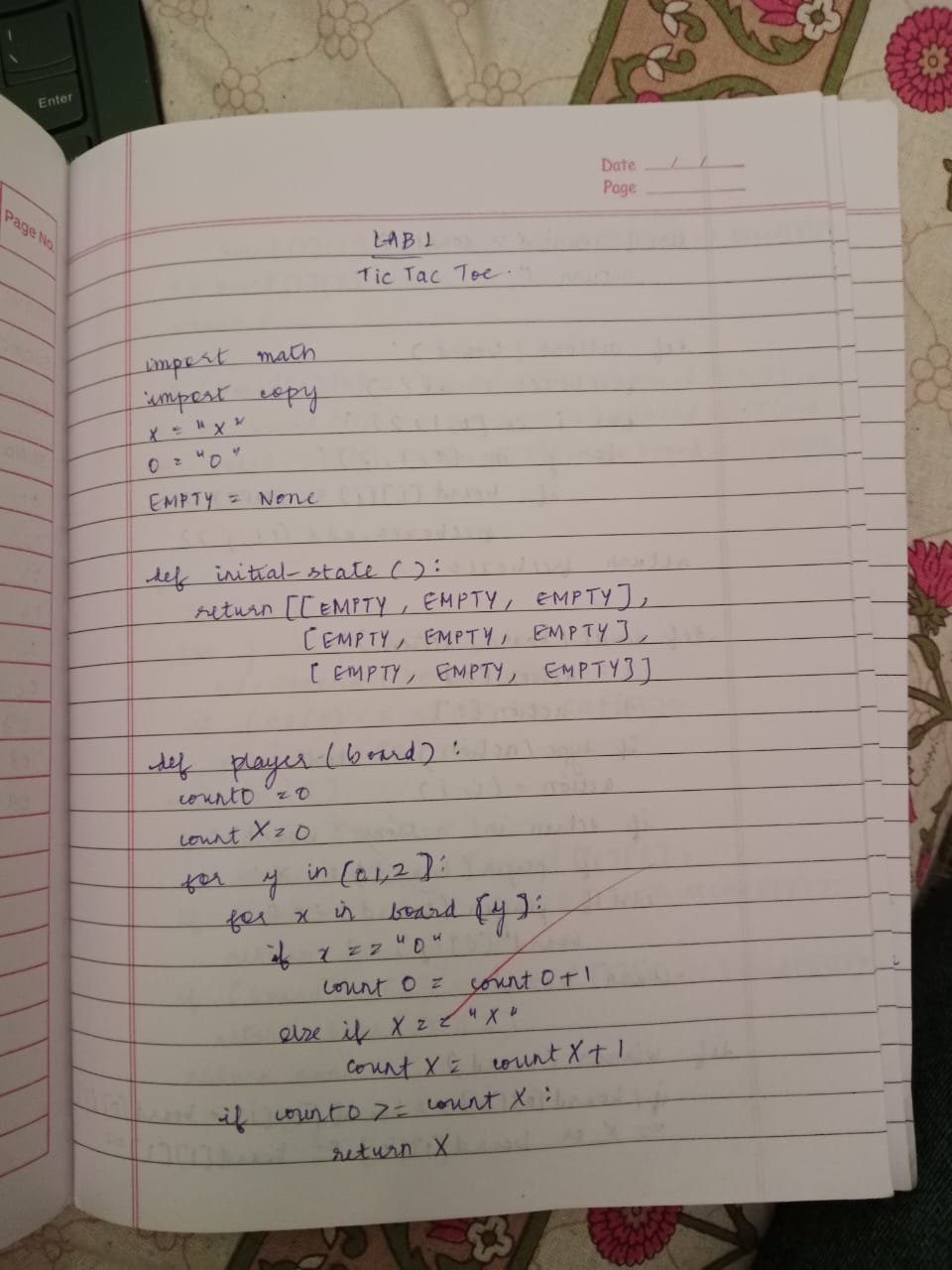
Department of Computer Science and Engineering

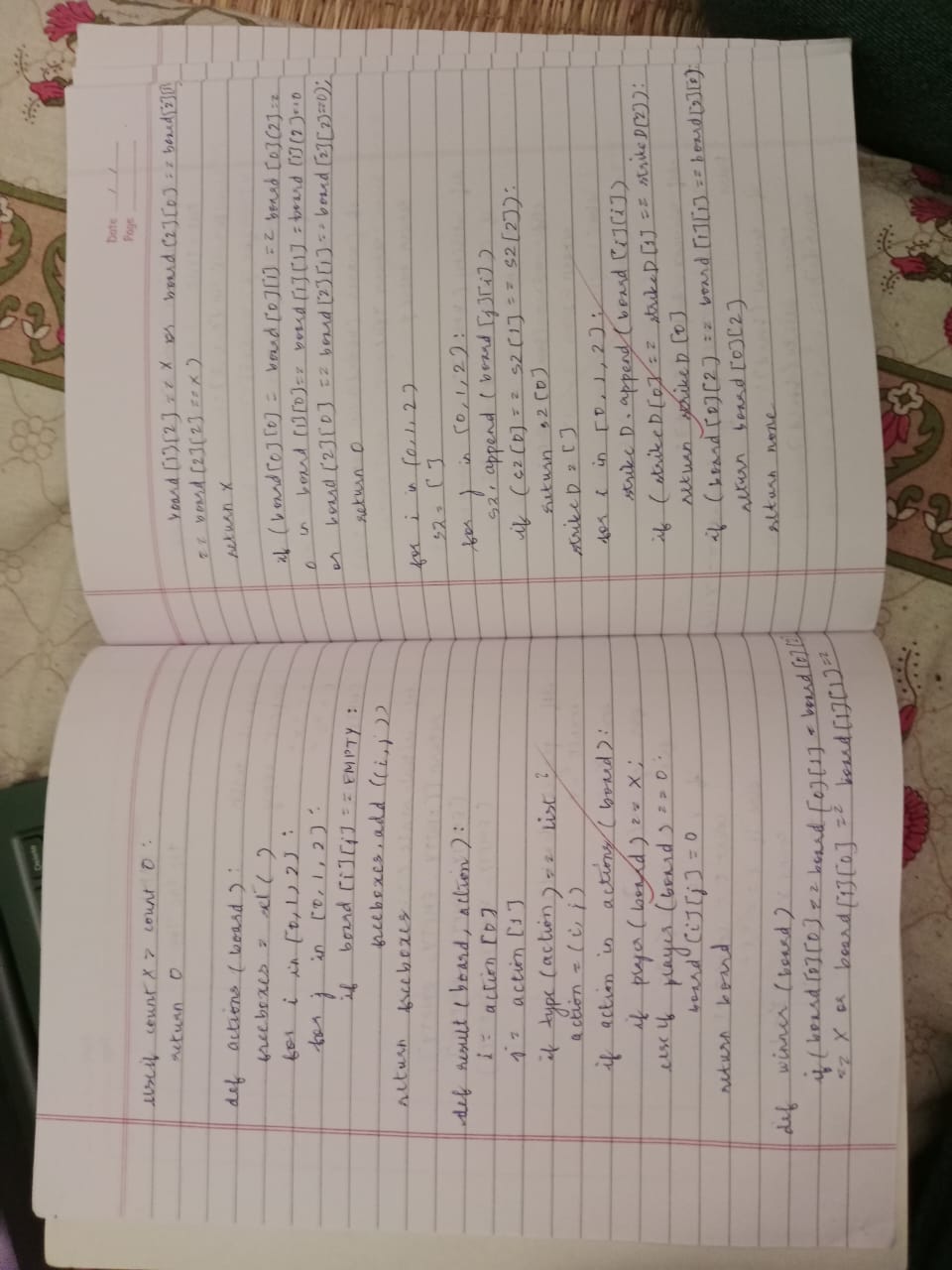
B.M.S. College of Engineering, Bangalore

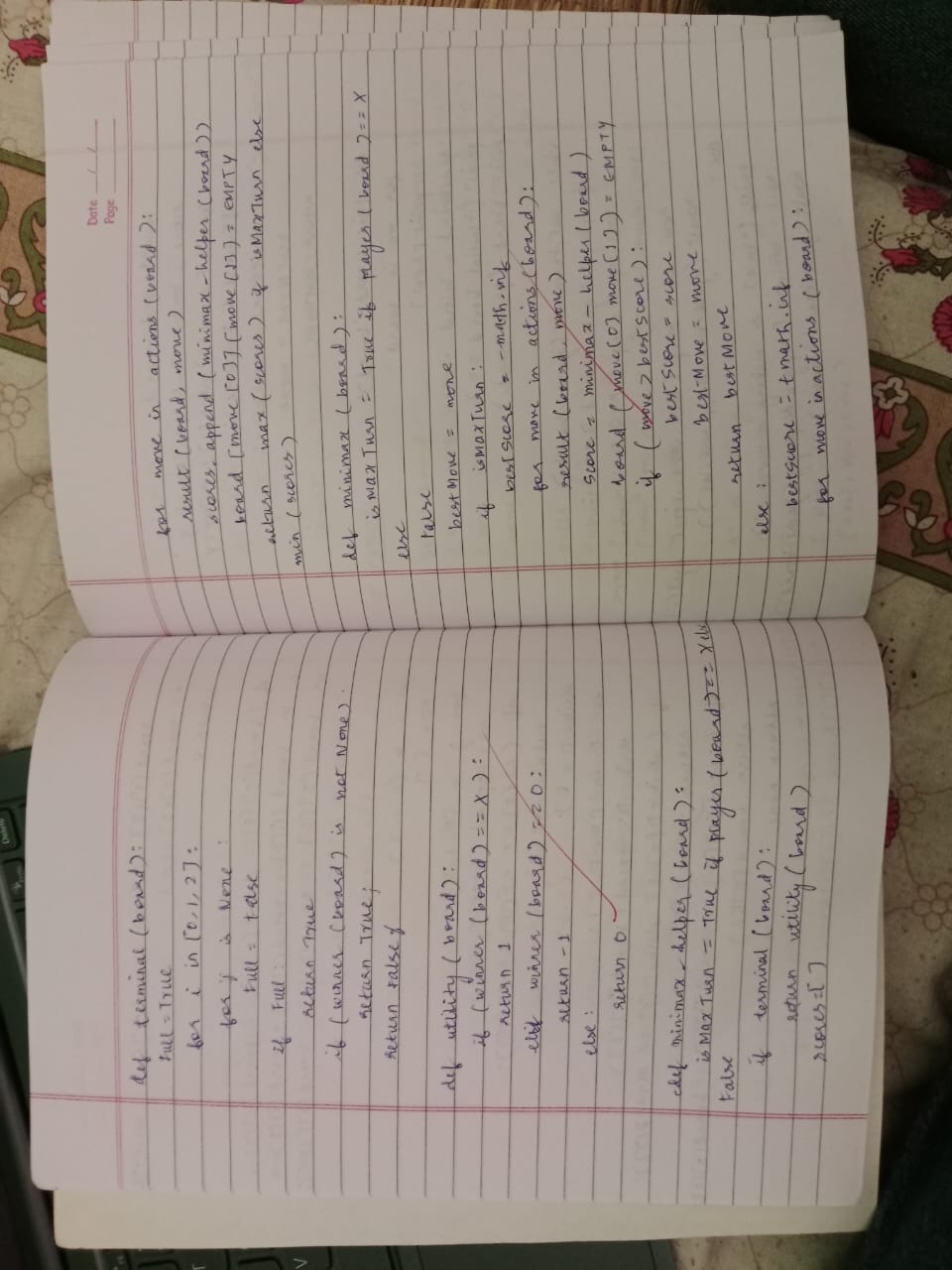
# Table of Contents

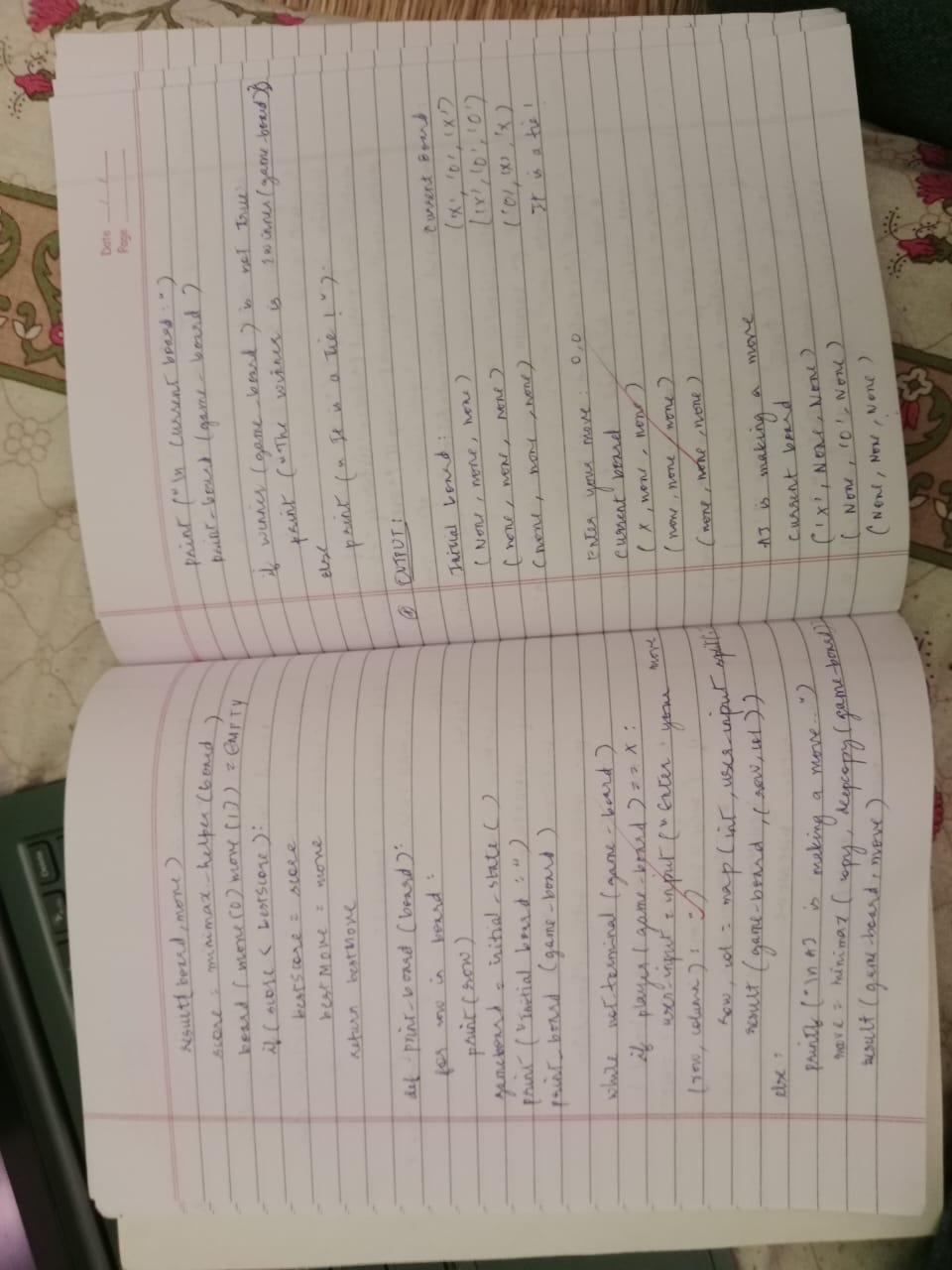
|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Title** | **Page No.** |
| 1. | Tic Tac Toe | 4-13 |
| 2. | 8 Puzzle Breadth First Search Algorithm | 14-21 |
| 3. | 8 Puzzle Iterative Deepening Search Algorithm | 22-28 |
| 4. | 8 Puzzle A\* Search Algorithm | 29-33 |
| 5. | Vacuum Cleaner | 34-38 |
| 6. | Knowledge Base Entailment | 39-44 |
| 7. | Knowledge Base Resolution | 45-49 |
| 8. | Unification | 50-55 |
| 9. | FOL to CNF | 56-60 |
| 10. | Forward reasoning | 61-64 |

**Lab-Program-1**

****







# 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(board): 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 within the range!') except:

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

def main():

print('Welcome to Tic Tac Toe!') printBoard(board)

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(board): print('Tie Game!')

while True:

answer = input('Do you want to play again? (Y/N)') if answer.lower() == 'y' or answer.lower == 'yes':

board = [' ' for x in range(10)] print(' ') main()

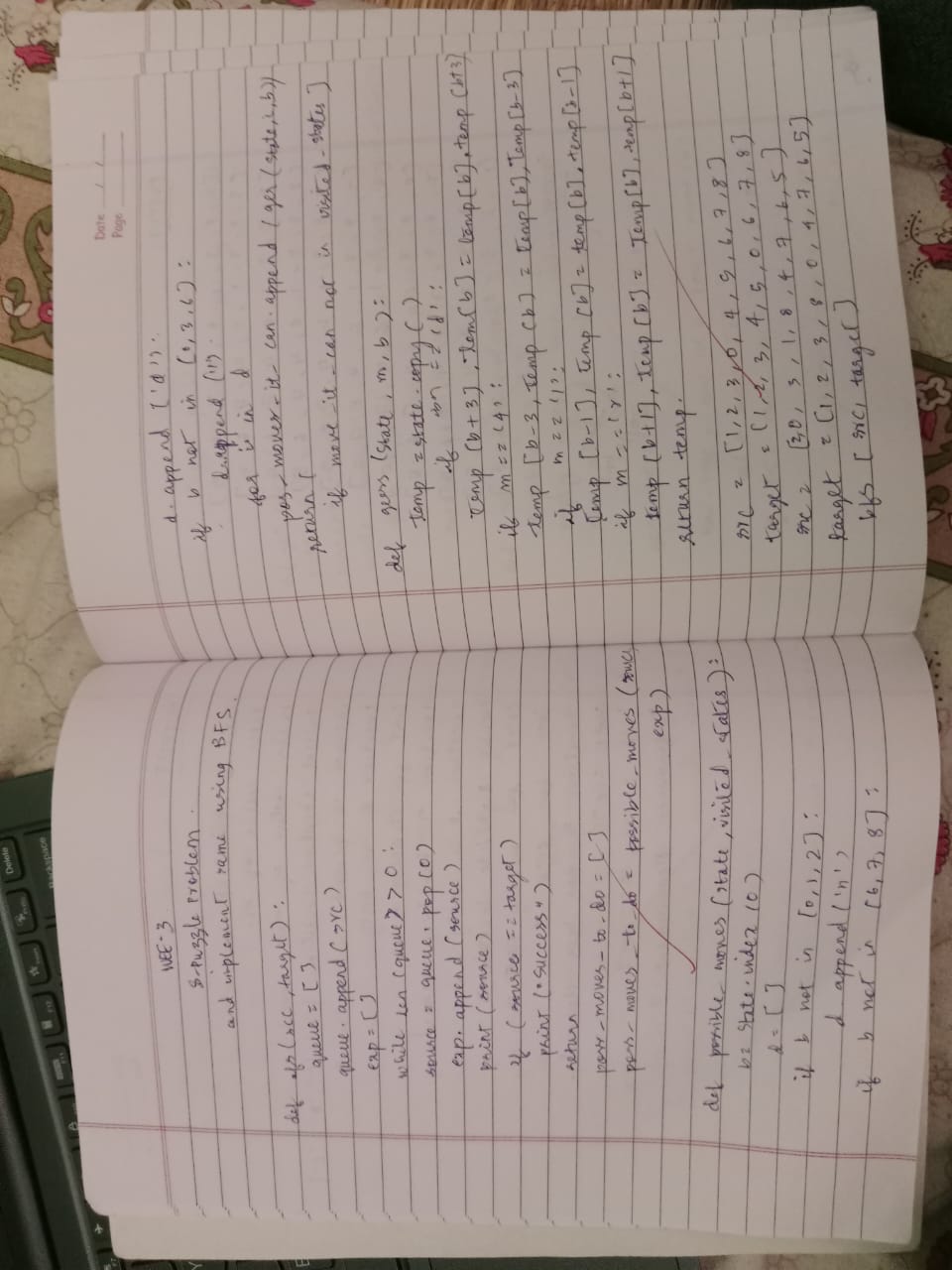
else:

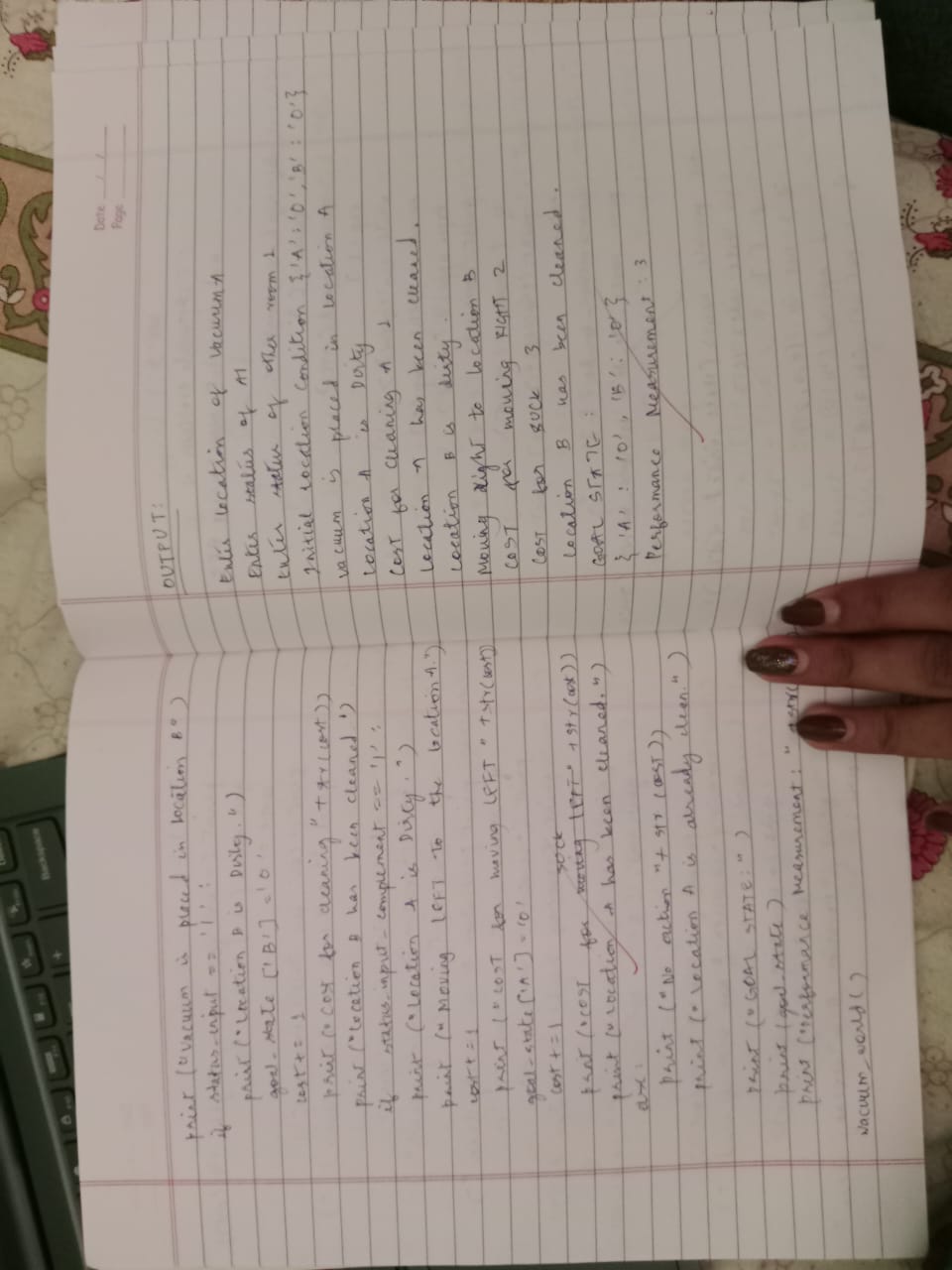
break;

# Output:



**Lab-Program-2**





# 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*

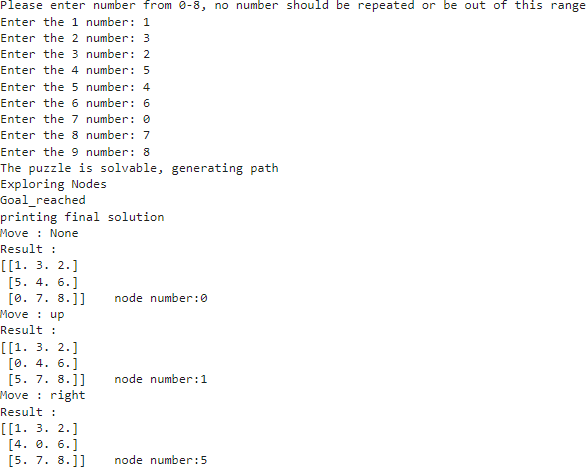
goal, s, v **=** exploring\_nodes(root)

**if** goal **is None and** s **is None and** v **is None**: print("Goal State could not be reached, Sorry")

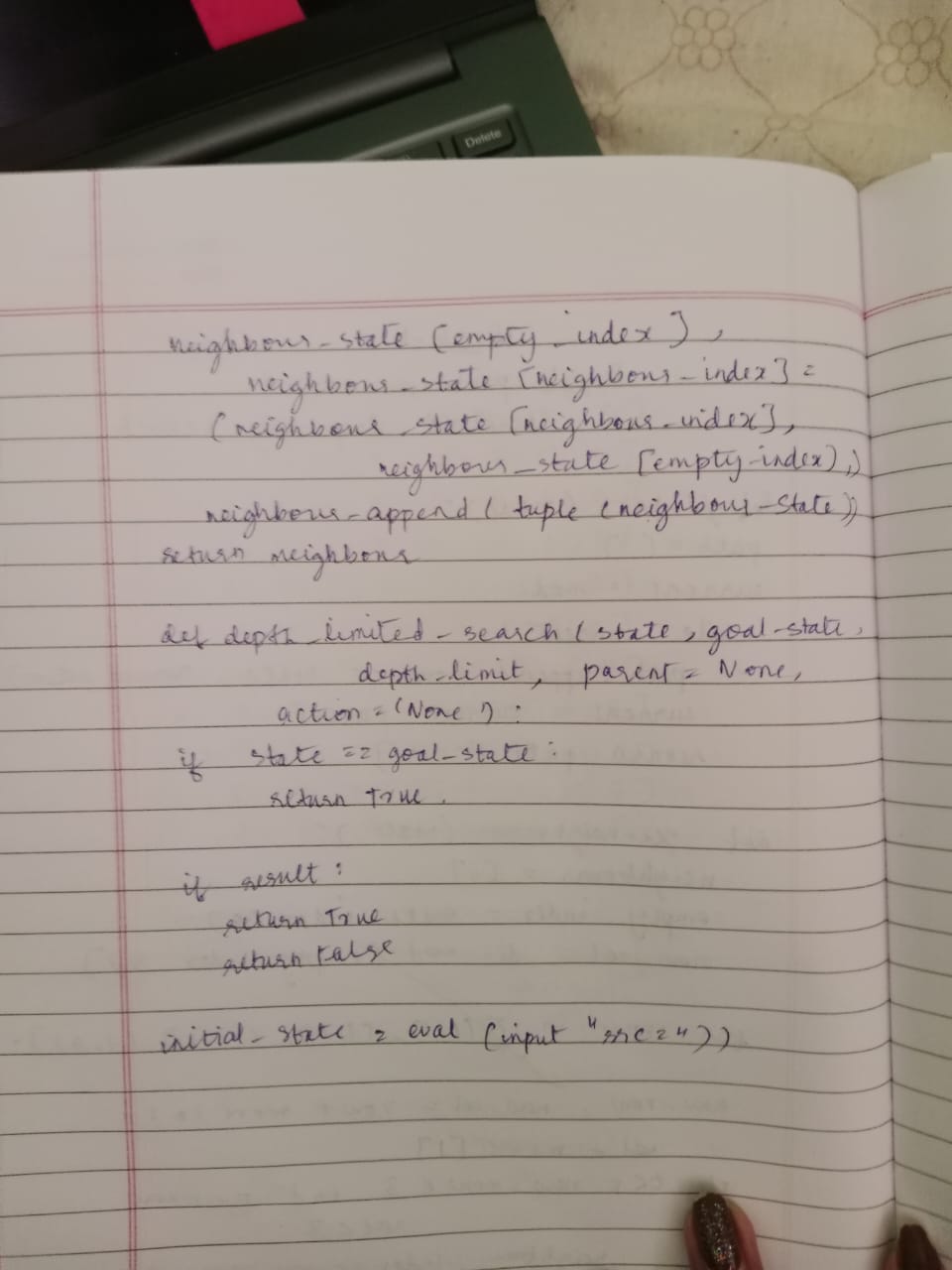
# else:

*# Print and write the final output* print\_states(path(goal)) write\_path(path(goal)) write\_node\_explored(s) write\_node\_info(v)

# Output:



**Lab-Program-3**



# 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=="down": 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=="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,"none"] stack.append(inpx) 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

return else:

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(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=="down": 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=="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,"none"] stack.append(inpx) 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

return else:

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(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])

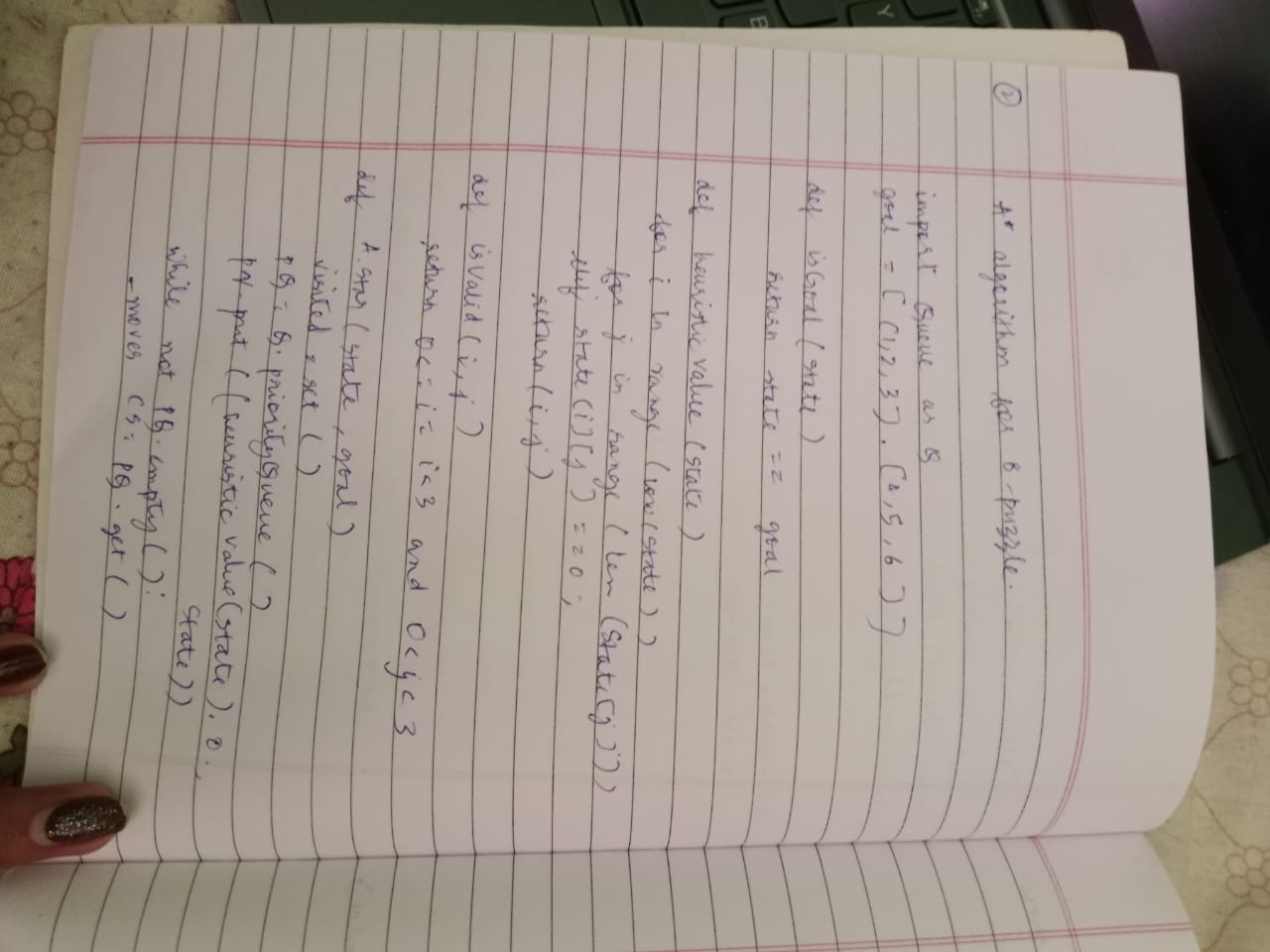
right=move(temp, "right") if(right!=puzzle[0]): rightx=[right,"right"] stack.insert(0, rightx)

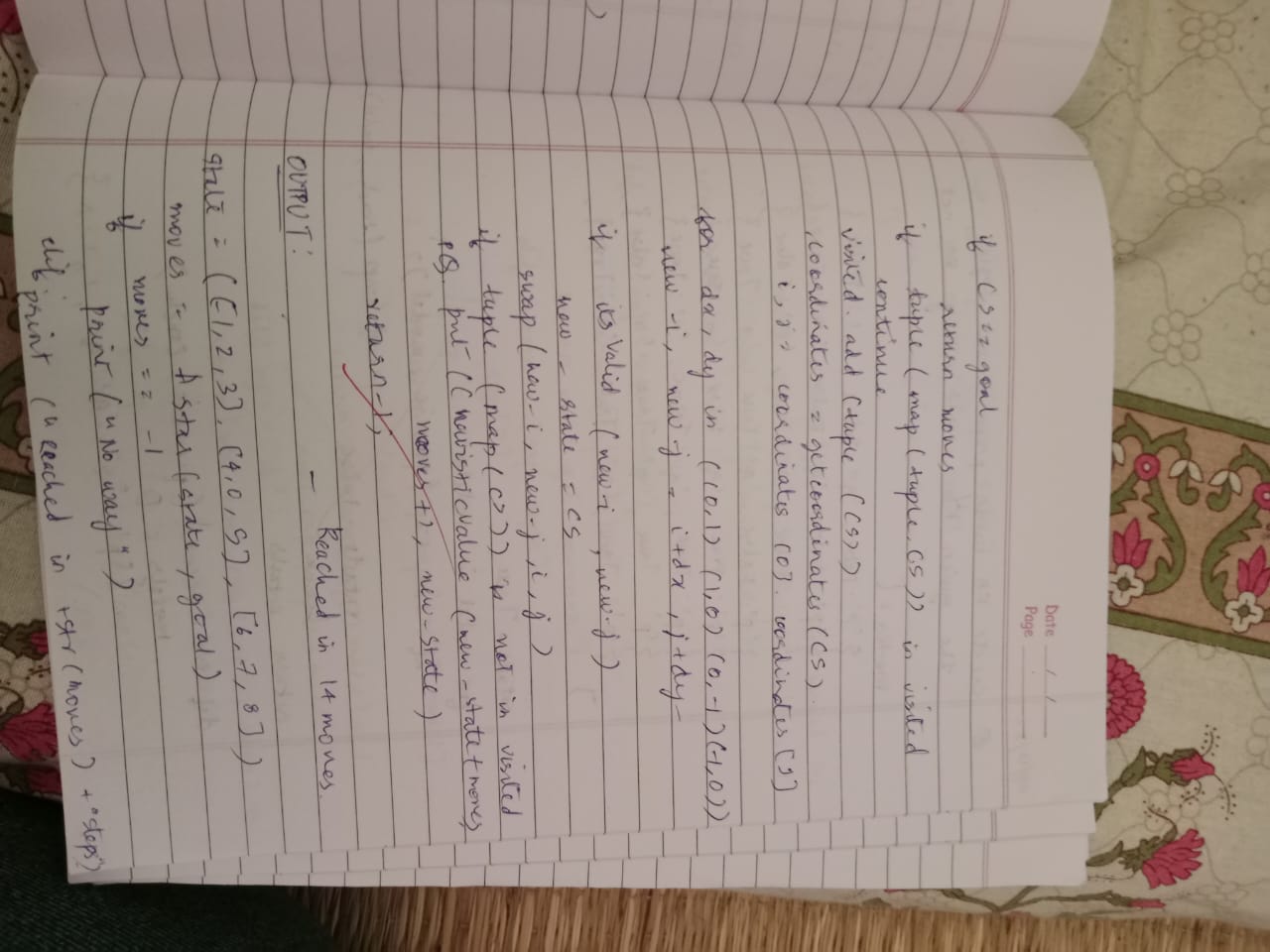
print('~~~~~~~~~~~~ IDS ~~~~~~~~~~~~') ids()

# Output:



Lab Program 4





# 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(state) print\_b(state)

if state == target:

print("Success") 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

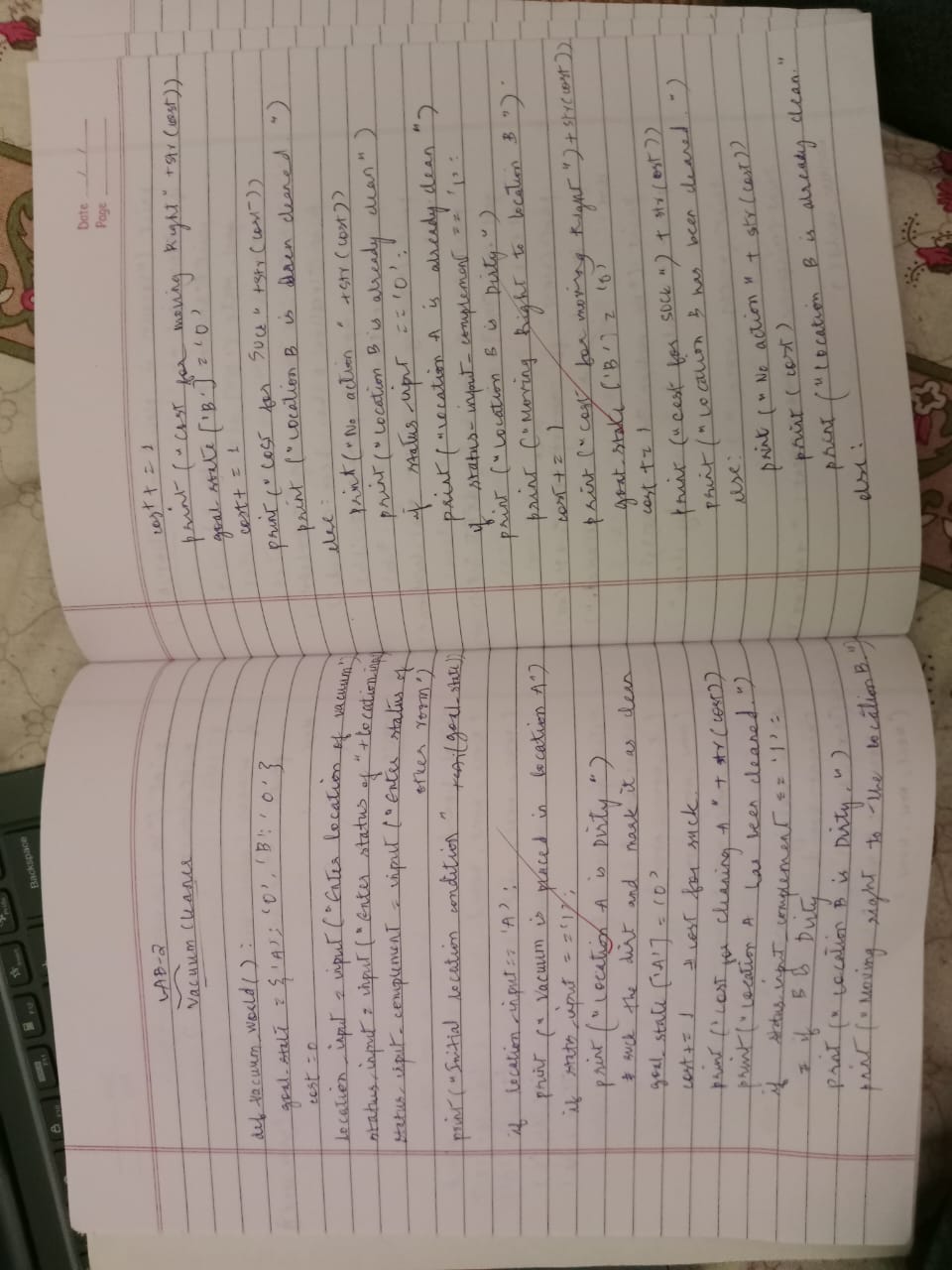
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]

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

astar(src, target)

# Output:

**Lab Program 5**

****

# 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.")

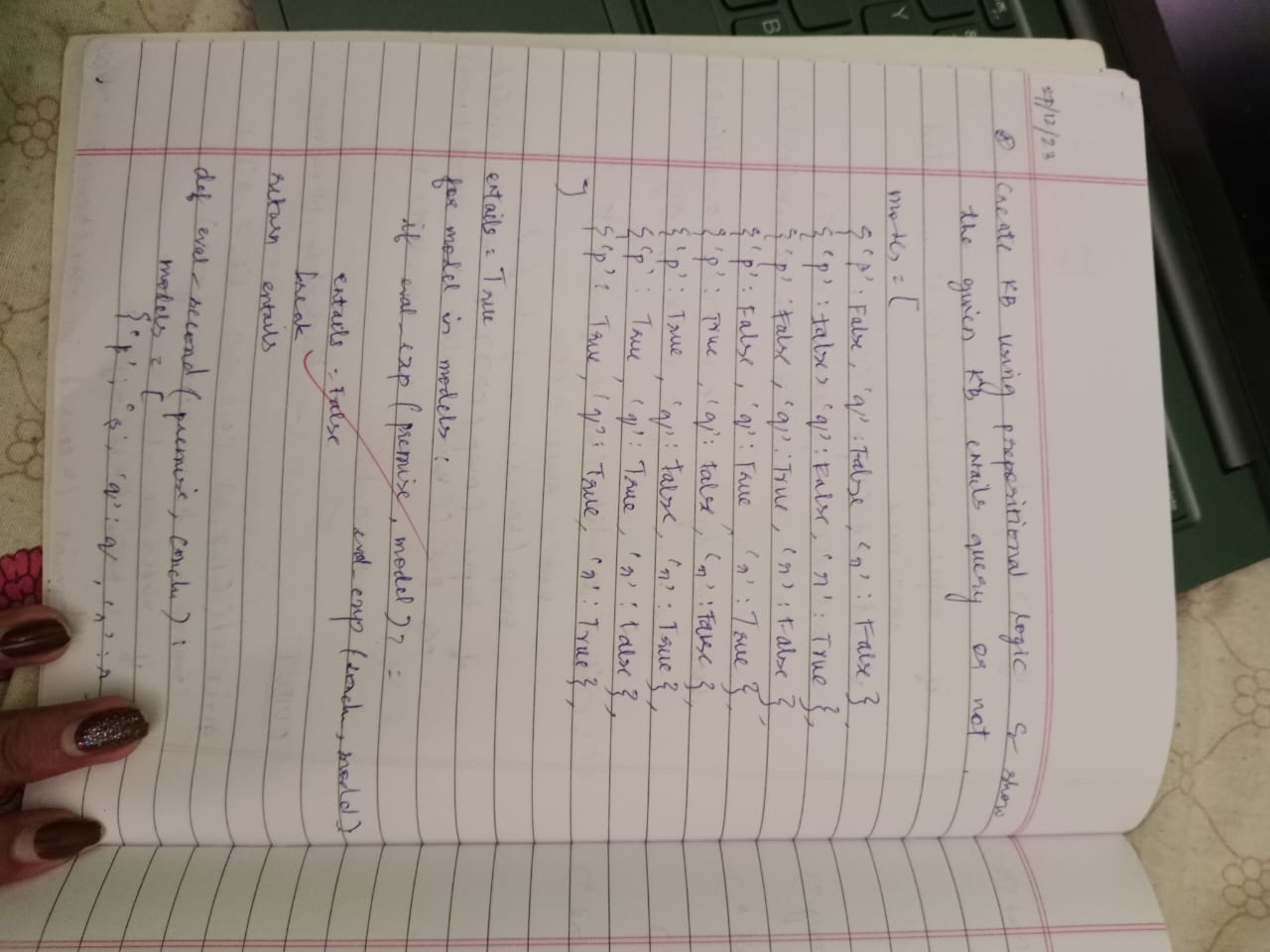
print("GOAL STATE: ")

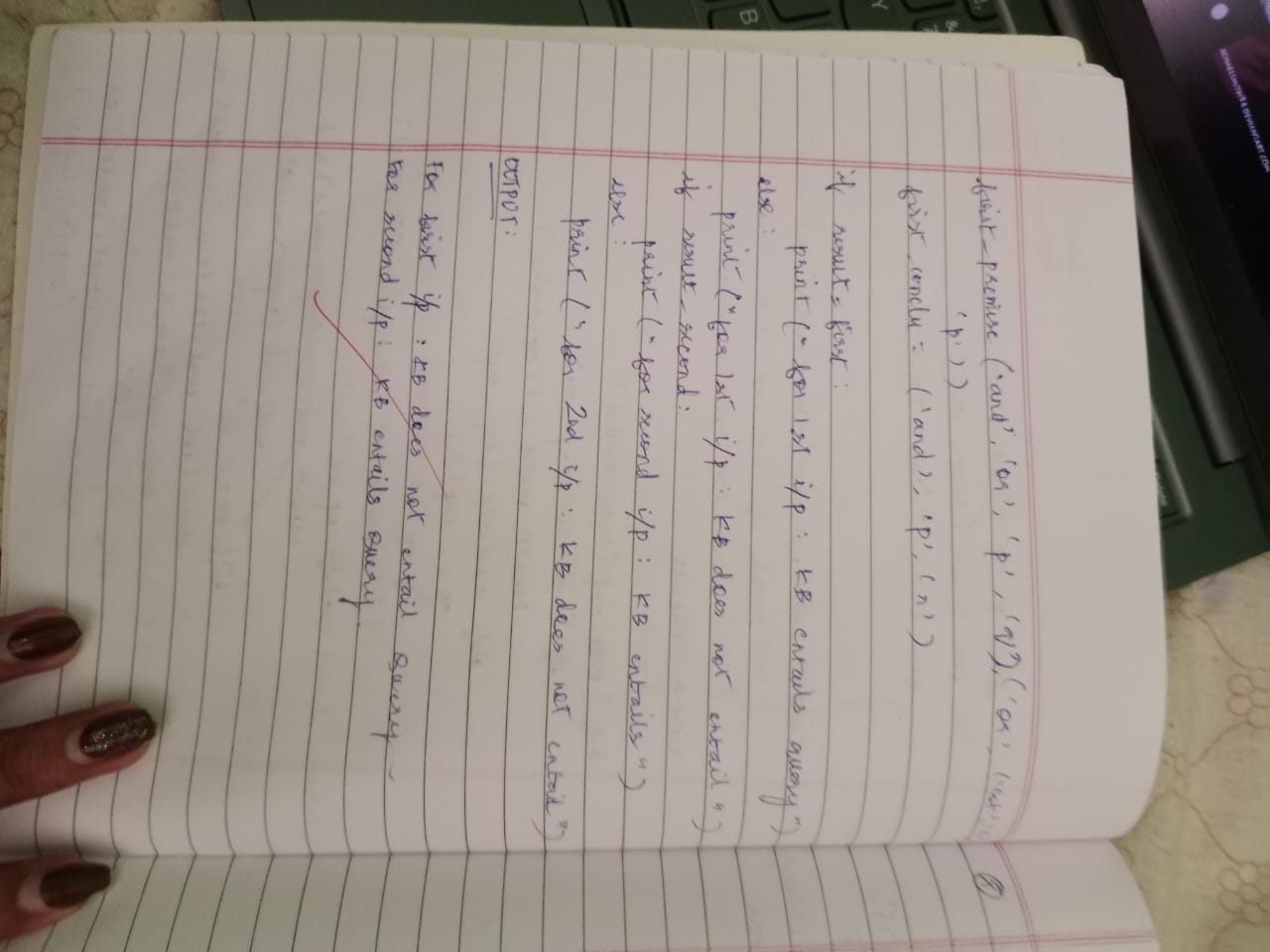
print(goal\_state)

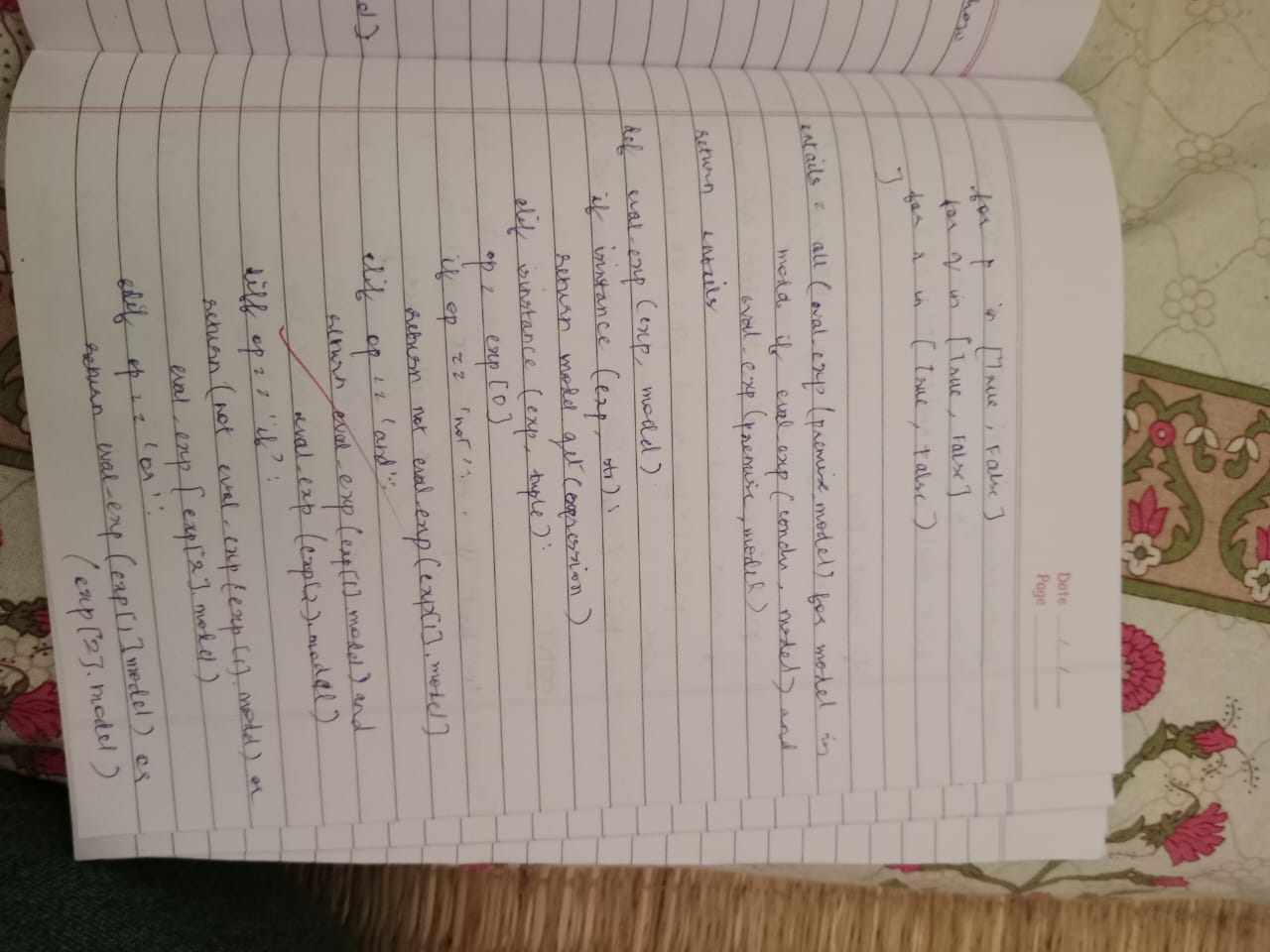
print("Performance Measurement: " + str(cost)) vacuum\_world()

# Output:

Lab Program 6







# 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

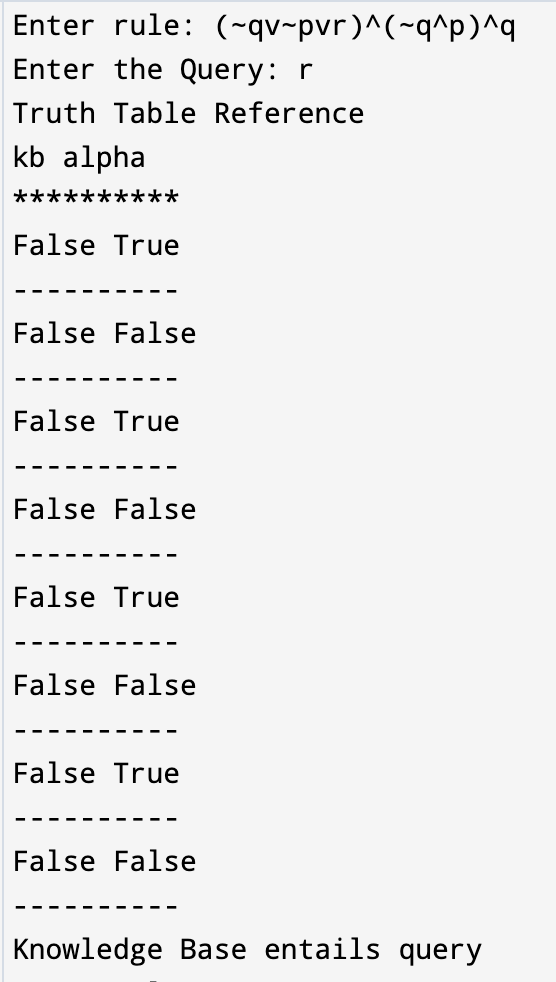
input\_rules()

ans = entailment() if ans:

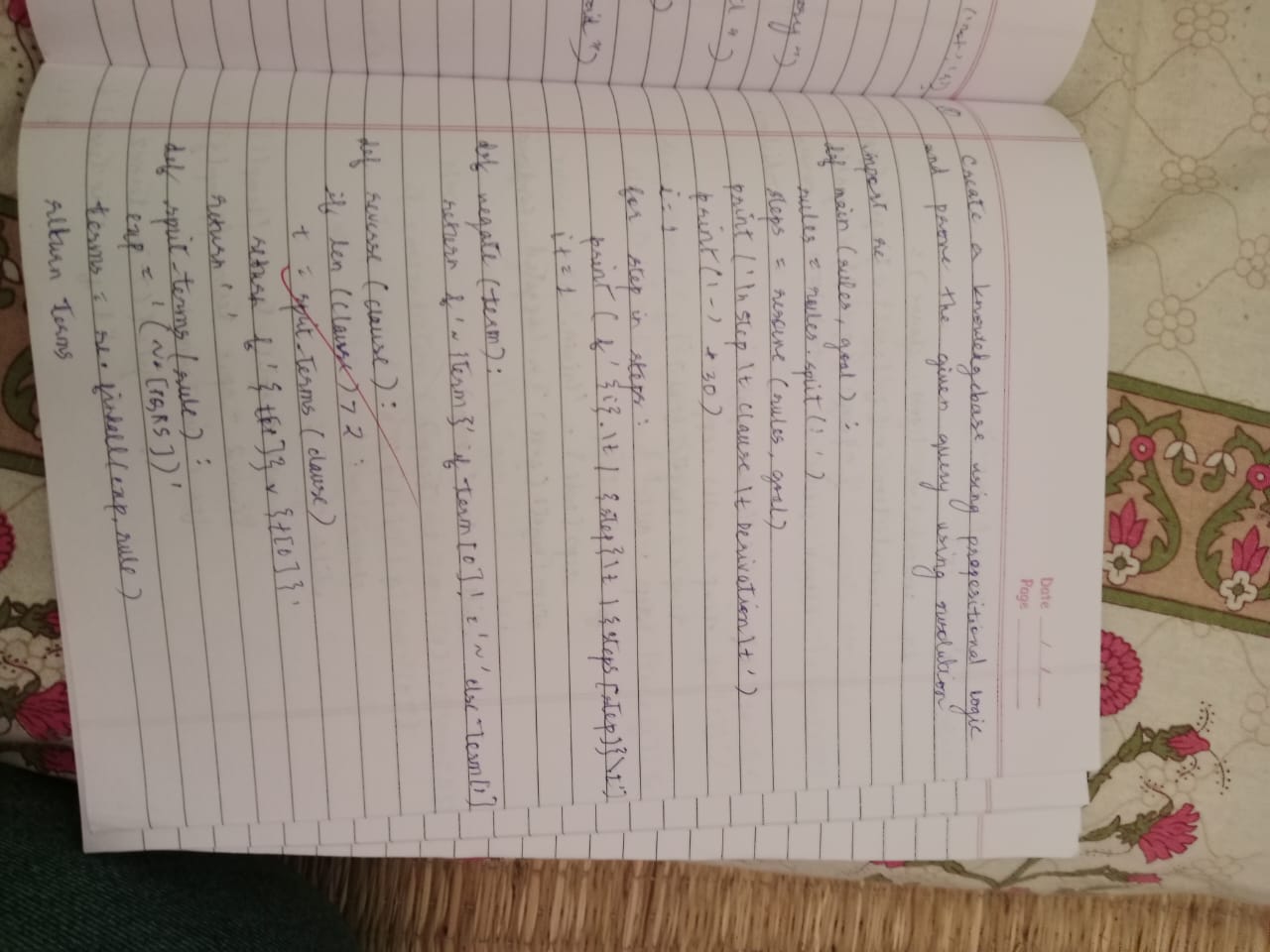
print("Knowledge Base entails query") else:

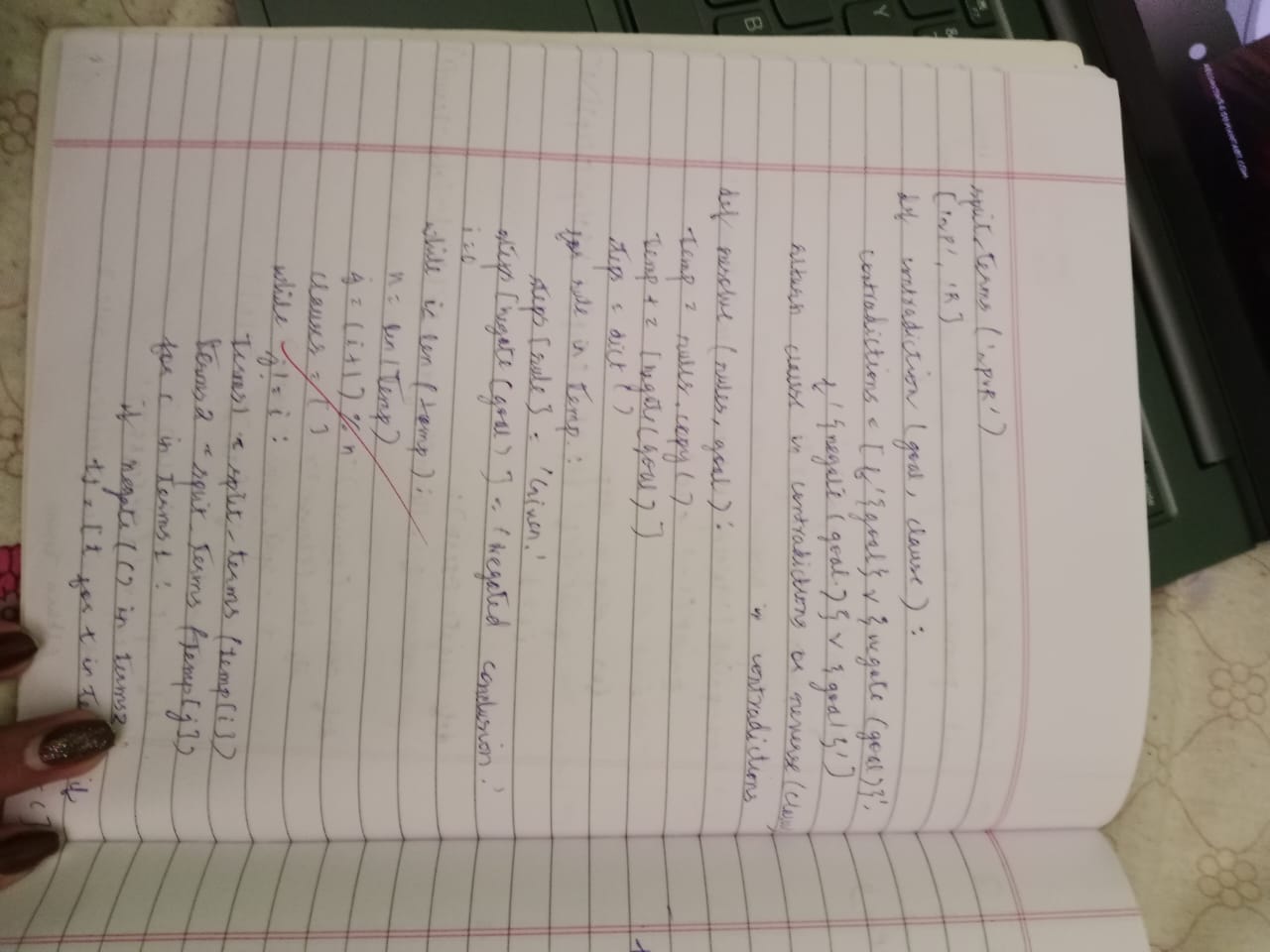
print("Knowledge Base does not entail query")

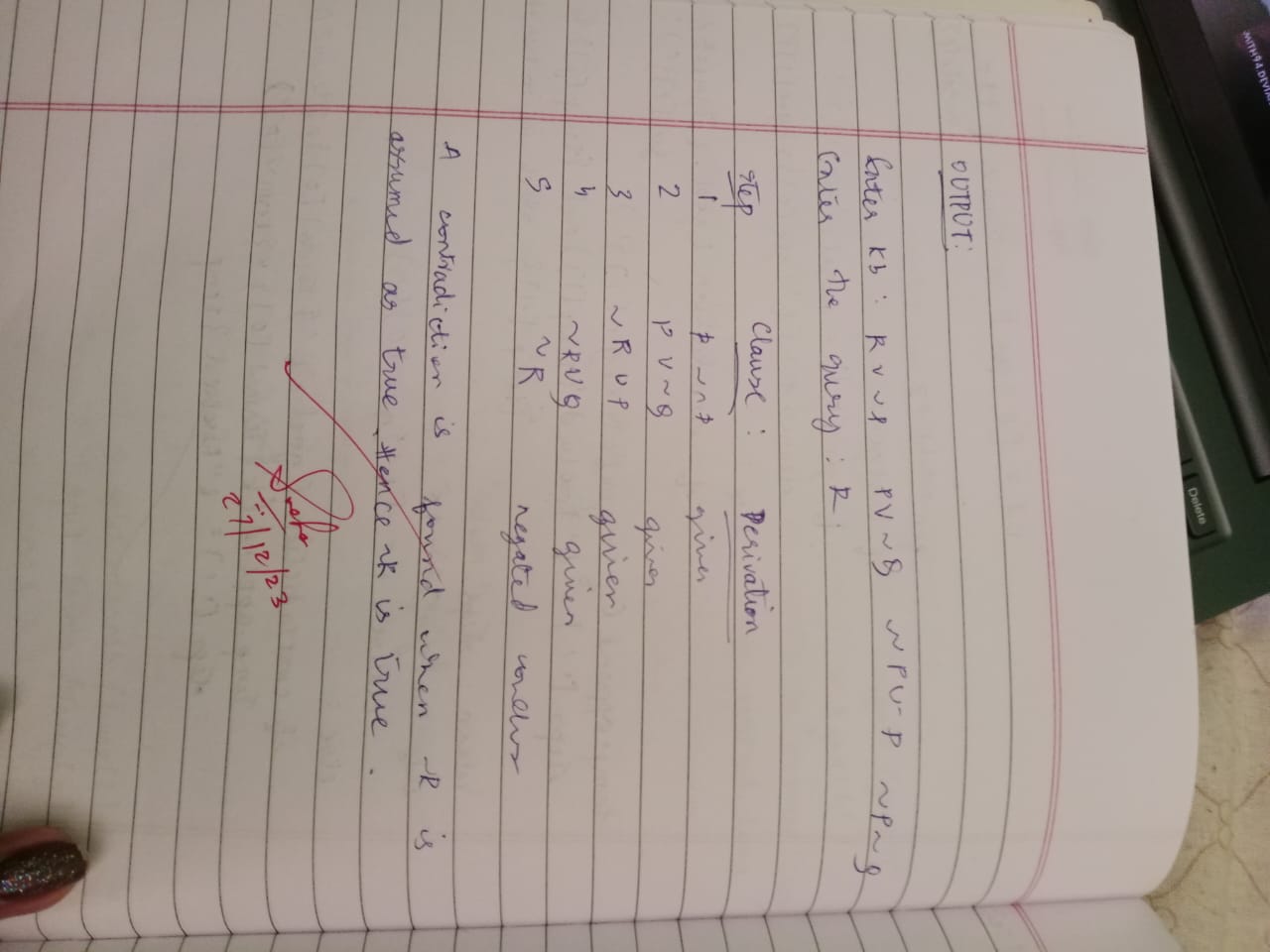
# Output:



Lab Program 7







# 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(clause)

if resolved:

print('Knowledge Base entails the query, proved by resolution') else:

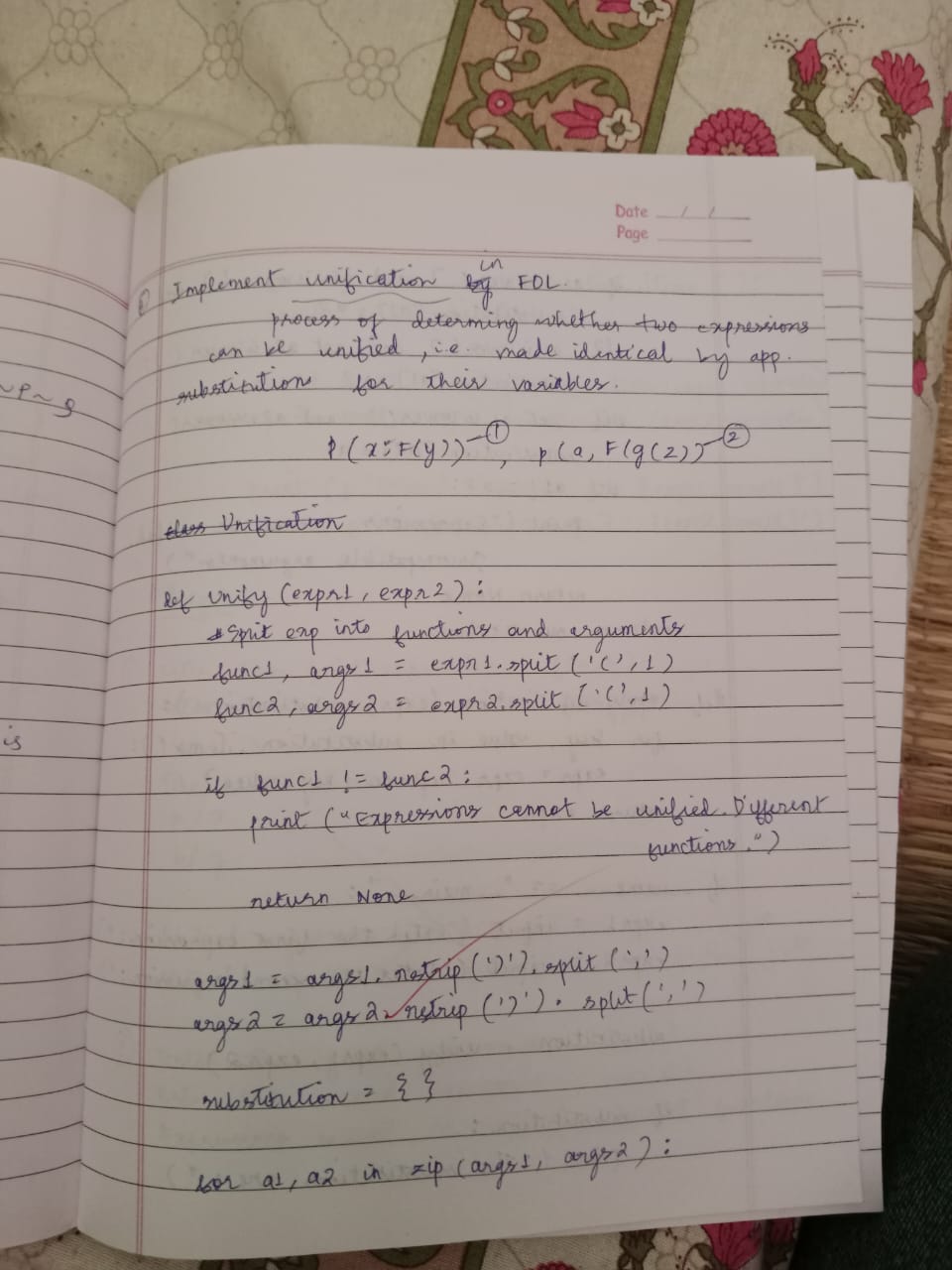
print("Knowledge Base doesn't entail the query, no empty set produced after resolution")

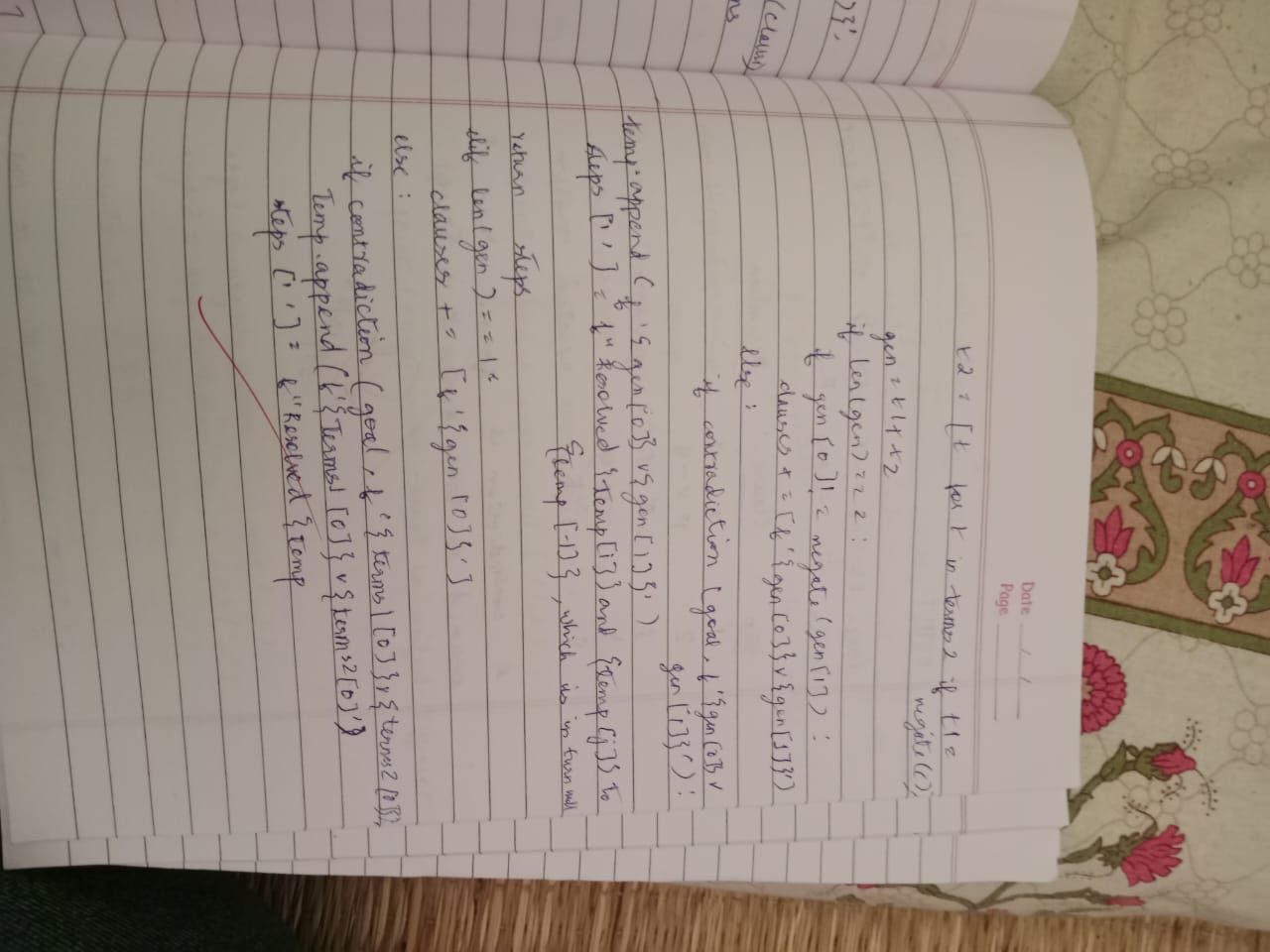
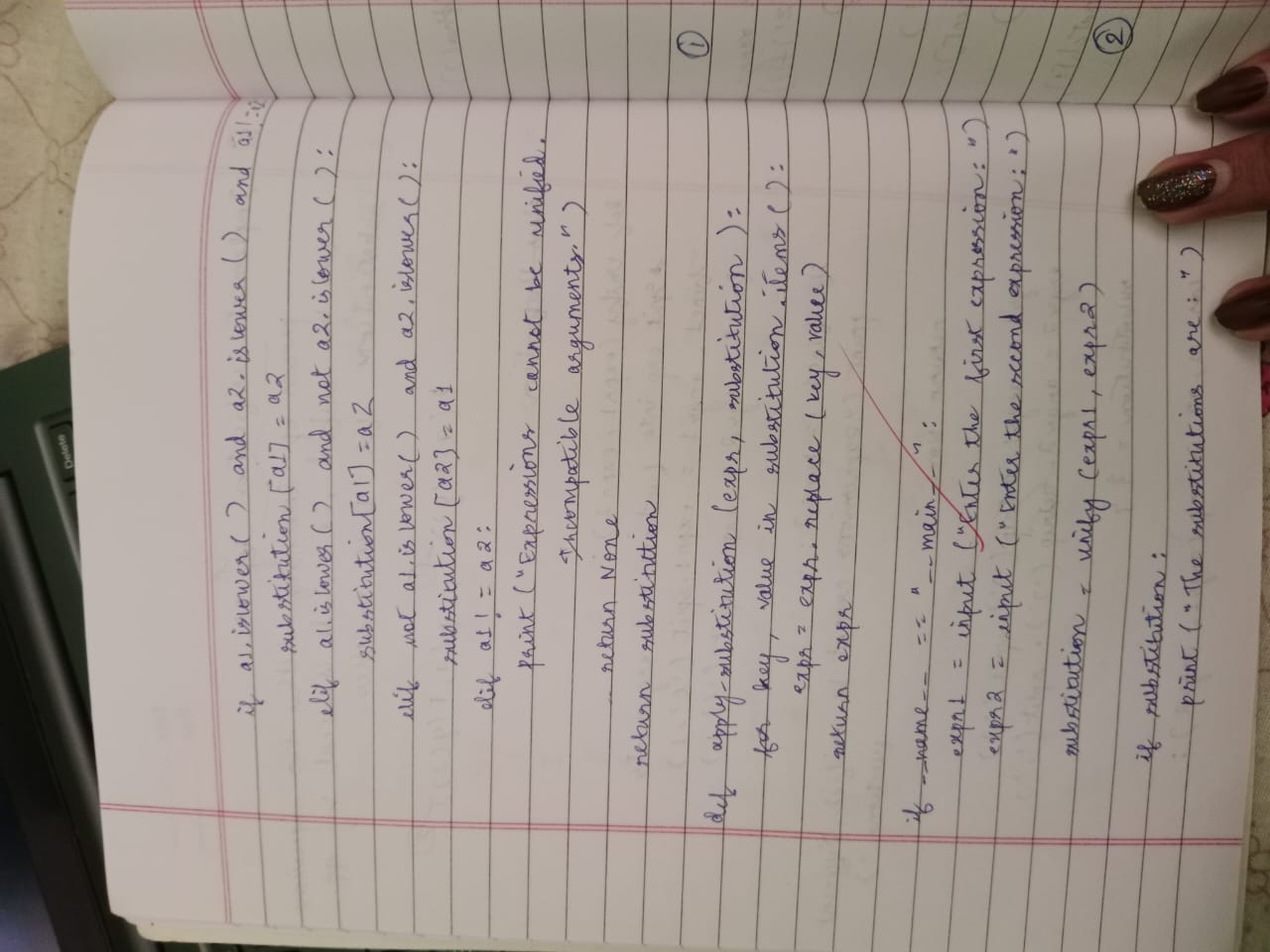
clauses = input('Enter the clauses ').split() query = input('Enter the query: ') checkResolution(clauses, query)

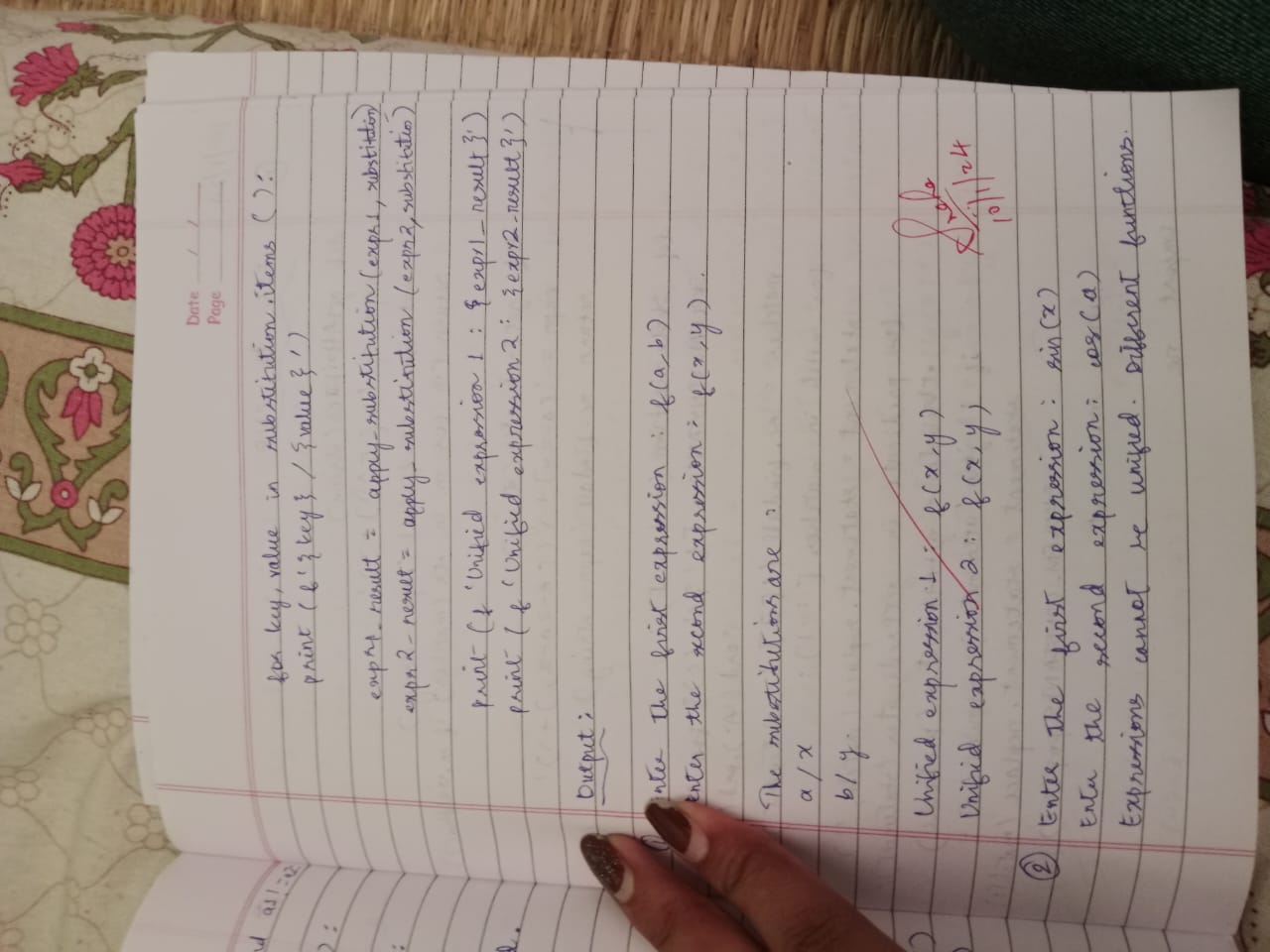
# Output:



Lab Program 8







# 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):

return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else [] if getInitialPredicate(exp1) != getInitialPredicate(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("Enter the first expression")

e1 = input()

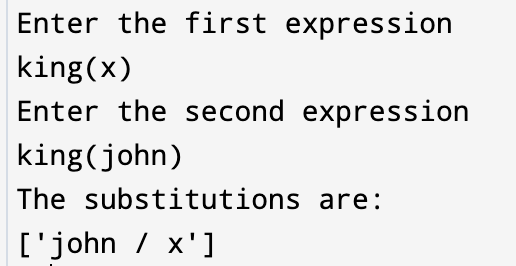
print("Enter the second expression") e2 = input()

substitutions = unify(e1, e2)

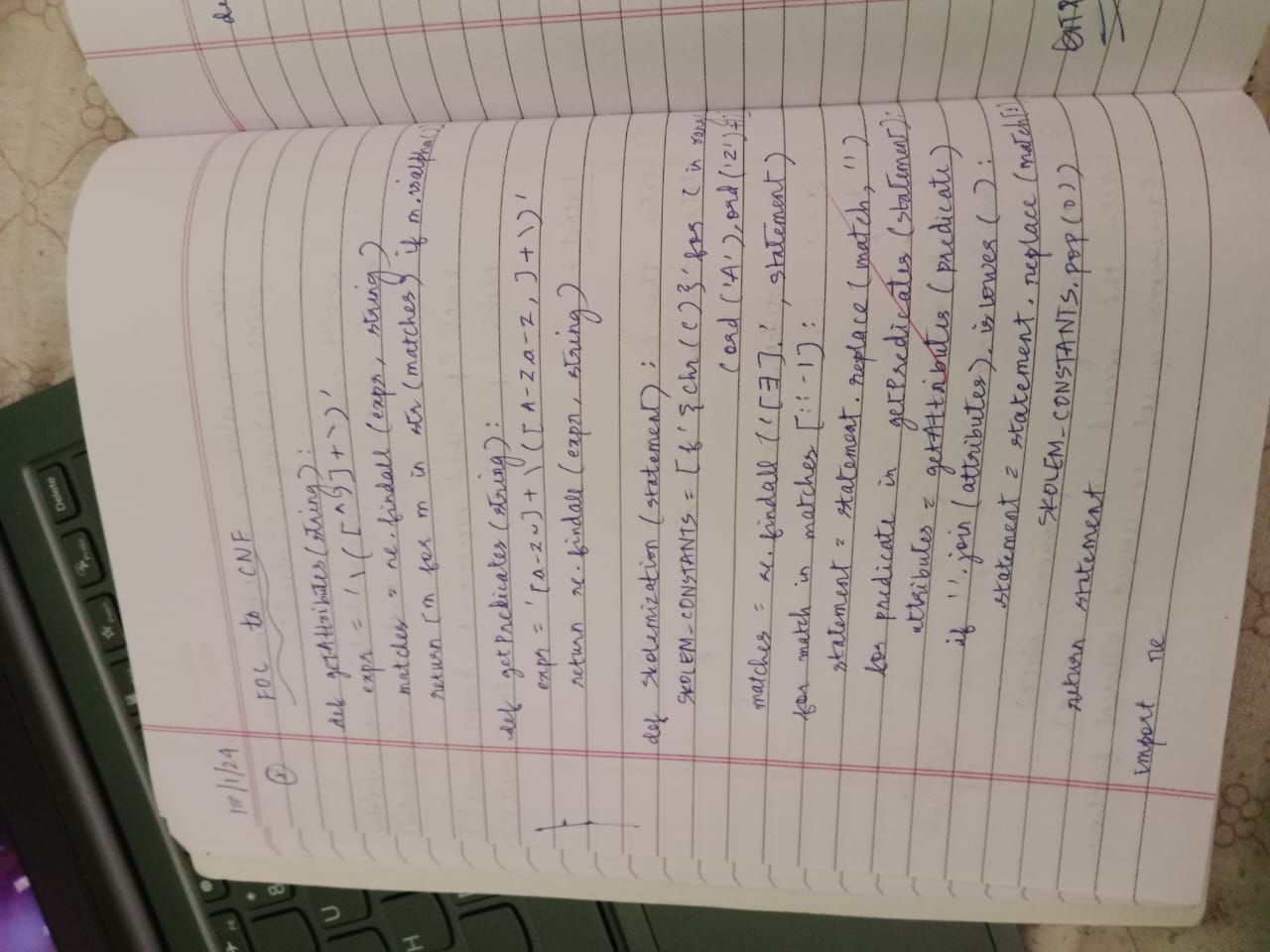
print("The substitutions are:")

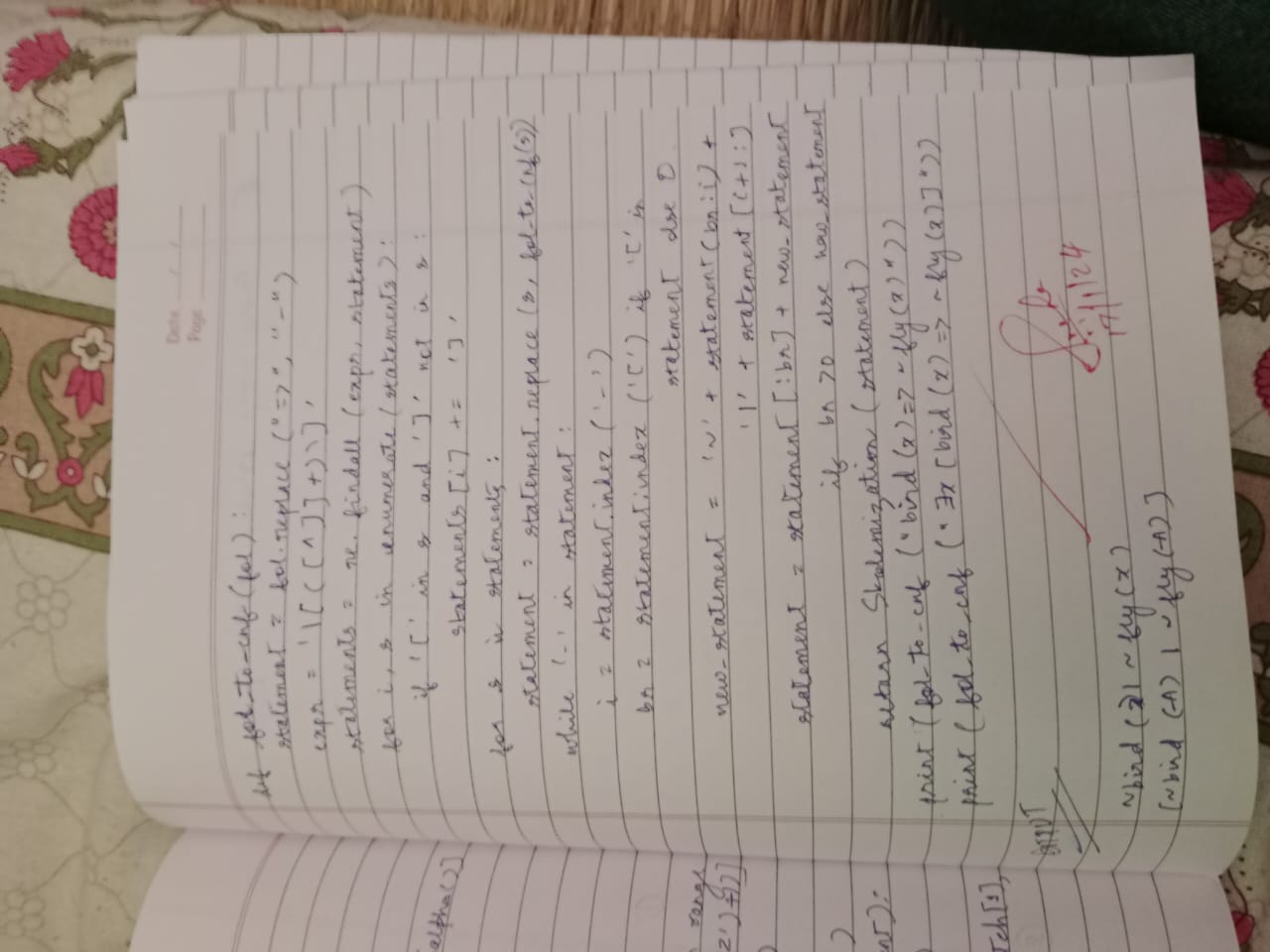
print([' / '.join(substitution) for substitution in substitutions])

# Output:



Lab Program 9





# 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] = '∃', statement[i+2], '~'

statement = ''.join(statement)

while '~∃' in statement:

i = statement.index('~∃')

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('~[∃','[~∃') expr = '(~[AV∃].)'

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:

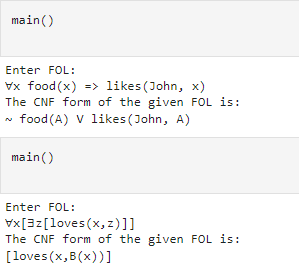
statement = statement.replace(s, DeMorgan(s)) return statement

def main(): print("Enter FOL:") fol = input()

print("The CNF form of the given FOL is: ") print(Skolemization(fol\_to\_cnf(fol)))

main()

# Output:



# 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 = 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(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}')

def main(): kb = KB()

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(fact)

print("Enter the query:") query = input() kb.ask(query) kb.display()

# Output:

