**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**ARTIFICIAL INTELLIGENCE**

***Submitted by***

**VRISHANK J VASIST(1BM21CS246)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Nov -2023 to Feb-2024**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “**Artificial Intelligence**” carried out by **VRISHANK J VASIST(1BM21CS246),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester Nov -2023 to Feb-2024.  The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

Dr Kayarvizhy N             Dr. Jyothi S Nayak

Associate Professor                             Professor and Head

Department of CSE                 Department of CSE

BMSCE, Bengaluru                 BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Lab Program No.** | **PROGRAM** | **Page No.** |
| 1 | Implement Tic –Tac –Toe Game | 5 |
| 2 | Implement vaccum cleaner agent | 10 |
| 3 | Analyse 8 Puzzle problem and implement the same using Breadth First Search Algorithm | 14 |
| 4 | Analyse Iterative Deepening Search Algorithm. Demonstrate how 8 Puzzle problem could be solved using this algorithm | 18 |
| 5 | Implement A\* search algorithm | 21 |
| 6 | Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not | 26 |
| 7 | Create a knowledge base using prepositional logic and prove the given query using resolution | 30 |
| 8 | Implement unification in first order logic | 34 |
| 9 | Convert a given first order logic statement into Conjunctive Normal Form (CNF) | 39 |
| 10 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning | 44 |

**Course Outcome**

|  |  |
| --- | --- |
| CO1 | Apply knowledge of agent architecture, searching and reasoning techniques for different applications. |
| CO2 | Analyse Searching and Inferencing Techniques. |
| CO3 | Design a reasoning system for a given requirement. |
| CO4 | Conduct practical experiments for demonstrating agents, searching and inferencing. |

1. **Implement Tic –Tac –Toe Game**

import math

def print\_board(board):

    for i in range(len(board)):

        for j in range(len(board[i])):

            print(board[i][j], end='')

            if j < len(board[i]) - 1:

                print('|', end='')

        print()

        if i < len(board) - 1:

            print('-'\*5)

    print()

def check\_winner(board):

    # Check rows, columns, and diagonals for a winner

    for i in range(3):

        if board[i][0] == board[i][1] == board[i][2] != ' ':

            return board[i][0]

        if board[0][i] == board[1][i] == board[2][i] != ' ':

            return board[0][i]

    if board[0][0] == board[1][1] == board[2][2] != ' ':

        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] != ' ':

        return board[0][2]

    return None

def get\_empty\_cells(board):

    # Returns a list of empty cells in the board

    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def minimax(board, depth, is\_maximizing):

    winner = check\_winner(board)

    if winner:

        return 10 - depth if winner == 'X' else -10 + depth

    elif not get\_empty\_cells(board):

        return 0

    if is\_maximizing:

        best\_score = -math.inf

        for i, j in get\_empty\_cells(board):

            board[i][j] = 'X'

            score = minimax(board, depth + 1, False)

            board[i][j] = ' '

            best\_score = max(score, best\_score)

        return best\_score

    else:

        best\_score = math.inf

        for i, j in get\_empty\_cells(board):

            board[i][j] = 'O'

            score = minimax(board, depth + 1, True)

            board[i][j] = ' '

            best\_score = min(score, best\_score)

        return best\_score

def best\_move(board):

    best\_score = -math.inf

    move = None

    for i, j in get\_empty\_cells(board):

        board[i][j] = 'X'

        score = minimax(board, 0, False)

        board[i][j] = ' '

        if score > best\_score:

            best\_score = score

            move = (i, j)

    return move

def play\_game():

    board = [[' ' for \_ in range(3)] for \_ in range(3)]

    print("Welcome to Tic Tac Toe!")

    print\_board(board)

    while not check\_winner(board) and get\_empty\_cells(board):

        user\_move = input("Enter your move (row and column separated by a space): ")

        x, y = map(int, user\_move.split())

        if board[x][y] == ' ':

            board[x][y] = 'O'

            print\_board(board)

        else:

            print("Invalid move. Try again.")

            continue

        if not get\_empty\_cells(board):

            break

        computer\_move = best\_move(board)

        board[computer\_move[0]][computer\_move[1]] = 'X'

        print("Computer's move:")

        print\_board(board)

    winner = check\_winner(board)

    if winner:

        print(f"Player {winner} wins!")

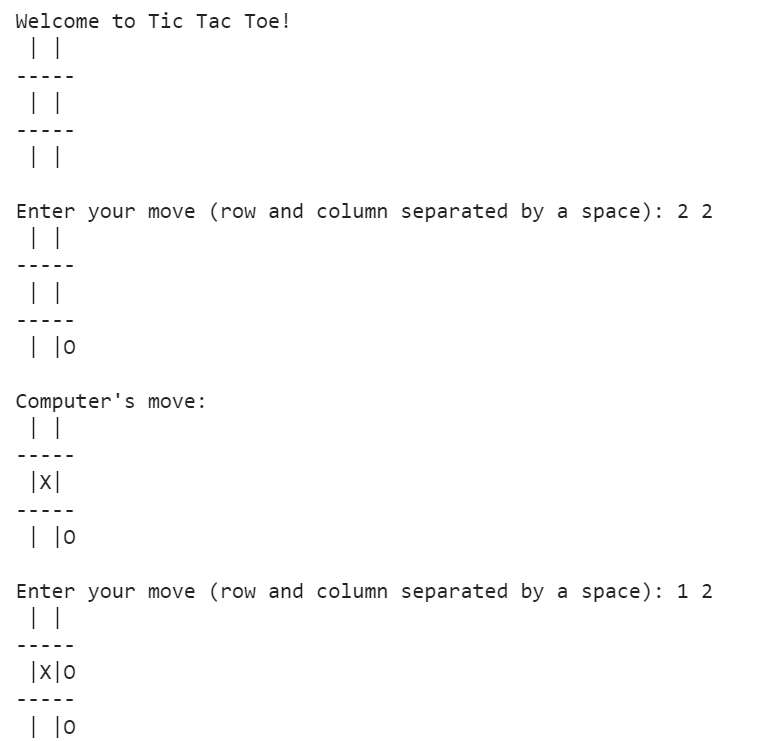
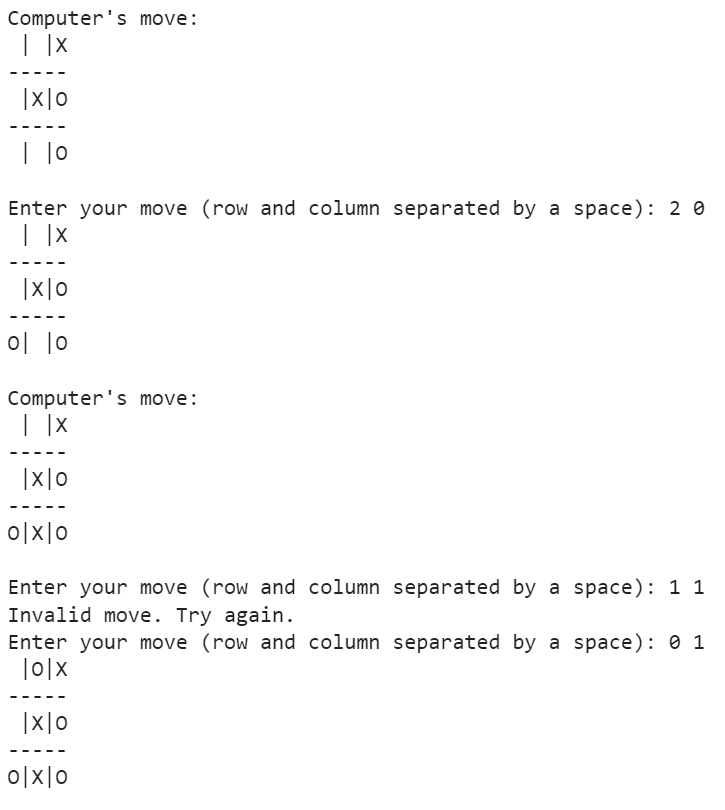
    else:

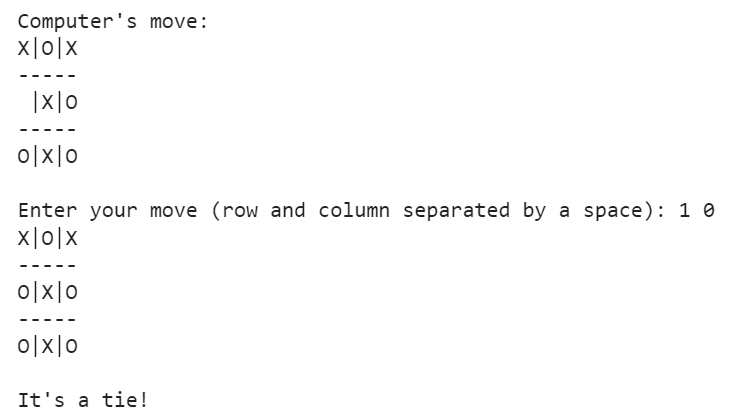
        print("It's a tie!")

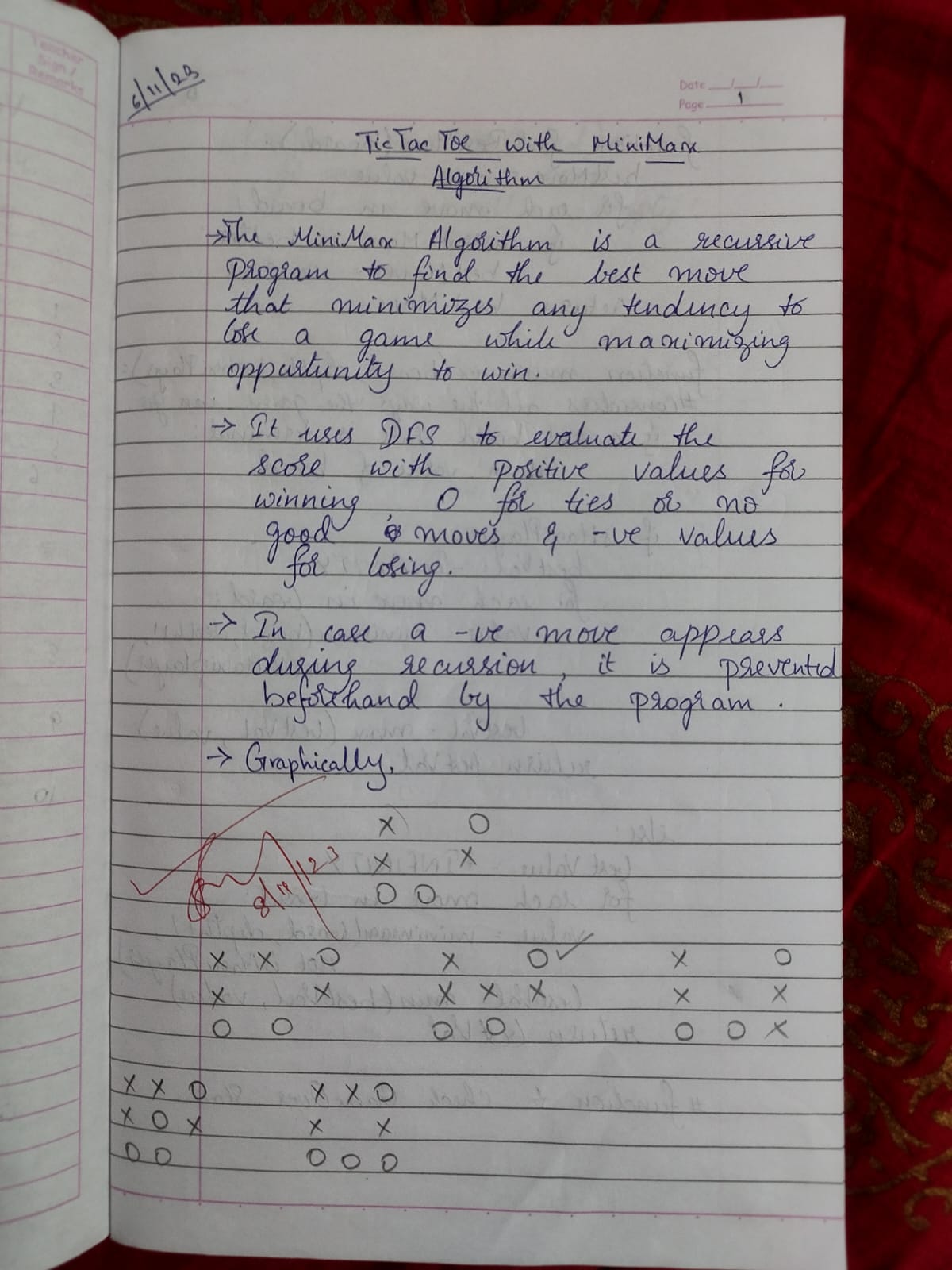
if \_\_name\_\_ == "\_\_main\_\_":

    play\_game()

OUTPUT





1. **Implement vaccum cleaner agent**

def printInformation(location):

    print("Location " + location + " is Dirty.")

    print("Cost for CLEANING " + location + ": 1")

    print("Location " + location + " has been Cleaned.")

def vacuumCleaner(goalState, currentState, location):

    # printing necessary data

    print("Goal State Required:", goalState)

    print("Vacuum is placed in Location " + location)

    # cleaning locations

    totalCost = 0

    while (currentState != goalState):

        if (location == "A"):

            # cleaning

            if (currentState["A"] == 1):

                currentState["A"] = 0

                totalCost += 1

                printInformation("A")

            # moving

            elif (currentState["B"] == 1 ):

                print("Moving right to the location B.\nCost for moving RIGHT: 1")

                location = "B"

                totalCost += 1

        elif (location == "B"):

            # cleaning

            if (currentState["B"] == 1):

                currentState["B"] = 0

                totalCost += 1

                printInformation("B")

            # moving

            elif (currentState["A"] == 1):

                print("Moving left to the location A.\nCost for moving LEFT: 1")

                location = "A"

                totalCost += 1

    print("GOAL STATE:", currentState)

    return totalCost

# declaring dictionaries

goalState = {"A": 0, "B": 0}

currentState = {"A": -1, "B": -1}

# taking input from user

location = input("Enter Location of Vacuum (A/B): ");

currentState["A"] = int(input("Enter status of A (0/1): "))

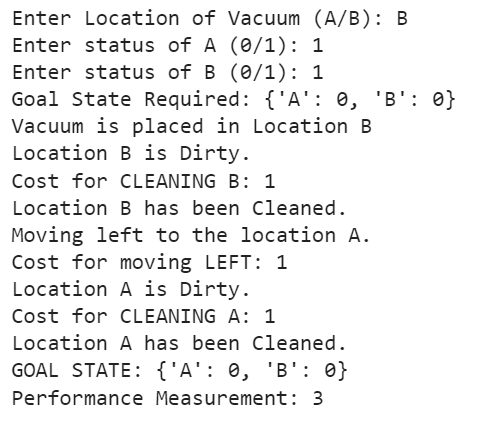
currentState["B"] = int(input("Enter status of B (0/1): "))

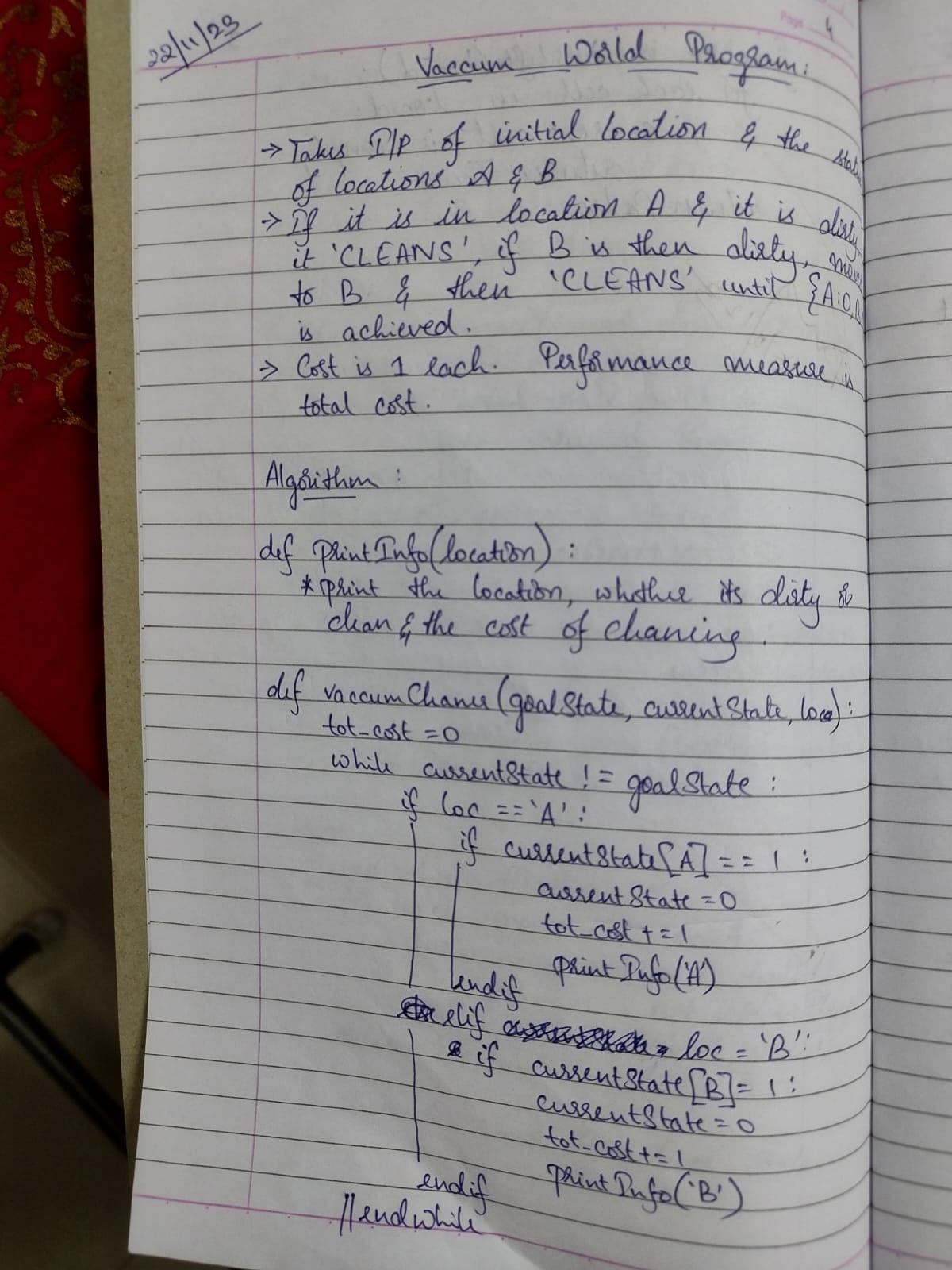
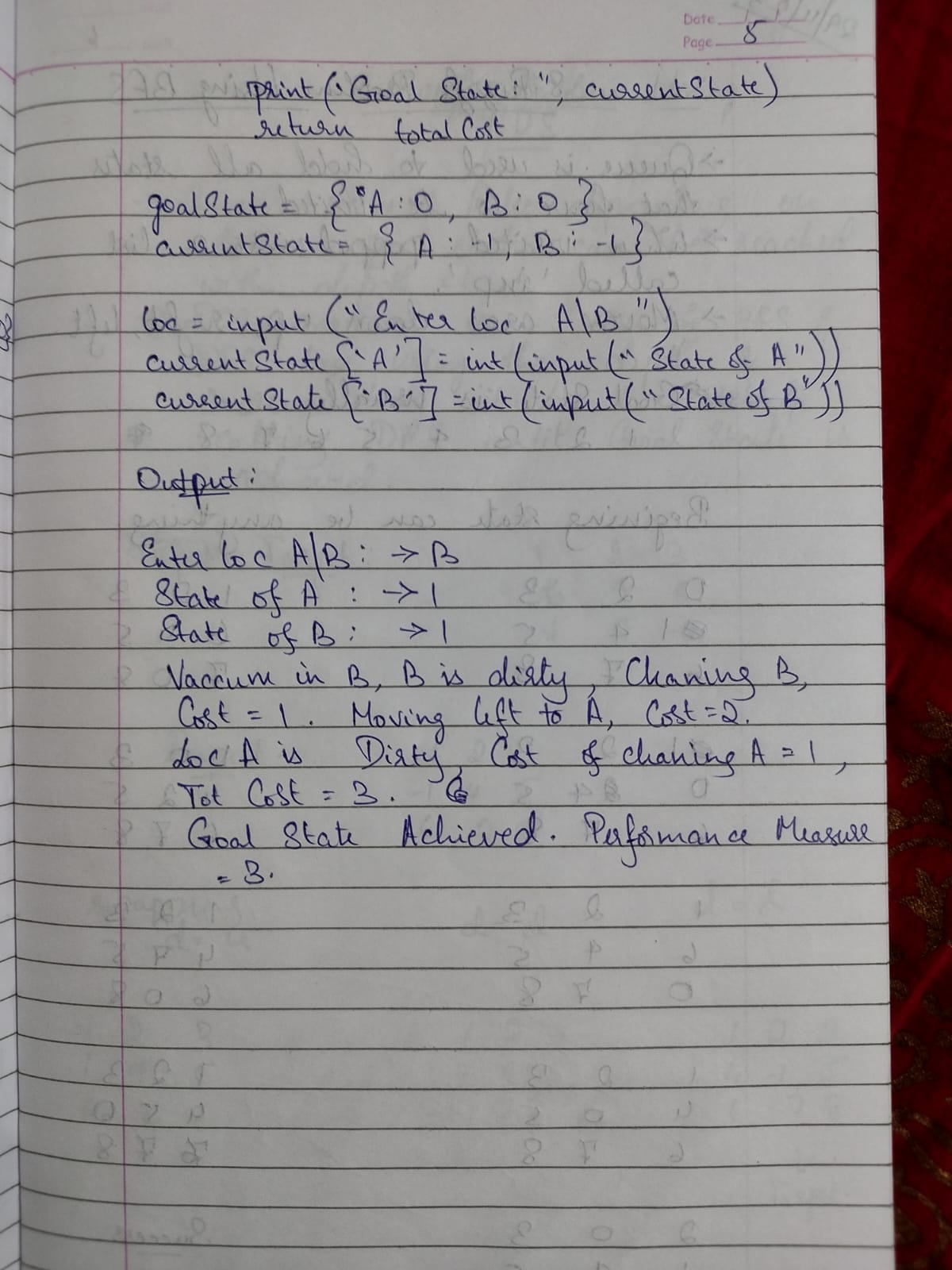
# calling function

totalCost = vacuumCleaner(goalState, currentState, location)

print("Performance Measurement:", totalCost)

OUTPUT



1. **Analyse 8 Puzzle problem and implement the same using Breadth First Search Algorithm**

def bfs(src, target):

    queue = []

    queue.append(src)

    visited = set()

    while queue:

        source = queue.pop(0)

        visited.add(tuple(source))  # Store visited states as tuples for faster lookup

        print(source[0], '|', source[1], '|', source[2])

        print(source[3], '|', source[4], '|', source[5])

        print(source[6], '|', source[7], '|', source[8])

        print("-----------")

        if source == target:

            print("Success")

            return

        poss\_moves\_to\_do = possible\_moves(source, visited)

        for move in poss\_moves\_to\_do:

            queue.append(move)

def possible\_moves(state, visited\_states):

    b = state.index(0)

    d = []

    # Add possible directions to move based on the position of the empty cell

    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 i in d:

        pos\_moves\_it\_can.append(gen(state, i, b))

    # Return possible moves that have not been visited yet

    return [move\_it\_can for move\_it\_can in pos\_moves\_it\_can if tuple(move\_it\_can) not in visited\_states]

def gen(state, move, b):

    temp = state.copy()

    if move == 'd':

        temp[b + 3], temp[b] = temp[b], temp[b + 3]

    if move == 'u':

        temp[b - 3], temp[b] = temp[b], temp[b - 3]

    if move == 'l':

        temp[b - 1], temp[b] = temp[b], temp[b - 1]

    if move == 'r':

        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    return temp

# Taking input for initial and goal states

print("Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):")

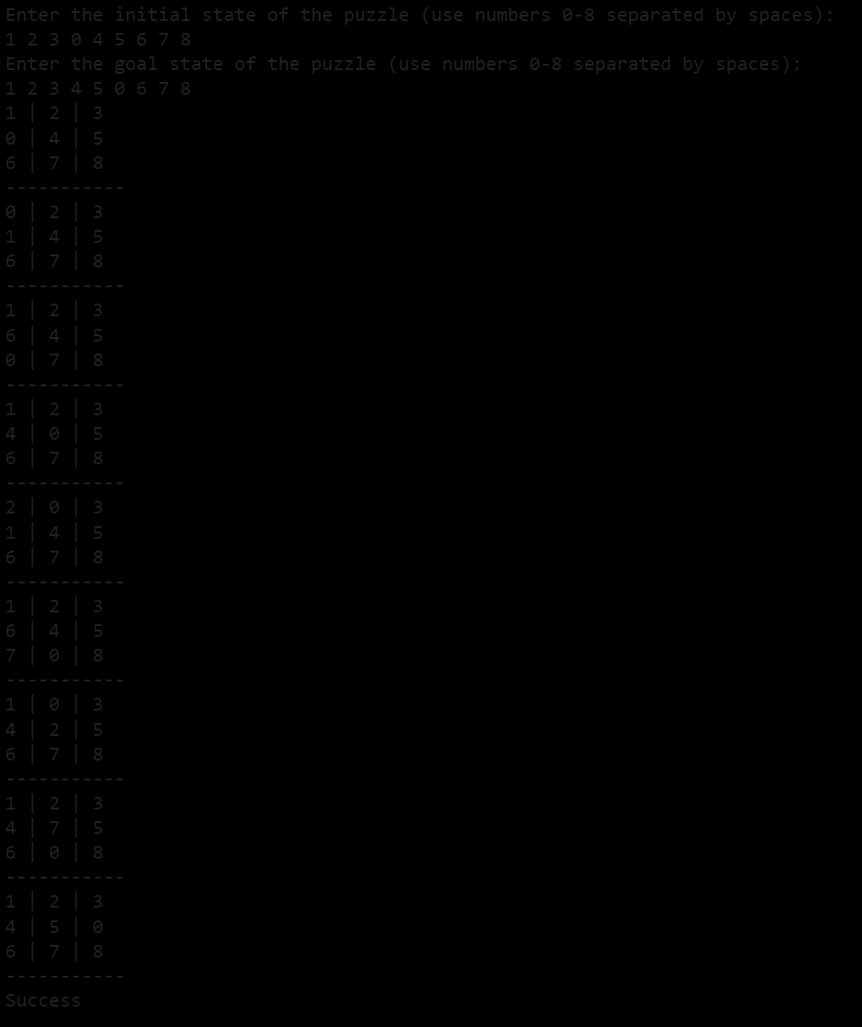
src = list(map(int, input().split()))

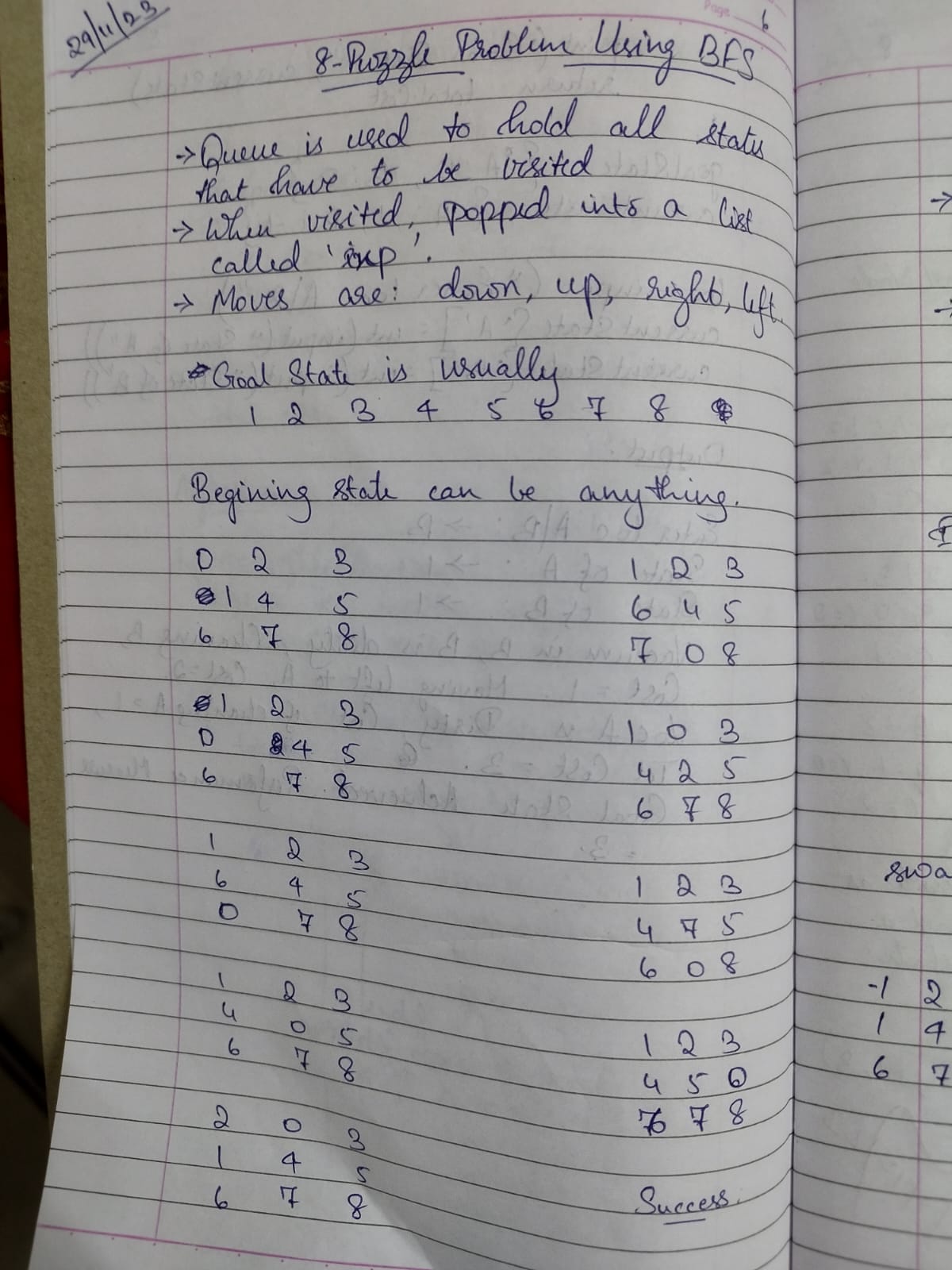
print("Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):")

target = list(map(int, input().split()))

bfs(src, target)

OUTPUT





1. **Analyse Iterative Deepening Search Algorithm. Demonstrate how 8 Puzzle problem could be solved using this algorithm**

def dfs(src,target,limit,visited\_states):

    if src == target:

        return True

    if limit <= 0:

        return False

    visited\_states.append(src)

    moves = possible\_moves(src,visited\_states)

    for move in moves:

        if dfs(move, target, limit-1, visited\_states):

            return True

    return False

def possible\_moves(state,visited\_states):

    b = state.index(-1)

    d = []

    if b not in [0,1,2]:

        d += 'u'

    if b not in [6,7,8]:

        d += 'd'

    if b not in [2,5,8]:

        d += 'r'

    if b not in [0,3,6]:

        d += 'l'

    pos\_moves = []

    for move in d:

        pos\_moves.append(gen(state,move,b))

    return [move for move in pos\_moves if move not in visited\_states]

def gen(state, move, blank):

    temp = state.copy()

    if move == 'u':

        temp[blank-3], temp[blank] = temp[blank], temp[blank-3]

    if move == 'd':

        temp[blank+3], temp[blank] = temp[blank], temp[blank+3]

    if move == 'r':

        temp[blank+1], temp[blank] = temp[blank], temp[blank+1]

    if move == 'l':

        temp[blank-1], temp[blank] = temp[blank], temp[blank-1]

    return temp

def iddfs(src,target,depth):

    for i in range(depth):

        visited\_states = []

        if dfs(src,target,i+1,visited\_states):

            return True, i+1

    return False

print("Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):")

src = list(map(int, input().split()))

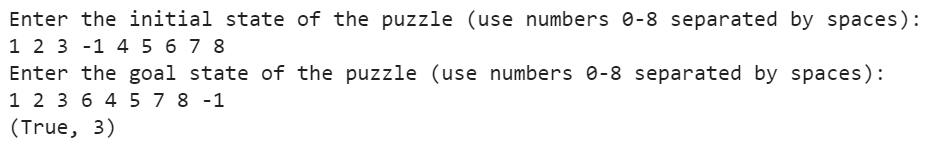
print("Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):")

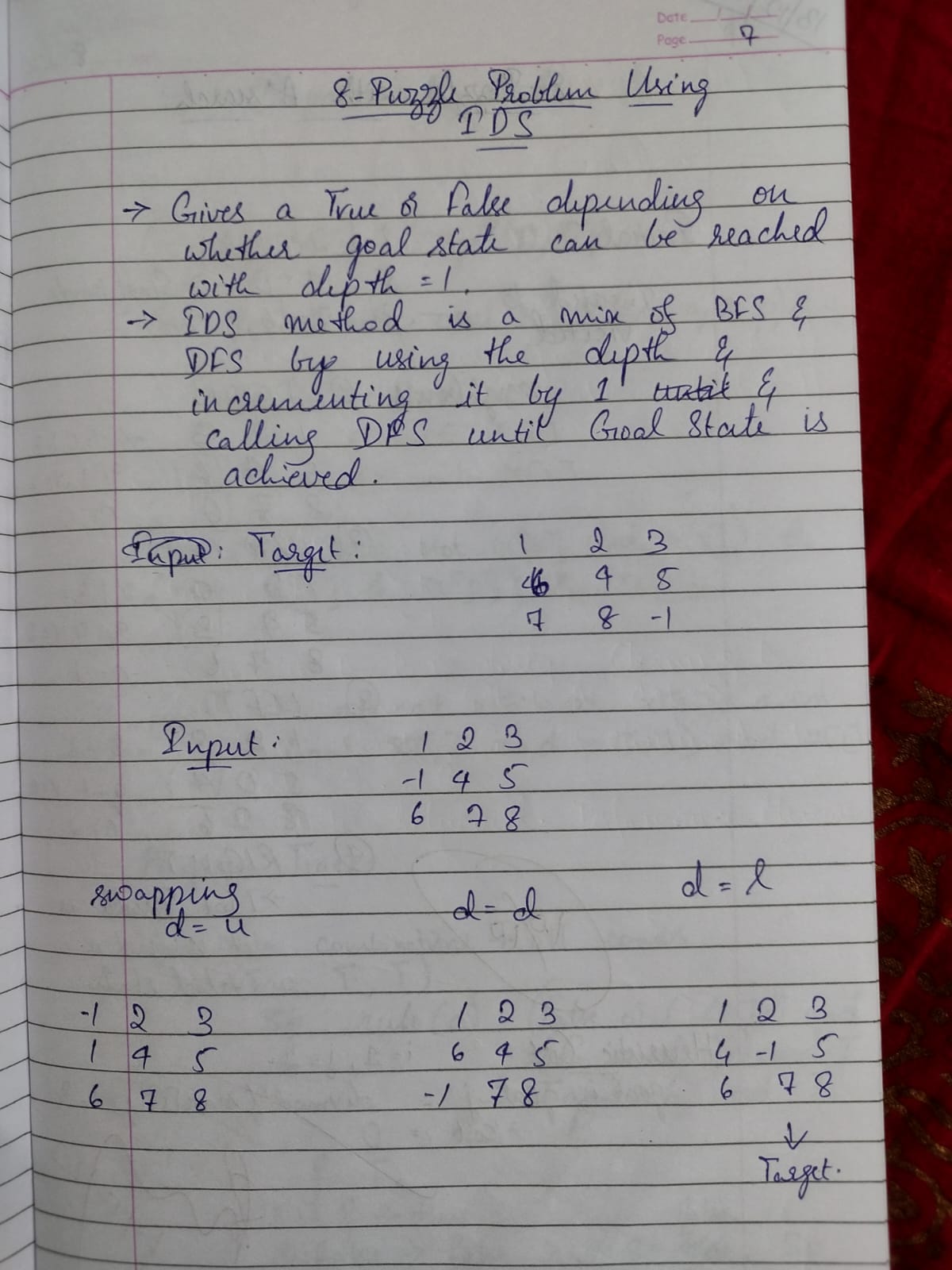
target = list(map(int, input().split()))

depth = 8

iddfs(src, target, depth)

OUTPUT





1. **Implement A\* search algorithm**

class Node:

    def \_\_init\_\_(self,data,level,fval):

        self.data = data

        self.level = level

        self.fval = fval

    def generate\_child(self):

        x,y = self.find(self.data,'\_')

        val\_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]

        children = []

        for i in val\_list:

            child = self.shuffle(self.data,x,y,i[0],i[1])

            if child is not None:

                child\_node = Node(child,self.level+1,0)

                children.append(child\_node)

        return children

    def shuffle(self,puz,x1,y1,x2,y2):

        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

            temp\_puz = []

            temp\_puz = self.copy(puz)

            temp = temp\_puz[x2][y2]

            temp\_puz[x2][y2] = temp\_puz[x1][y1]

            temp\_puz[x1][y1] = temp

            return temp\_puz

        else:

            return None

    def copy(self,root):

        temp = []

        for i in root:

            t = []

            for j in i:

                t.append(j)

            temp.append(t)

        return temp

    def find(self,puz,x):

        for i in range(0,len(self.data)):

            for j in range(0,len(self.data)):

                if puz[i][j] == x:

                    return i,j

class Puzzle:

    def \_\_init\_\_(self,size):

        self.n = size

        self.open = []

        self.closed = []

    def accept(self):

        puz = []

        for i in range(0,self.n):

            temp = input().split(" ")

            puz.append(temp)

        return puz

    def f(self,start,goal):

        return self.h(start.data,goal)+start.level

    def h(self,start,goal):

        temp = 0

        for i in range(0,self.n):

            for j in range(0,self.n):

                if start[i][j] != goal[i][j] and start[i][j] != '\_':

                    temp += 1

        return temp

    def process(self):

        print("Enter the start state matrix \n")

        start = self.accept()

        print("Enter the goal state matrix \n")

        goal = self.accept()

        start = Node(start,0,0)

        start.fval = self.f(start,goal)

        self.open.append(start)

        print("\n\n")

        while True:

            cur = self.open[0]

            print("")

            print("  | ")

            print("  | ")

            print(" \\\'/ \n")

            for i in cur.data:

                for j in i:

                    print(j,end=" ")

                print("")

            if(self.h(cur.data,goal) == 0):

                break

            for i in cur.generate\_child():

                i.fval = self.f(i,goal)

                self.open.append(i)

            self.closed.append(cur)

            del self.open[0]

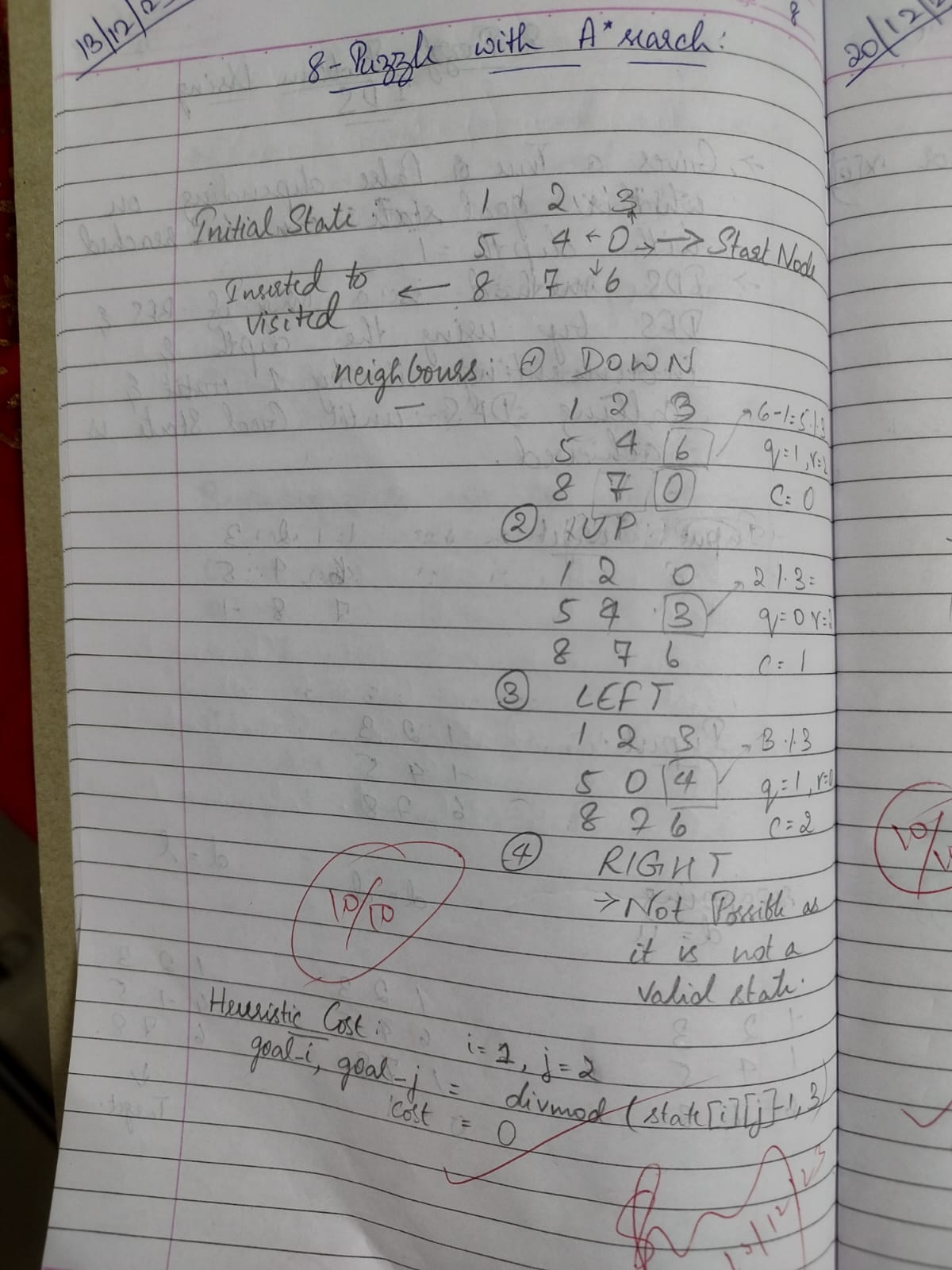
            self.open.sort(key = lambda x:x.fval,reverse=False)

puz = Puzzle(3)

puz.process()

OUTPUT





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

def tell(kb, rule):

    kb.append(rule)

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)]

def ask(kb, q):

    for c in combinations:

        s = r1(c)

        f = q(c)

        print(s, f)

        if s != f and s != False:

            return 'Does not entail'

    return 'Entails'

kb = []

rule\_str = input("Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): ")

r1 = eval(rule\_str)

tell(kb, r1)

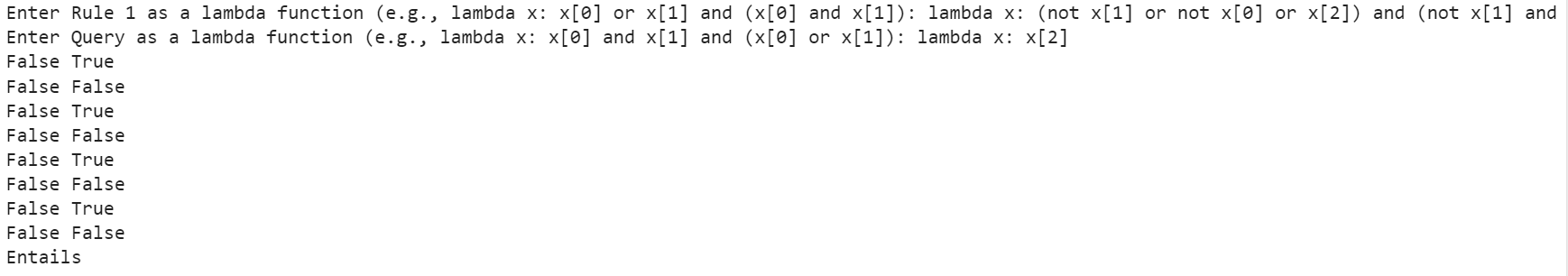
query\_str = input("Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): ")

q = eval(query\_str)

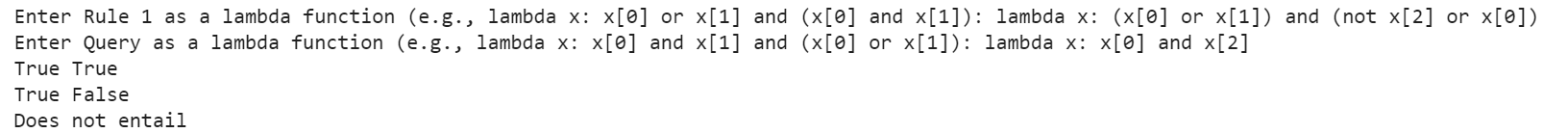
result = ask(kb, q)

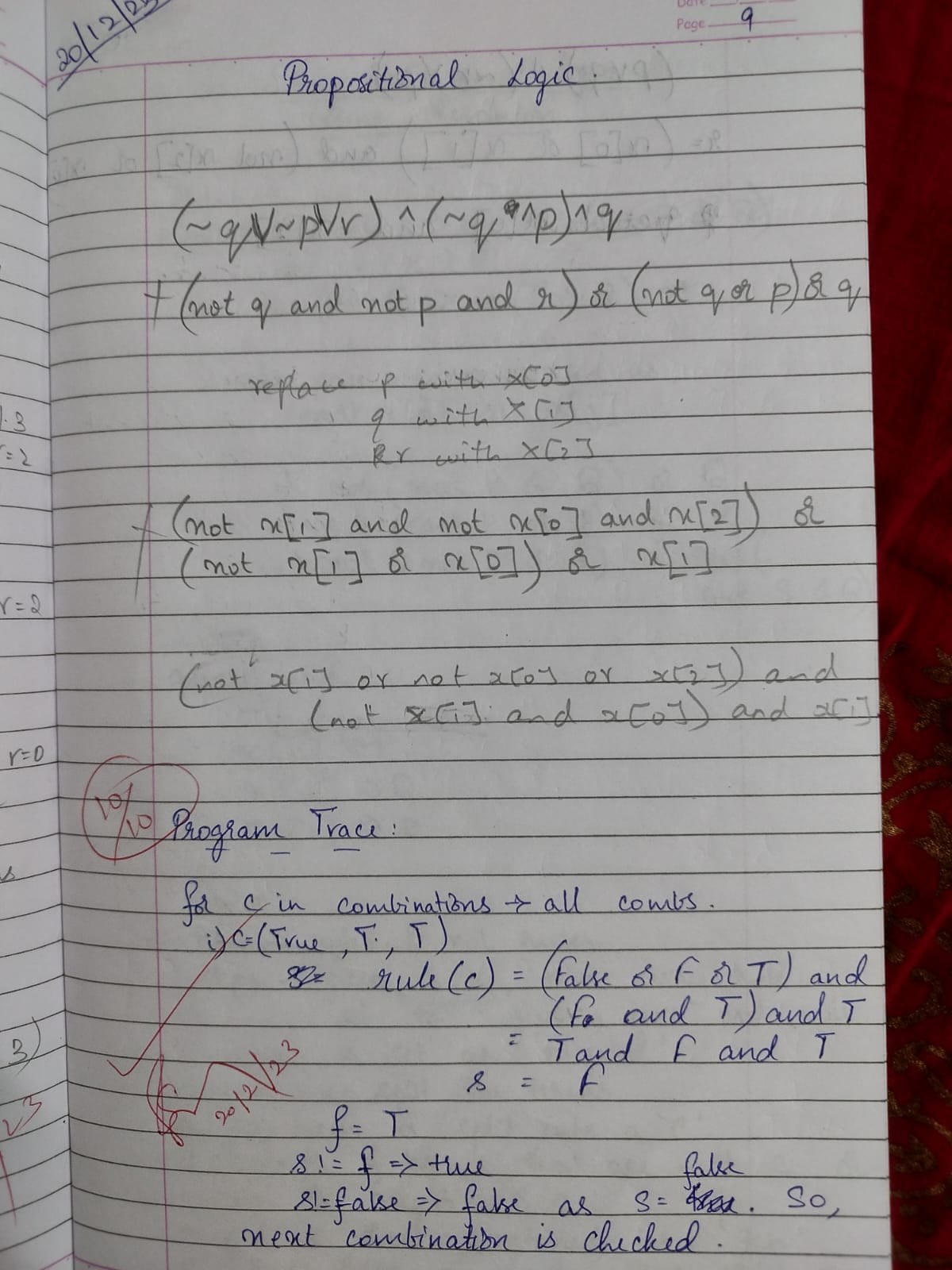
print(result)

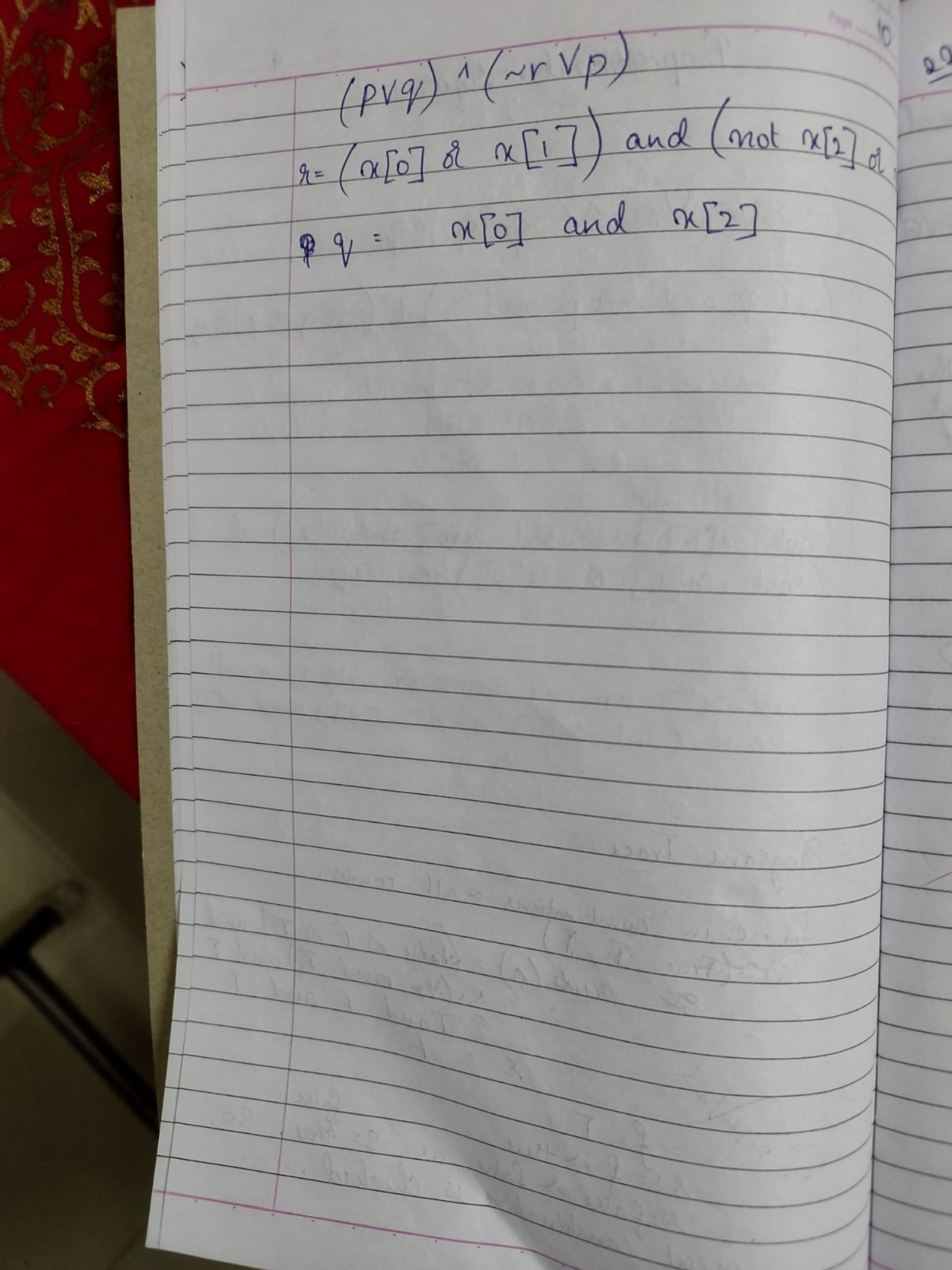
OUTPUT 1



OUTPUT 2







1. **Create a knowledge base using prepositional logic and prove the**

**given query using resolution**

import re

def main():

    rules = input("Enter the rules (space-separated): ")

    goal = input("Enter the goal: ")

    rules = rules.split(' ')

    steps = resolve(rules, goal)

    print('\nStep\t|Clause\t|Derivation\t')

    print('-' \* 30)

    i = 1

    for step in steps:

        print(f' {i}.\t| {step}\t| {steps[step]}\t')

        i += 1

def negate(term):

    return f'~{term}' if term[0] != '~' else term[1]

def split\_terms(rule):

    exp = '(~\*[PQRS])'

    terms = re.findall(exp, rule)

    return terms

def contradiction(goal, clause):

    contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']

    return 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) == 2:

                        if gen[0] != negate(gen[1]):

                            clauses += [f'{gen[0]}v{gen[1]}']

                            if contradiction(goal,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]}, which is in turn null. \

                                \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."

                                return steps

                    elif len(gen) == 1:

                        clauses += [f'{gen[0]}']

                    else:

                        if contradiction(goal,f'{terms1[0]}v{terms2[0]}'):

                            temp.append(f'{terms1[0]}v{terms2[0]}')

                            steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \

                            \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."

                            return steps

            for clause in clauses:

                if clause not in temp :

                    temp.append(clause)

                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'

            j = (j + 1) % n

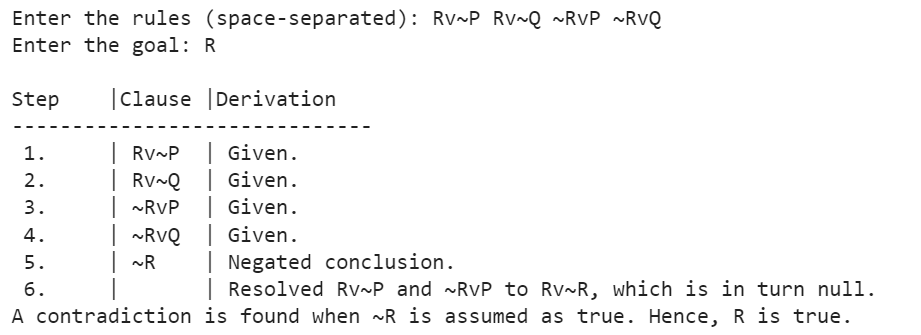
        i += 1

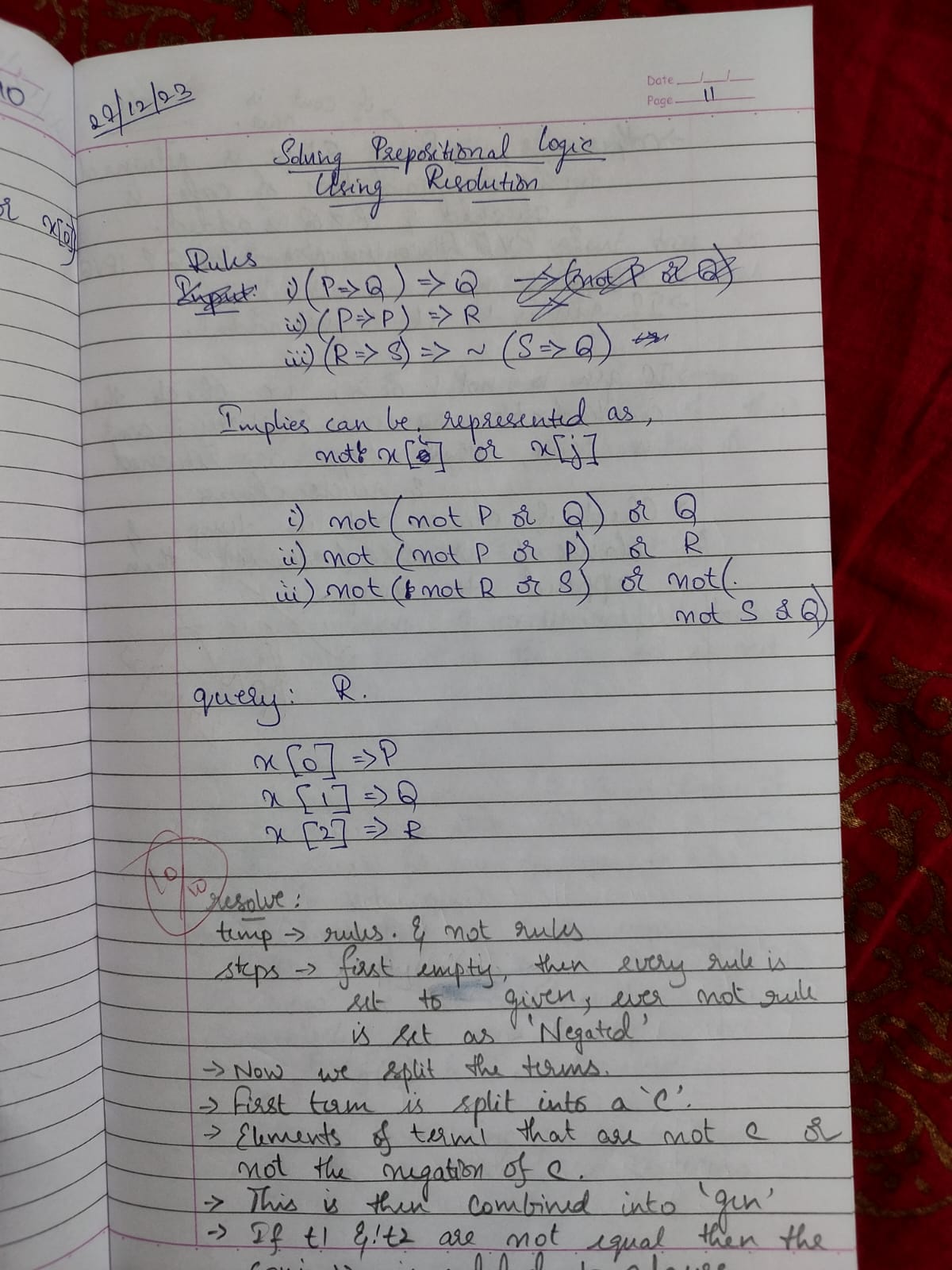
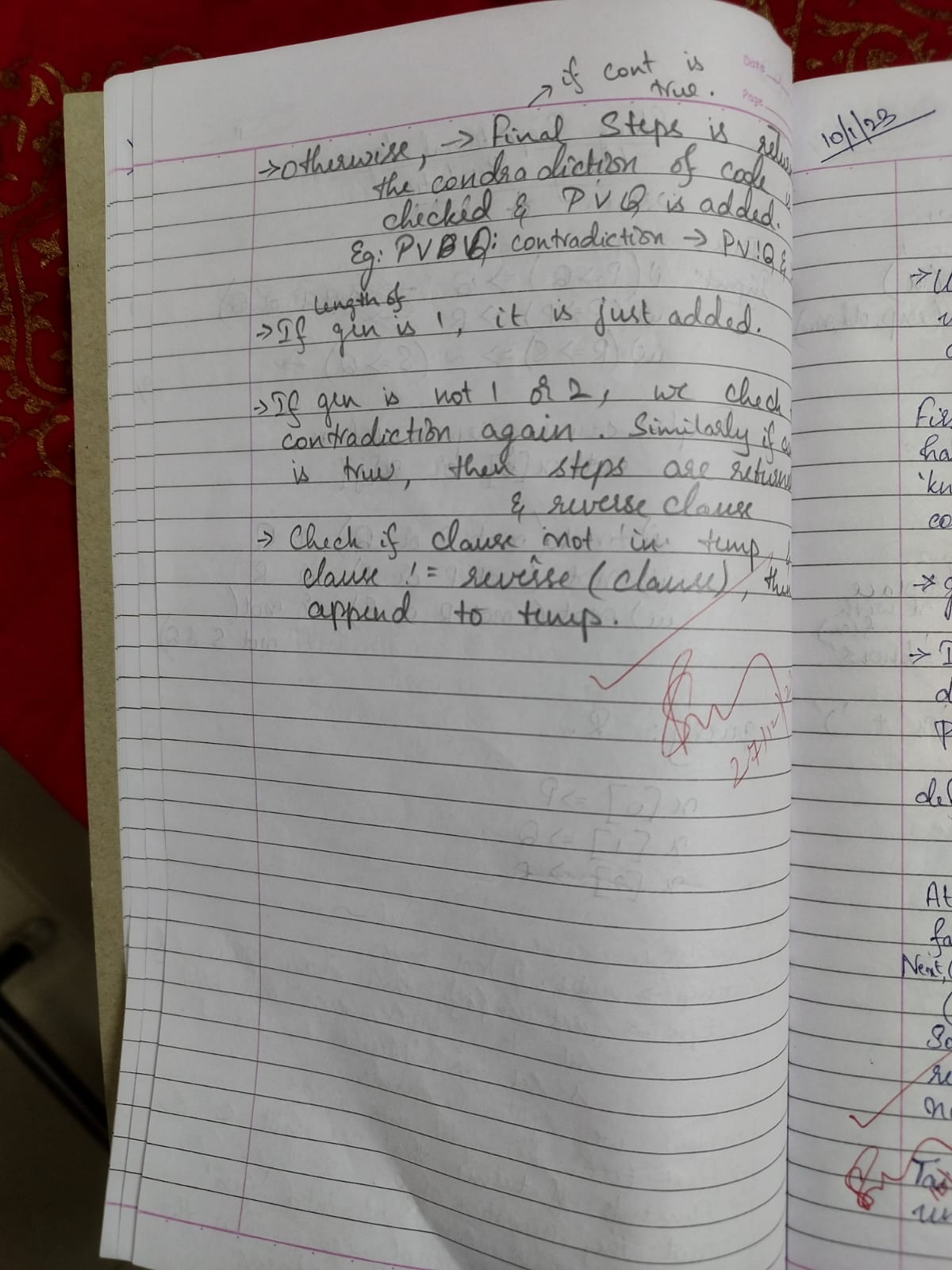
    return steps

if \_\_name\_\_ == "\_\_main\_\_":

    main()

OUTPUT



1. **Implement unification in first order logic**

import re

def getAttributes(expression):

    expression = expression.split("(")[1:]

    expression = "(".join(expression)

    expression = expression[:-1]

    expression = re.split("(?<!\(.),(?!.\))", expression)

    return expression

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)

    for index, val in enumerate(attributes):

        if val == old:

            attributes[index] = new

    predicate = getInitialPredicate(exp)

    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:

            return False

    if isConstant(exp1):

        return [(exp1, exp2)]

    if isConstant(exp2):

        return [(exp2, exp1)]

    if isVariable(exp1):

        if checkOccurs(exp1, exp2):

            return False

        else:

            return [(exp2, exp1)]

    if isVariable(exp2):

        if checkOccurs(exp2, exp1):

            return False

        else:

            return [(exp1, exp2)]

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):

        print("Predicates do not match. Cannot be unified")

        return False

    attributeCount1 = len(getAttributes(exp1))

    attributeCount2 = len(getAttributes(exp2))

    if attributeCount1 != attributeCount2:

        return False

    head1 = getFirstPart(exp1)

    head2 = getFirstPart(exp2)

    initialSubstitution = unify(head1, head2)

    if not initialSubstitution:

        return False

    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 False

    initialSubstitution.extend(remainingSubstitution)

    return initialSubstitution

exp1 = input("Enter the first expression: ")

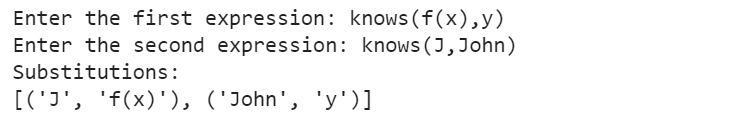
exp2 = input("Enter the second expression: ")

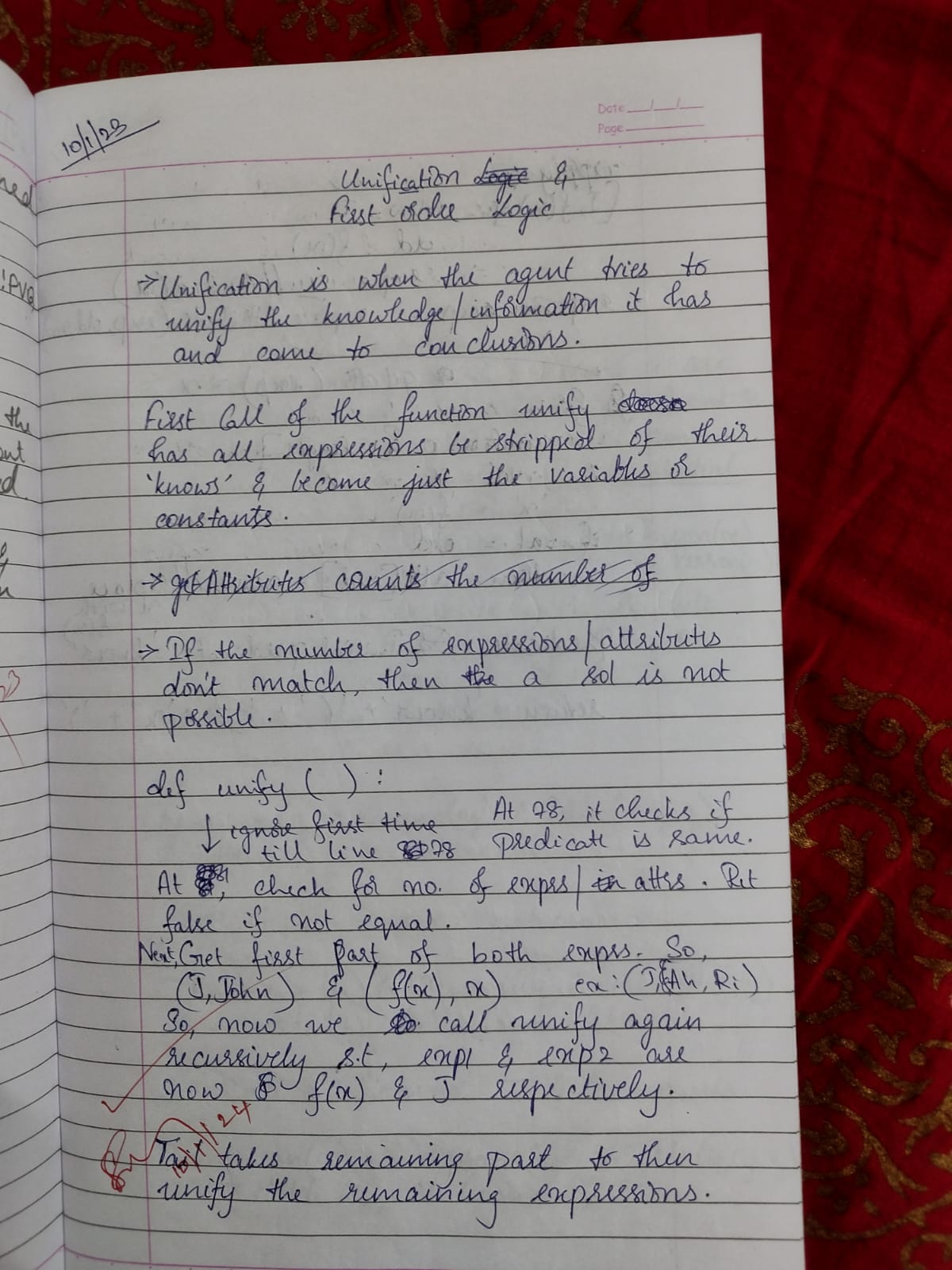
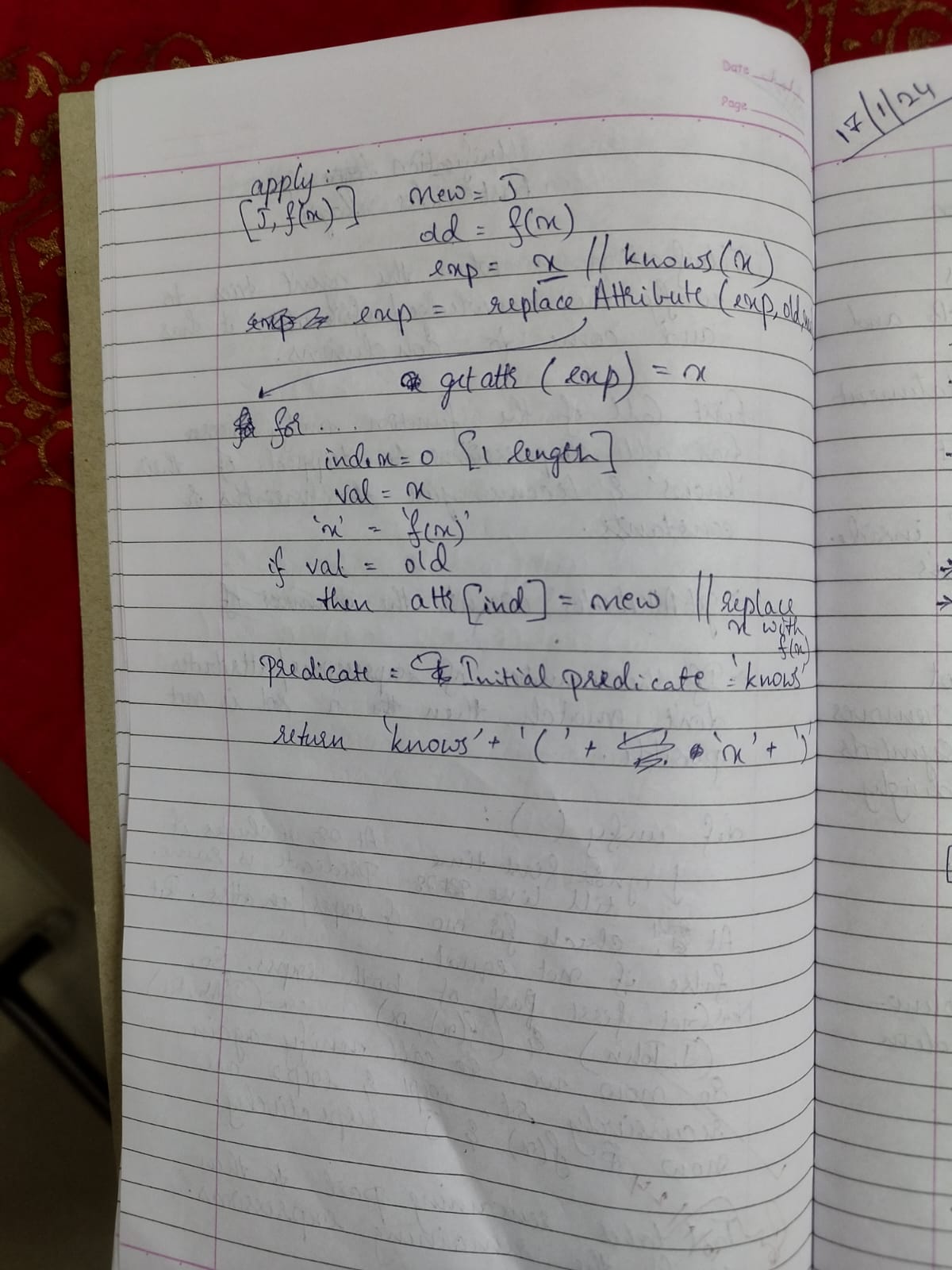
substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)

OUTPUT



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

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 == '|':

            s[i] = '&'

        elif c == '&':

            s[i] = '|'

    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('[∀∃].', 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] + '|' + statement[i+1:]

        statement = statement[:br] + new\_statement if br > 0 else new\_statement

    while '~∀' in statement:

        i = statement.index('~∀')

        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] = '∀', s[i+2], '~'

        statement = ''.join(s)

    statement = statement.replace('~[∀','[~∀')

    statement = statement.replace('~[∃','[~∃')

    expr = '(~[∀|∃].)'

    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))

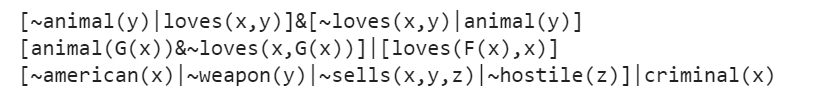
    return statement

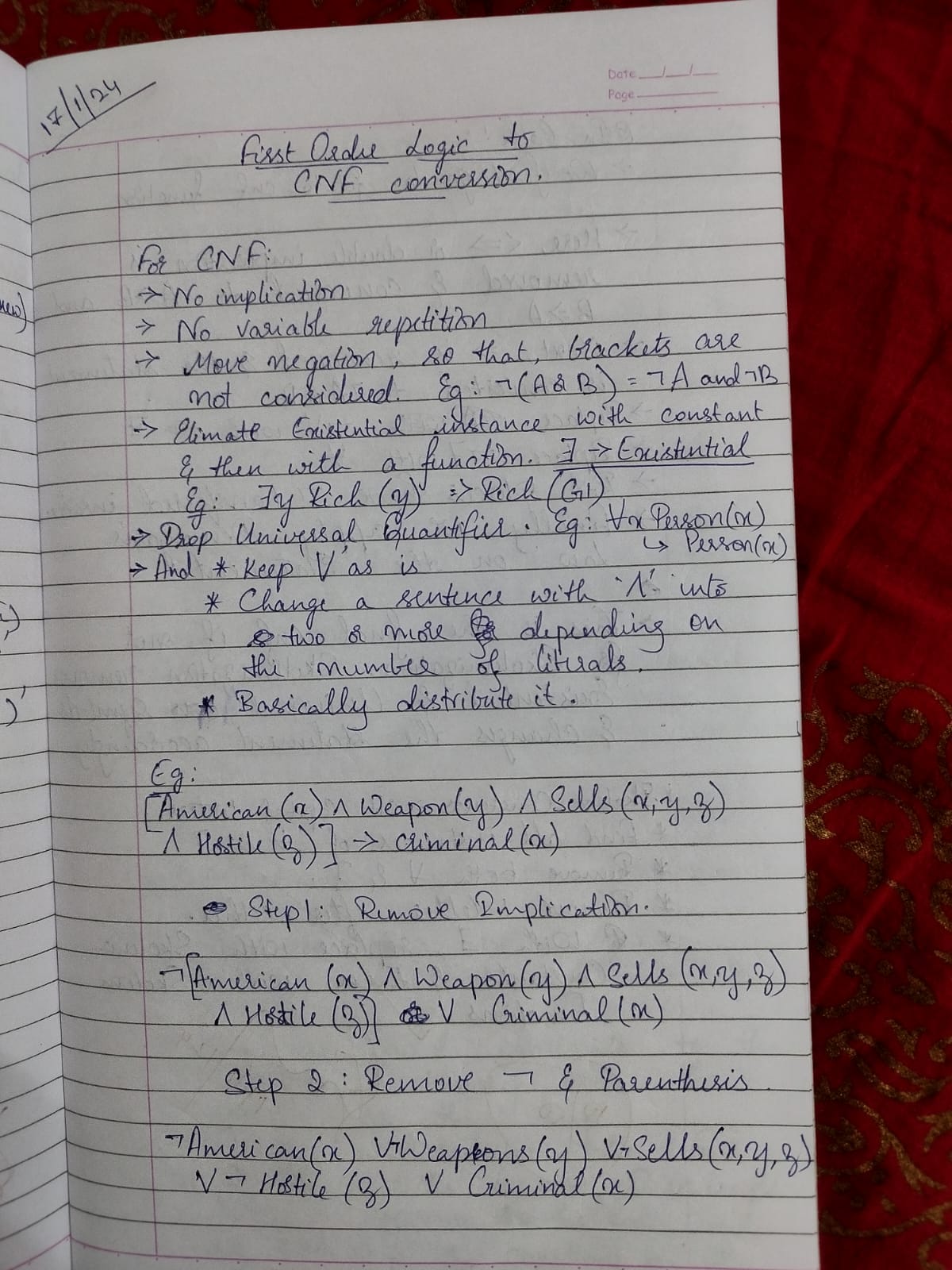
print(Skolemization(fol\_to\_cnf("animal(y)<=>loves(x,y)")))

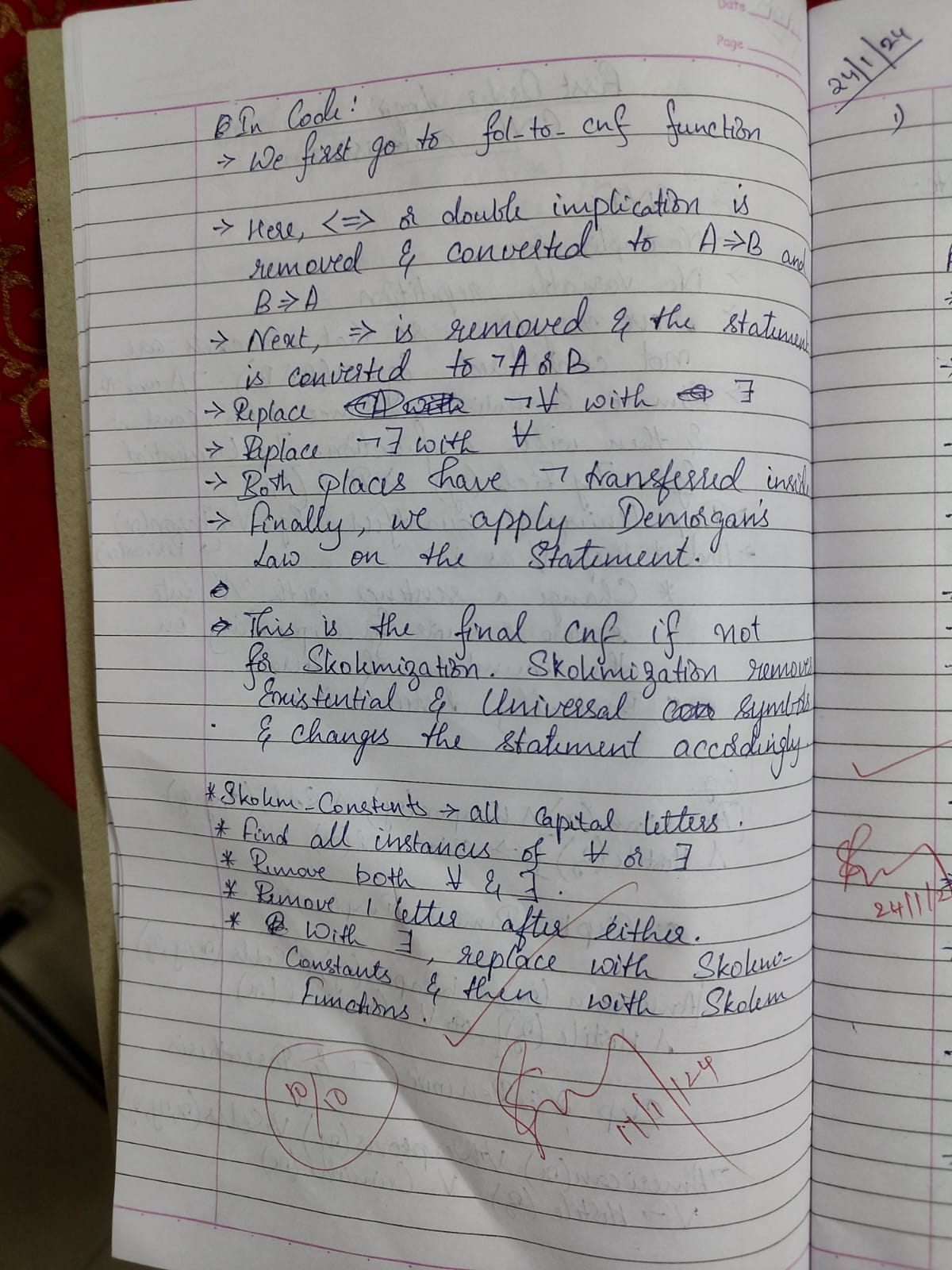
print(Skolemization(fol\_to\_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))

print(fol\_to\_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))

OUTPUT







1. **Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning**

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]

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 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.expression for f in self.facts])):

            print(f'\t{i+1}. {f}')

kb = KB()

kb.tell('missile(x)=>weapon(x)')

kb.tell('missile(M1)')

kb.tell('enemy(x,America)=>hostile(x)')

kb.tell('american(West)')

kb.tell('enemy(Nono,America)')

kb.tell('owns(Nono,M1)')

kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')

kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')

kb.query('criminal(x)')

kb.display()

OUTPUT

