B.M.S. COLLEGE OF ENGINEERING
BENGALURU-560019
Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

(22CS5PCAIN)

Submitted in partial fulfillment for the 5^{5h} Semester Laboratory

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

Tanisha Gotadke
1BM21CS229

Department of Computer Science and
Engineering B.M.S. College of
Engineering Bull Temple Road,
Basavanagudi, Bangalore- 560019
Nov 23-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 Tanisha Gotadke (1BM21CS229) during the 5th Semester November - February 2024.

Signature of the Faculty Incharge:
Sandhya A Kulkarni
Assistant Professor

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

Table of Contents

Program-1

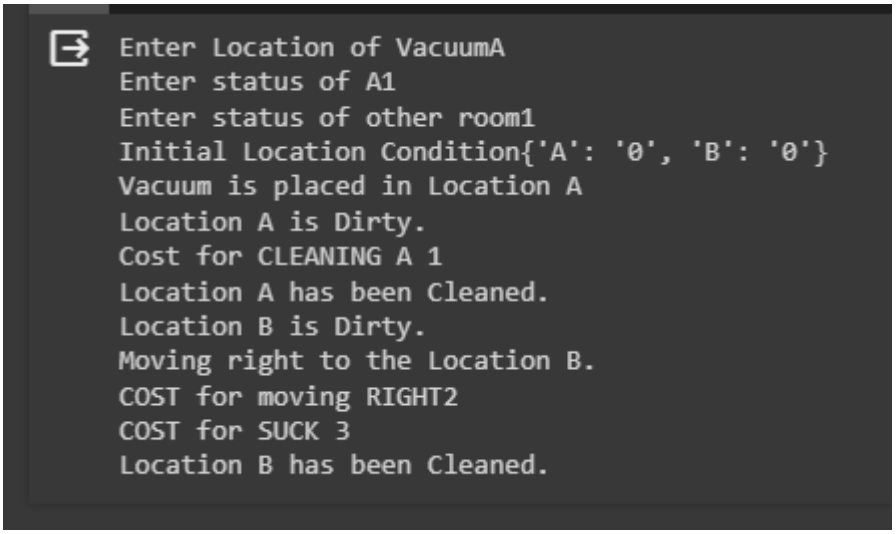
Implement Vacuum cleaner problem for 2 rooms ,any type of agent can be considered simple reflex or model based etc.

Code:

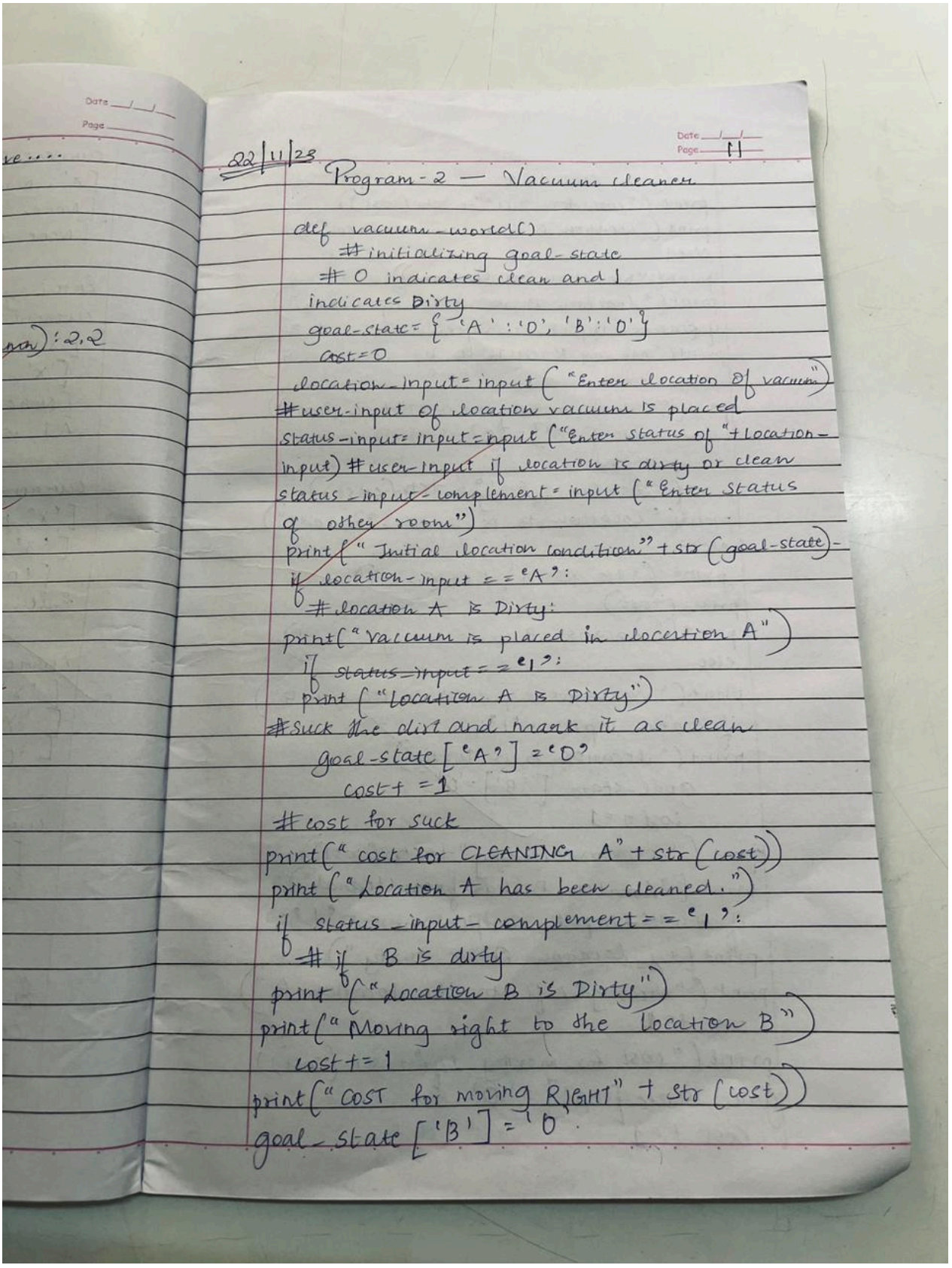
```
def clean(floor, row, col):
    i, j, m, n = row, col, len(floor), len(floor[0])
    goRight = goDown = True
    cleaned = [not any(f) for f in floor]
    while not all(cleaned):
        while any(floor[i]):
            print_floor(floor, i, j)
            if floor[i][j]:
                floor[i][j] = 0
                print_floor(floor, i, j)
            if not any(floor[i]):
                cleaned[i] = True
                break
            if j == n - 1:
                j -= 1
                goRight = False
            elif j == 0:
                j += 1
                goRight = True
            else:
                j += 1 if goRight else -1
        if all(cleaned):
            break
        if i == m - 1:
            i -= 1
            goDown = False
        elif i == 0:
            i += 1
```

```
            goDown = True
        else:
            i += 1 if goDown else -1
        if cleaned[i]:
            print_floor(floor, i, j)

def print_floor(floor, row, col): # row, col represent the current
    vacuum cleaner position for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r == row and c == col:
                print(f" >{floor[r][c]}< ", end = '')
            else:
                print(f"  {floor[r][c]}  ", end = '')
        print(end = '\n')
    print(end = '\n')

# Test 1
floor = [[1, 0, 0, 0],
         [0, 1, 0, 1],
         [1, 0, 1, 1]]

print("Room Condition: ")
for row in floor:
    print(row)
print("\n")
clean(floor, 1, 2)
```

Output:


Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.

Notes:



22/11/23

Program - 2 — Vacuum cleaner

```
def vacuum_world()
    #initializing goal_state
    # 0 indicates clean and 1
    indicates Dirty
    goal_state = {'A':'0', 'B':'0'}
    cost=0
    location_input = input("Enter location of vacuum")
#user_input of location vacuum is placed
status_input= input=input ("Enter status of "+location_
input) #user_input if location is dirty or clean
status_input_complement = input ("Enter status
of other room")
print(" Initial location condition" + str(goal_state)-
if location_input == 'A':
    # location A is Dirty:
print("vacuum is placed in location A")
    if status_input == '1':
    print ("Location A is Dirty")
#Suck the dirt and mark it as clean
    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':
    # if B is dirty
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'.
```

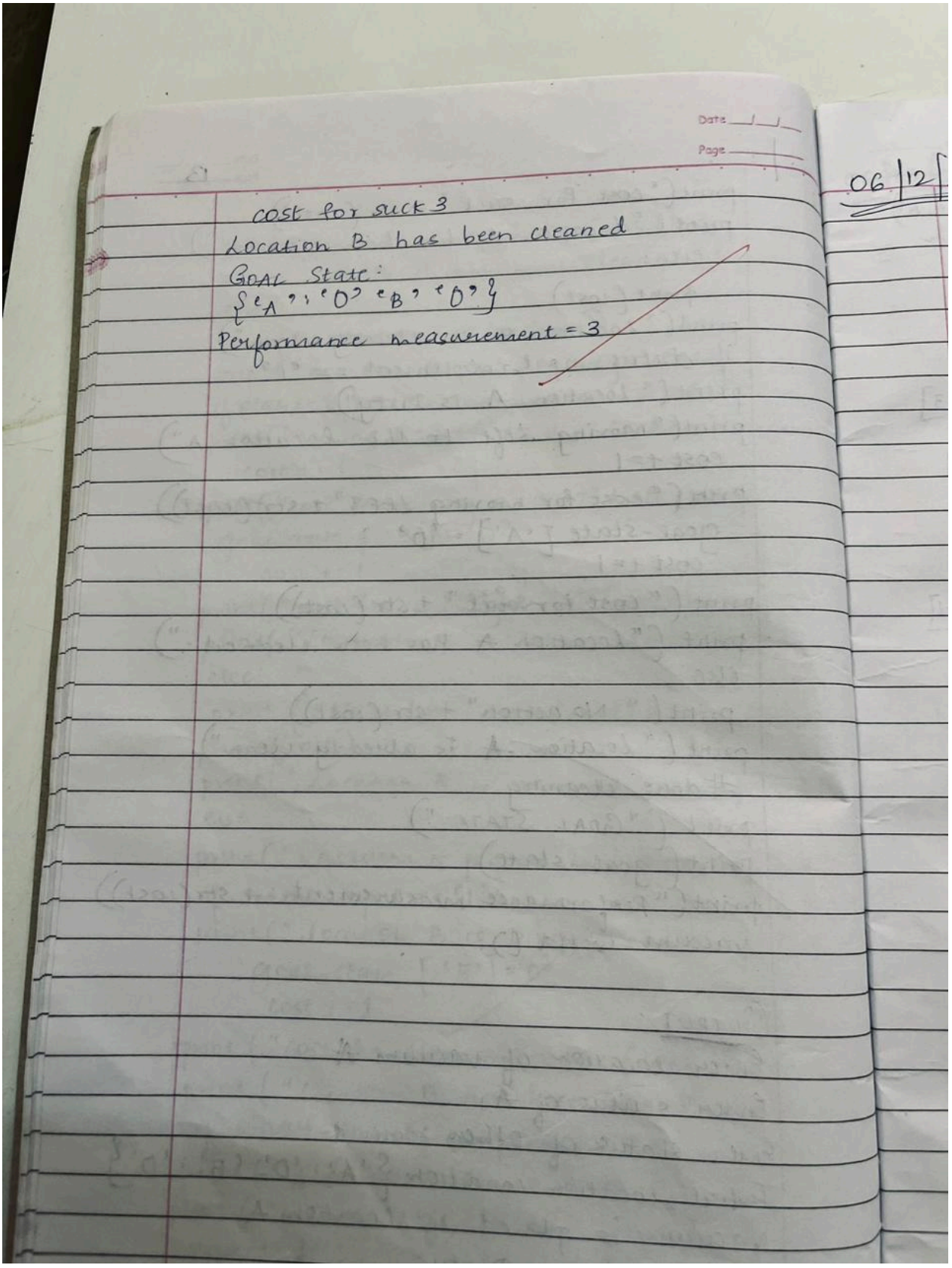
```
        cost += 1
print("cost for suck" + str(cost))
print("Location B has been cleaned")
else:
print("No action" + str(cost))
print("Location B is already clean")
if status_input_complement == '1':
print("Moving RIGHT to the Location B")
    cost += 1
print("cost for moving RIGHT" + str(cost))
 goal_state['B'] = '0'
    cost += 1
print("cost for suck" + str(cost))
print("Location B is been cleaned:")
else:
 print("No action" + str(cost))
print(cost)
print("Location B is already clean")
else
print("vaccum 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
```

Date / /
Page 13

```
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")
#done cleaning
print("GOAL STATE:")
print(goal-state)
print("Performance measurement:" + str(cost))
vaccum_world().
```

OUTPUT:

```
Enter location of vacuum A
Enter status of A1
Enter status of other room 1
Initial location condition {'A': '0', 'B': '0'}
vacuum is placed in location A
location A is Dirty
cost for cleaning A1
location A has been cleaned
location B is Dirty
Moving right to the location B
COST for moving RIGHT 2
```


Date
Page
06 | 12
cost for suck 3
Location B has been cleaned
GOAL State:
{'A', 'O', 'B', 'O'}
Performance measurement = 3

Program- 2

Explore the working of Tic Tac Toe using Min max strategy

Code:

```
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
            elif x == "X":
                countX = countX + 1
    if countO >= countX:
        return X
    elif 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
        elif 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 (board[0][0] == board[0][1] == board[0][2] == O or
board[1][0] == board[1][1] == board[1][2] == O or board[2][0] ==
board[2][1] == board[2][2] == O):
        return O
    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][i])
    if (strikeD[0] == strikeD[1] == strikeD[2]):
        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) is not None):
        return True
    return False


def utility(board):
    if (winner(board) == X):
        return 1
    elif winner(board) == O:


        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(minimax_helper(board))
        board[move[0]][move[1]] = EMPTY
    return max(scores) if isMaxTurn else min(scores)


def minimax(board):
    isMaxTurn = True if player(board) == X else False
    bestMove = None
    if isMaxTurn:
        bestScore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestScore):
                bestScore = score
                bestMove = move


        return bestMove
    else:
        bestScore = +math.inf
```

```
            for move in actions(board):
                result(board, move)
                score = minimax_helper(board)
                board[move[0]][move[1]] = EMPTY
                if (score < bestScore):
                    bestScore = score
                    bestMove = move
            return bestMove


        def print_board(board):
            for row in board:
                print(row)


        # Example usage:
        game_board = initial_state()
        print("Initial Board:")
        print_board(game_board)

        while not terminal(game_board):
            if player(game_board) == X:
                user_input = input("\nEnter your move (row, column): ")
                row, col = map(int, user_input.split(','))
                result(game_board, (row, col))
            else:
                print("\nAI is making a move...")


move = minimax(copy.deepcopy(game_board))
result(game_board, move)

print("\nCurrent Board:")
```

print_board(game_board)

Determine the winner

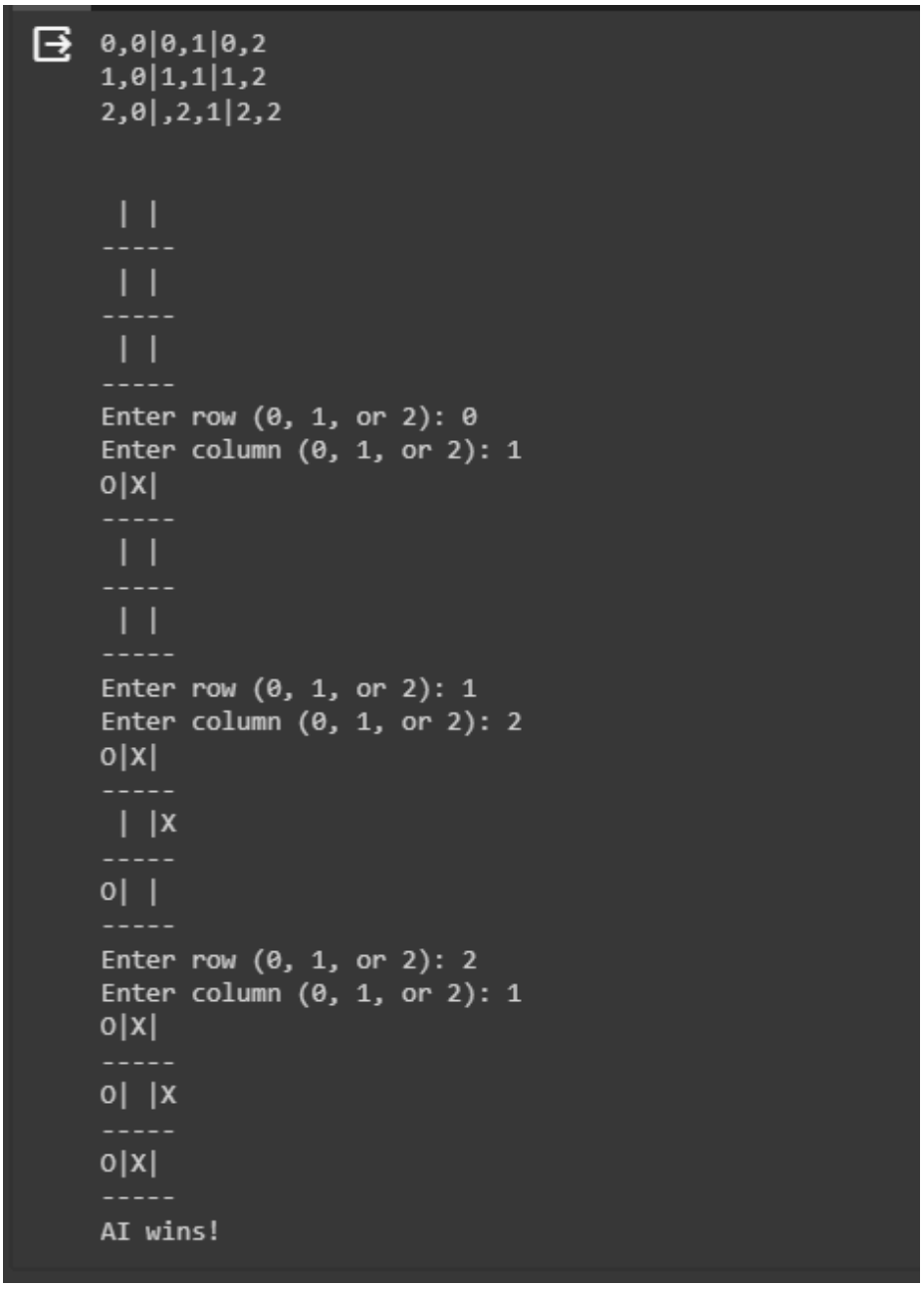
if winner(game_board) is not None:

print(f"\nThe winner is: {winner(game_board)}")

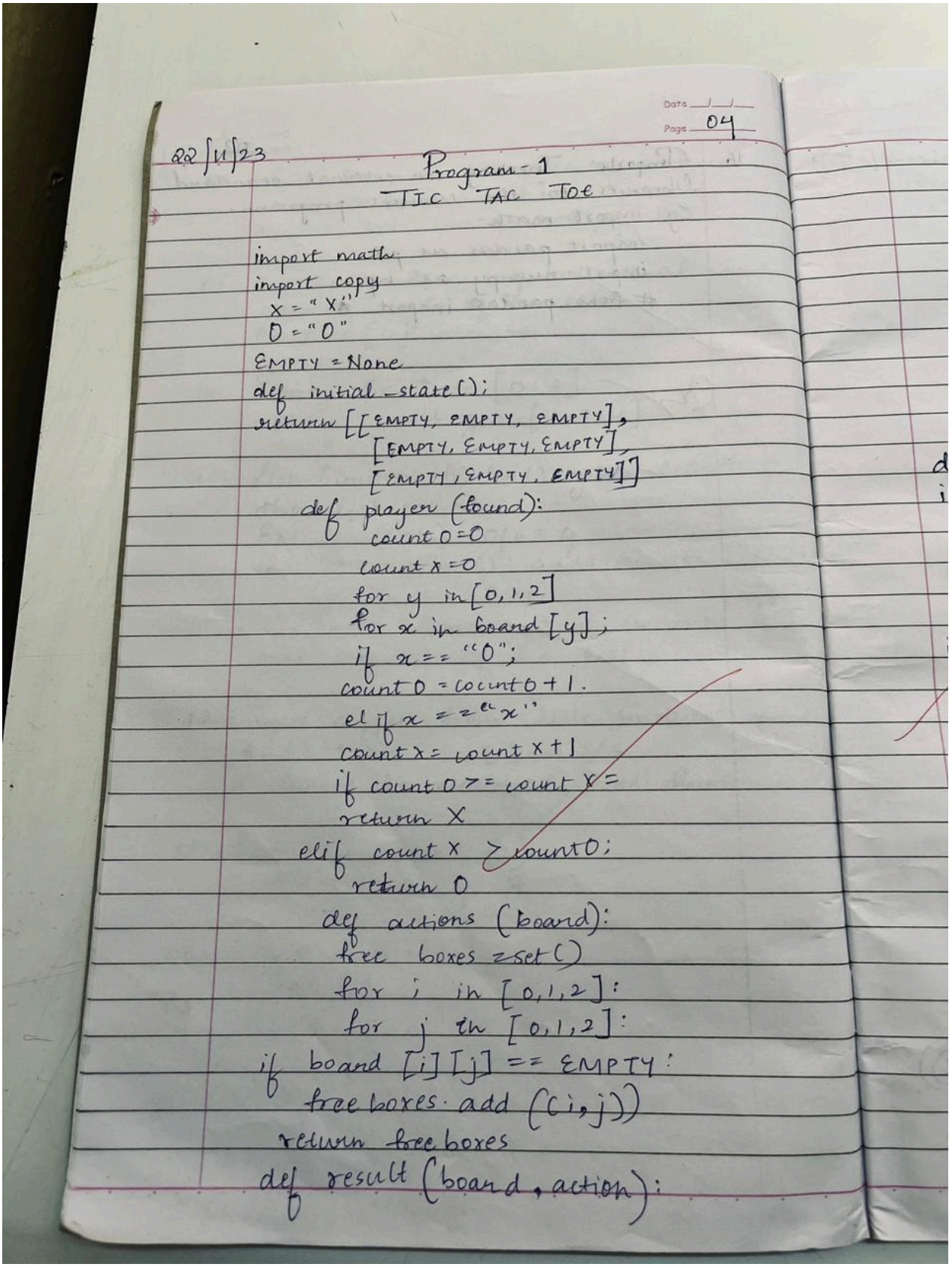
else:

print("\nIt's a tie!")

Output:



Notes:



Date / /
Page 04

22/11/23

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):
        count O=0
        count X=0
        for y in [0,1,2]
        for x in board [y];
        if x == "O";
        count O = count O + 1.
        el if x == "X"
        count X = count X + 1
        if count O >= count X =
        return X
    elif count X > count O;
        return O
    def actions (board):
        free boxes = set()
        for i in [0,1,2]:
        for j in [0,1,2]:
if board [i] [j] == EMPTY:
    free boxes. add ((i,j))
return free boxes
def result (board, action):
```

Date / / Page 04

Date / / Page 5

```
        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
    elif 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][2] == X
    return X
if (board[0][0] == board[0][1] ==
board[0][2] == O or board[1][0] ==
board[1][1] == board[1][2] == O or
board[2][0] == board[2][1] ==
    board[2][2] == O:
    return O
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][i])
if (striked[0] == striked[1] == striked[2]):
    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) is not None):
        return True
    return False
def utility (board):
    if (winner(board) == X):
        return 1
    elif winner(board) == O:
        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 (minimax-helper(board))
    board [move[0]] [move[1]] = EMPTY
return max(scores) if isMaxTurn
    else min(scores)
```

```
def minimax(board):
is Max Turn = True if player(board).
== X else FALSE.
best Move = None
if is Max Turn:
best Score = -math.inf
for move in actions(board)
result(board, move)
score =
minimax - helper(board)
board[move[0]][move[1]] = EMPTY
if (score > best Score)
best Score = Score
best Move = move
return best Move
else:
best Score = +math.inf
for move in actions(board):
result(board, move)
Score =
minimax-helper(board)
board[move[0]][move[1]] = EMPTY.
if (score < best Score):
best Score = Score
best Move = move
return best Move

def print-board(board)
for row in board:
print(row)

# Example usage:
game-board = initial-state()
```

```
print ("Initial Board:")
print_board (game_board)
while not terminal (game_board):
if player (game_board) == X:
user_input = input ("\n Enter your
move (row, column):")
row, col = map (int, user_input. split (','))
result (game_board, (row, col))
else:
print ("\n AI is making a move....")
move =
minimax (copy. deepcopy (game_board))
result (game_board, move)
print ("\n Current Board:")
print_board (game_board)

# Determine the winner
if winner (game_board) is not None:
print (f"\n The winner is:
{winner (game_board)}")
else:
print ("\n It's a tie!")
```

Output:

```
Initial Board:
[None, None, None]
[None, None, None]
[None, None, None]
Enter your move (row, column): 0,0

Current Board
['X', None, None]
[None, None, None]
```

Date / /
Page 9

Current Board:
['X', None, None]
[None, 'O', None]
[None, None, None]

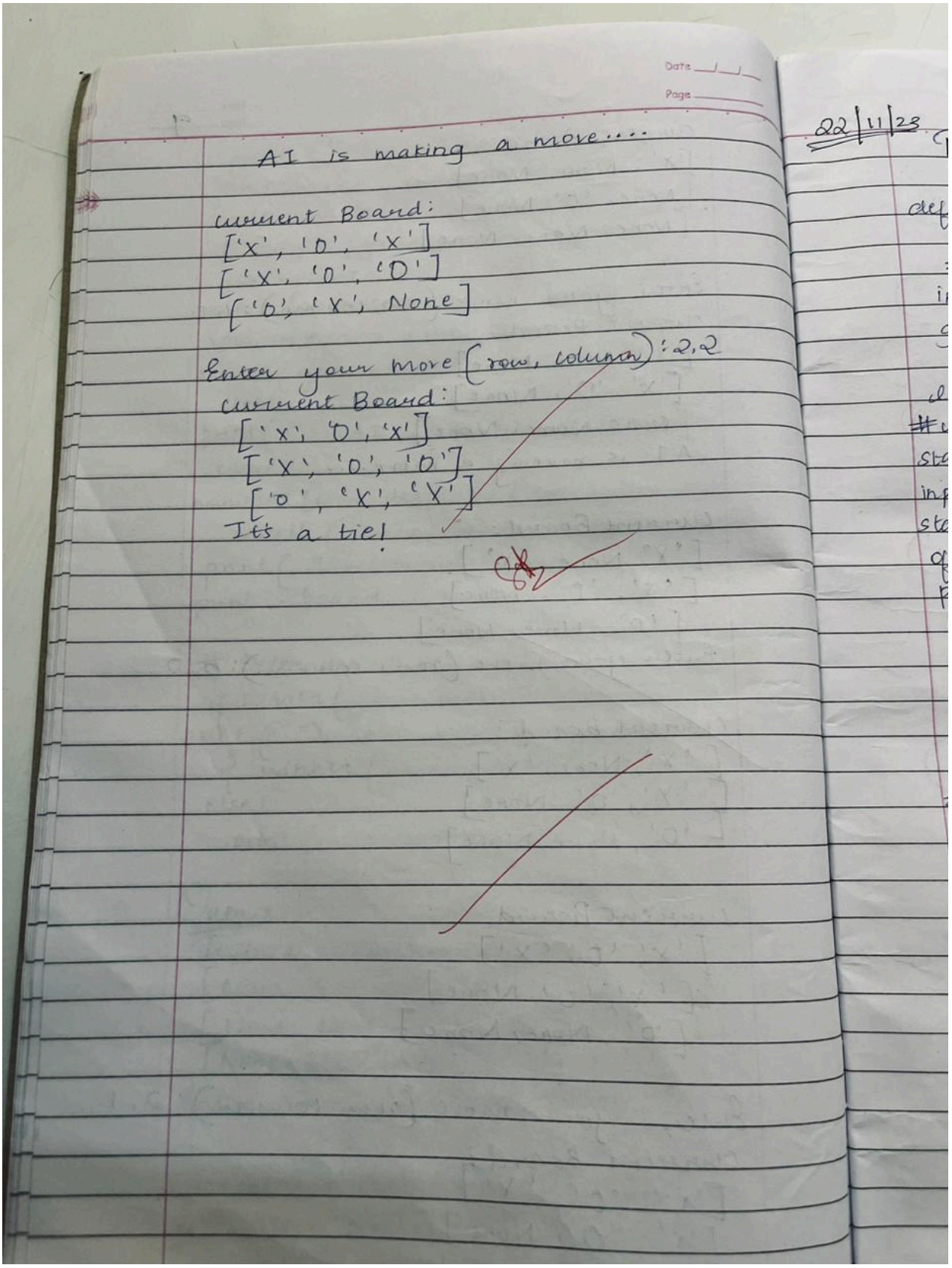
Enter your move (row, column): 1, 0.
current Board:
['X', None, None]
['X', 'O', None]
[None, None, None]
AI is making a move....

Current Board:
['X', None, 'X']
['X', 'O', None]
['O', None, None]
Enter your move (row, column): 0, 2.

Current Board:
['X', None, 'X']
['X', 'O', None]
['O', None, None]

current Board
['X', 'O', 'X']
['X', 'O', None]
['O', None, None]

Enter your move (row, column): 2, 1.
current Board:
['X', 'O', 'X']
['X', 'O', None]
['O', 'X', None]



AI is making a move....

current Board:
['X', 'O', 'X']
['X', 'O', 'O']
['O', 'X', None]

Enter your move (row, column): 2,2
current Board:
['X', 'O', 'X']
['X', 'O', 'O']
['O', 'X', 'X']
It's a tie!

22/11/23

Program- 3

Implement the 8 Puzzle Breadth First Search Algorithm.

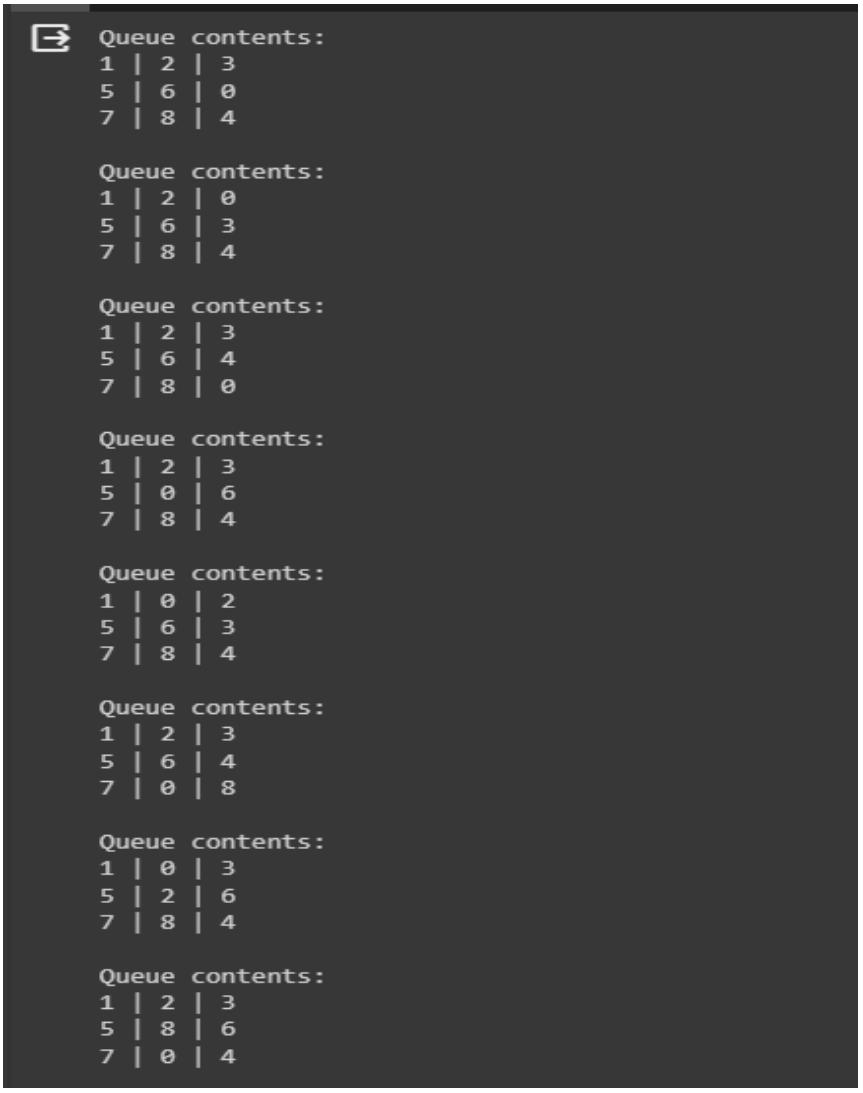
Code:

```
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):
  #index of empty spot
  b = state.index(0)

  #directions array
```

```
    d = []
    #Add all the possible directions

    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')



    # If direction is possible then add state to move
    pos_moves_it_can = []

    # for all possible directions find the state if that move is played
    ### Jump to gen function to generate all possible moves in the given directions

    for i in d:
        pos_moves_it_can.append(gen(state,i,b))

    return [move_it_can for move_it_can in pos_moves_it_can
if move_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[b+3]

    if m=='u':
```

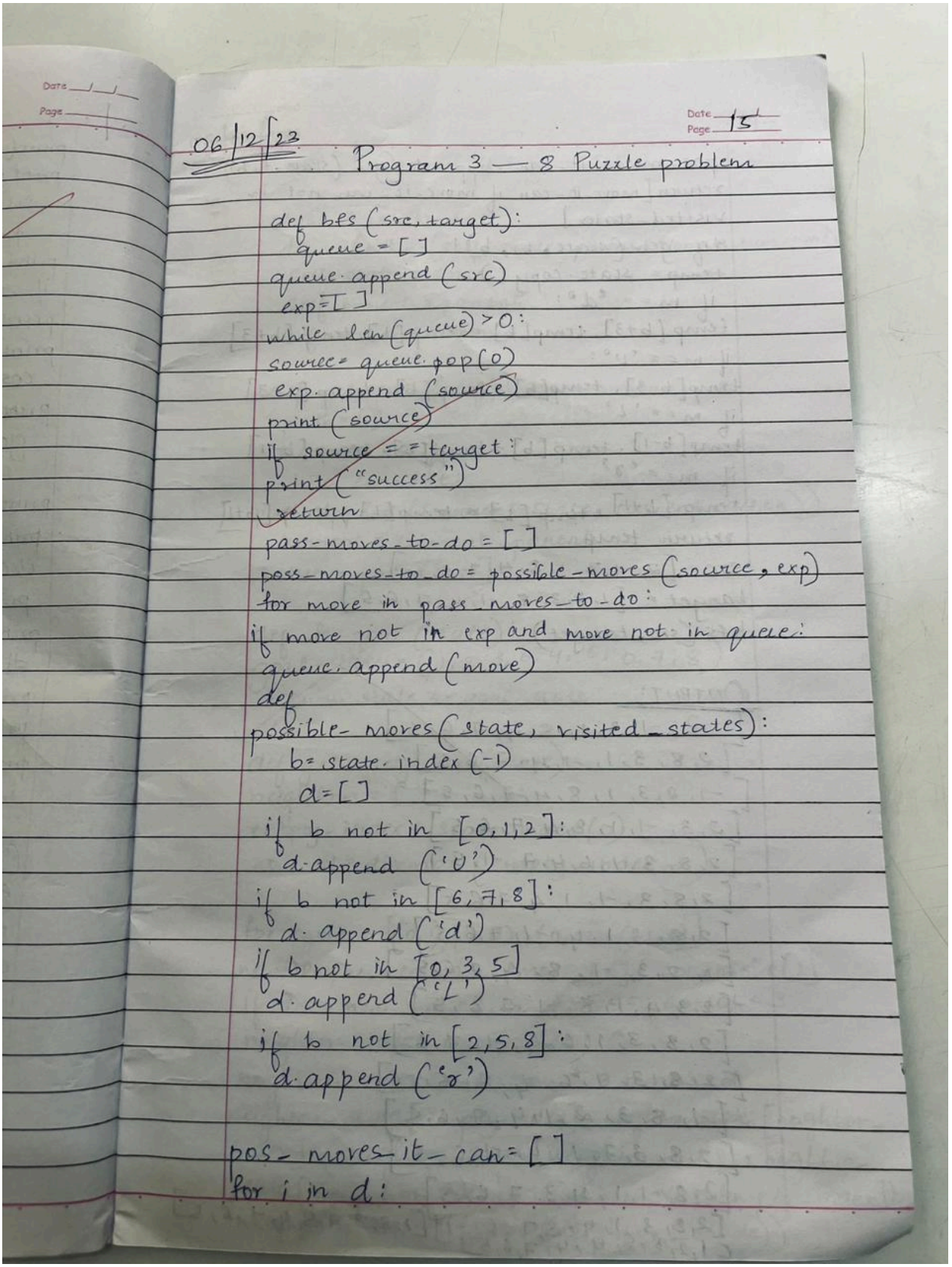
```
        temp[b-3],temp[b] = temp[b],temp[b-3]

    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]


    # return new state with tested move to later check if "src == target"
    return temp

print("Example 1")
src= [2,0,3,1,8,4,7,6,5]
target=[1,2,3,8,0,4,7,6,5]
print("Source: " , src)
print("Goal State: " , target)
bfs(src, target)

print("\nExample 2")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
bfs(src, target)
```

Output:


Queue contents:
1 | 2 | 3
5 | 6 | 0
7 | 8 | 4
Queue contents:
1 | 2 | 0
5 | 6 | 3
7 | 8 | 4
Queue contents:
1 | 2 | 3
5 | 6 | 4
7 | 8 | 0
Queue contents:
1 | 2 | 3
5 | 0 | 6
7 | 8 | 4
Queue contents:
1 | 0 | 2
5 | 6 | 3
7 | 8 | 4
Queue contents:
1 | 2 | 3
5 | 6 | 4
7 | 0 | 8
Queue contents:
1 | 0 | 3
5 | 2 | 6
7 | 8 | 4
Queue contents:
1 | 2 | 3
5 | 8 | 6
7 | 0 | 4

```
Queue contents:
1 | 6 | 2
5 | 0 | 3
7 | 8 | 4

Queue contents:
0 | 1 | 2
5 | 6 | 3
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 0 | 4
7 | 6 | 8

Queue contents:
1 | 2 | 3
5 | 6 | 4
0 | 7 | 8

Queue contents:
0 | 1 | 3
5 | 2 | 6
7 | 8 | 4

Queue contents:
1 | 3 | 0
5 | 2 | 6
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 8 | 6
0 | 7 | 4

success
Cost: 16
```

Notes:



06/12/22

Program 3 — 8 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
    pass_moves_to_do = []
    poss_moves_to_do = possible_moves(source, exp)
    for move in pass_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(-1)
        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 [0,3,5]
        d.append('L')
    if b not in [2,5,8]:
        d.append('r')

pos_moves_it_can = []
for i in d:
```

06/12/23

```
pos_moves_it_can.append(gen(state,i,b))
return[move_it_can if move_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[b+3]
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]
return temp
src = [2,-1,3,1,8,4,7,6,5]
target = [1,2,3,8,-1,4,7,6,5]
bfs(src, target)
```

OUTPUT:

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

Program- 4

Implement Iterative deepening search algo.

Code:

```
def iterative_deepening_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: # Set a reasonable depth limit to
            avoid an infinite loop print("Solution not found
            within depth limit.")
            return

def depth_limited_search(src, target, depth_limit, visited_states):
    if src == target:
        print_state(src)
        return src

    if depth_limit == 0:
        return None

    visited_states.append(src)
    poss_moves_to_do = possible_moves(src, visited_states)

    for move in poss_moves_to_do:
        if move not in visited_states:
            print_state(move)
```

```
            result = depth_limited_search(move, target,
            depth_limit - 1, visited_states) if result is not None:


                return result

    return None

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 i in d:
        pos_moves_it_can.append(gen(state, i, b))

    return [move_it_can for move_it_can in pos_moves_it_can
if move_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[b + 3]
    elif m == 'u':

        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    elif m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    elif m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    return temp

def print_state(state):
    print(f"{state[0]} {state[1]} {state[2]}\n{state[3]} {state[4]} {state[5]}\n{state[6]} {state[7]} {state[8]}\n")

print("Example 1")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
iterative_deepening_search(src, target)
```

Output:

```
Enter the number of verices:7
Enter the first vertex:0
Enter the second vertex:1
Enter the first vertex:0
Enter the second vertex:2
Enter the first vertex:1
Enter the second vertex:3
Enter the first vertex:1
Enter the second vertex:4
Enter the first vertex:2
Enter the second vertex:5
Enter the first vertex:2
Enter the second vertex:6
Enter the target vertex:6
Enter the max depth:3
Target is NOT reachable from source within 1
Target is NOT reachable from source within 2
Target is reachable from source within 3
COST:6
```

Program- 5

Implement A* search algo.

```
def print_grid(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):
    #Manhattan distance
    dist = 0
    for i in state:
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3
        x2, y2 = d2 % 3, d2 // 3

        dist += abs(x1-x2) + abs(y1-y2)
    return dist

def astar(src, target):
    states = [src]
    g = 0
    visited_states = set()
    while len(states):
        moves = []
```

```
        for state in states:
            visited_states.add(tuple(state))
            print_grid(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_states):
    b = state.index(-1)
    d = []
    if 9 > b - 3 >= 0:
        d += 'u'
    if 9 > b + 3 >= 0:
        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 tuple(move) not in visited_states]


def gen(state, direction, b):
    temp = state.copy()
```

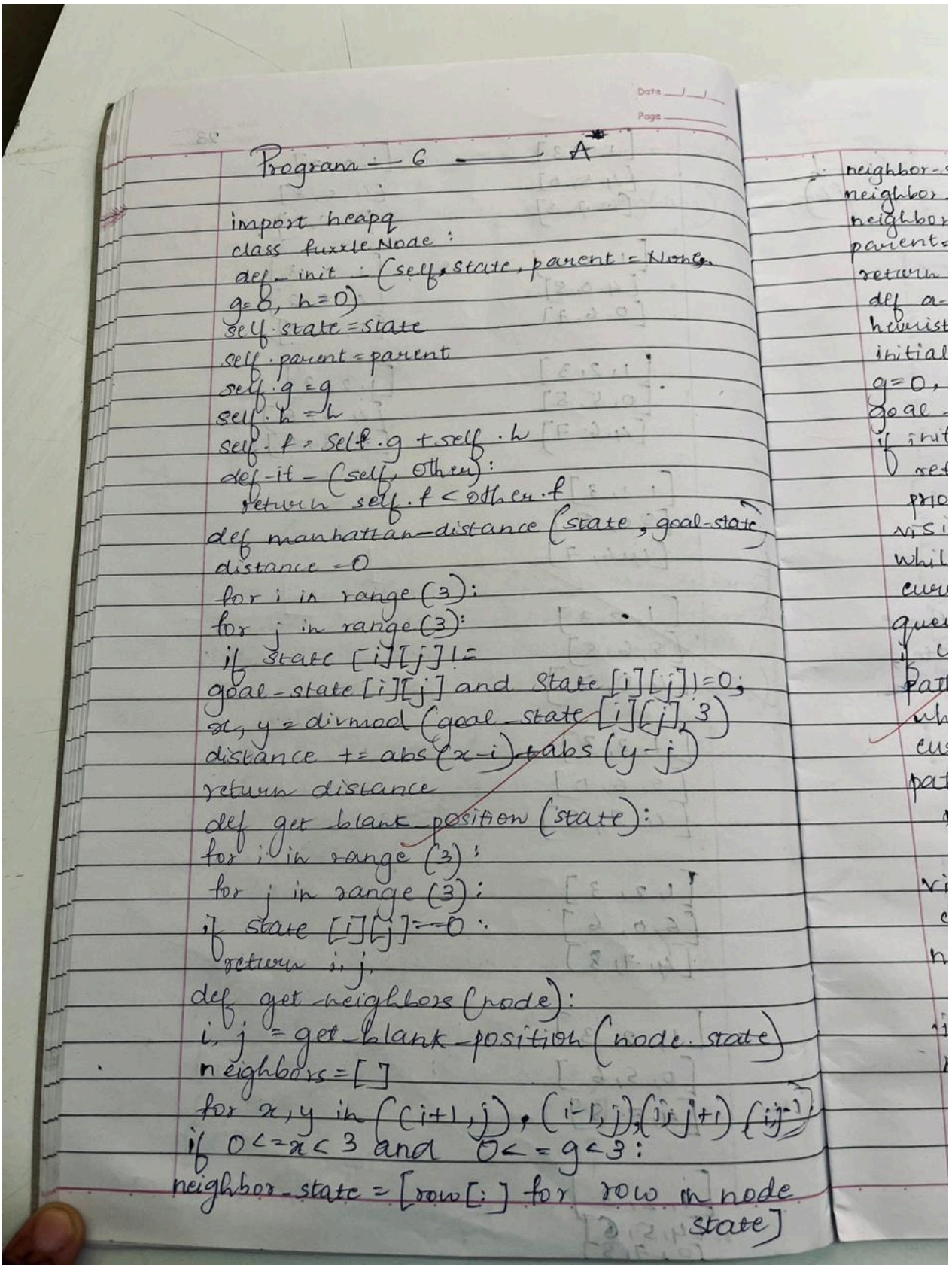
```
    if direction == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if direction == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if direction == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    if direction == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    return temp

#Test 1
print("Example 1")
src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)


# Test 2
print("Example 2")


src = [1,2,3,-1,4,5,6,7,8]
target=[1,2,3,6,4,5,-1,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)


# Test 3
print("Example 3")
```

```
src = [1,2,3,7,4,5,6,-1,8]
target=[1,2,3,6,4,5,-1,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)
```

Output:

```
 Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :5
enter vals :6
enter vals :0
enter vals :7
enter vals :8
enter vals :4
 Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :5
Enter vals :8
Enter vals :6
Enter vals :0
Enter vals :7
Enter vals :4
1. Manhattan distance
2. Misplaced tiles2
 The 8 puzzle is solvable

  1 2 3
  5 6 0
  7 8 4

  1 2 3
  5 0 6
  7 8 4

  1 2 3
  5 8 6
  7 0 4

  1 2 3
  5 8 6
  0 7 4
Steps to reach goal: 3
Total nodes visited:  3

Total generated: 8
```

Notes :



Program — 6 — A*

```
import heapq
class PuzzleNode:
def __init__(self, state, parent=None,
g=0, h=0)
self.state=state
self.parent=parent
self.g=g
self.h=h
self.f=self.g+self.h
def __lt__(self, other):
return self.f<other.f
def manhattan_distance(state, goal_state)
distance=0
for i in range(3):
for j in range(3):
if state[i][j]!=
goal_state[i][j] and state[i][j]!=0:
x, y=divmod(goal_state[i][j], 3)
distance+=abs(x-i)+abs(y-j)
return distance
def get_blank_position(state):
for i in range(3):
for j in range(3):
if state[i][j]==0:
return i, j
def get_neighbors(node):
i, j=get_blank_position(node.state)
neighbors=[]
for x, y in [(i+1, j), (i-1, j), (i, j+1), (i, j-1)]:
if 0<=x<3 and 0<=y<3:
neighbor_state=[row[:] for row in node
state]
```

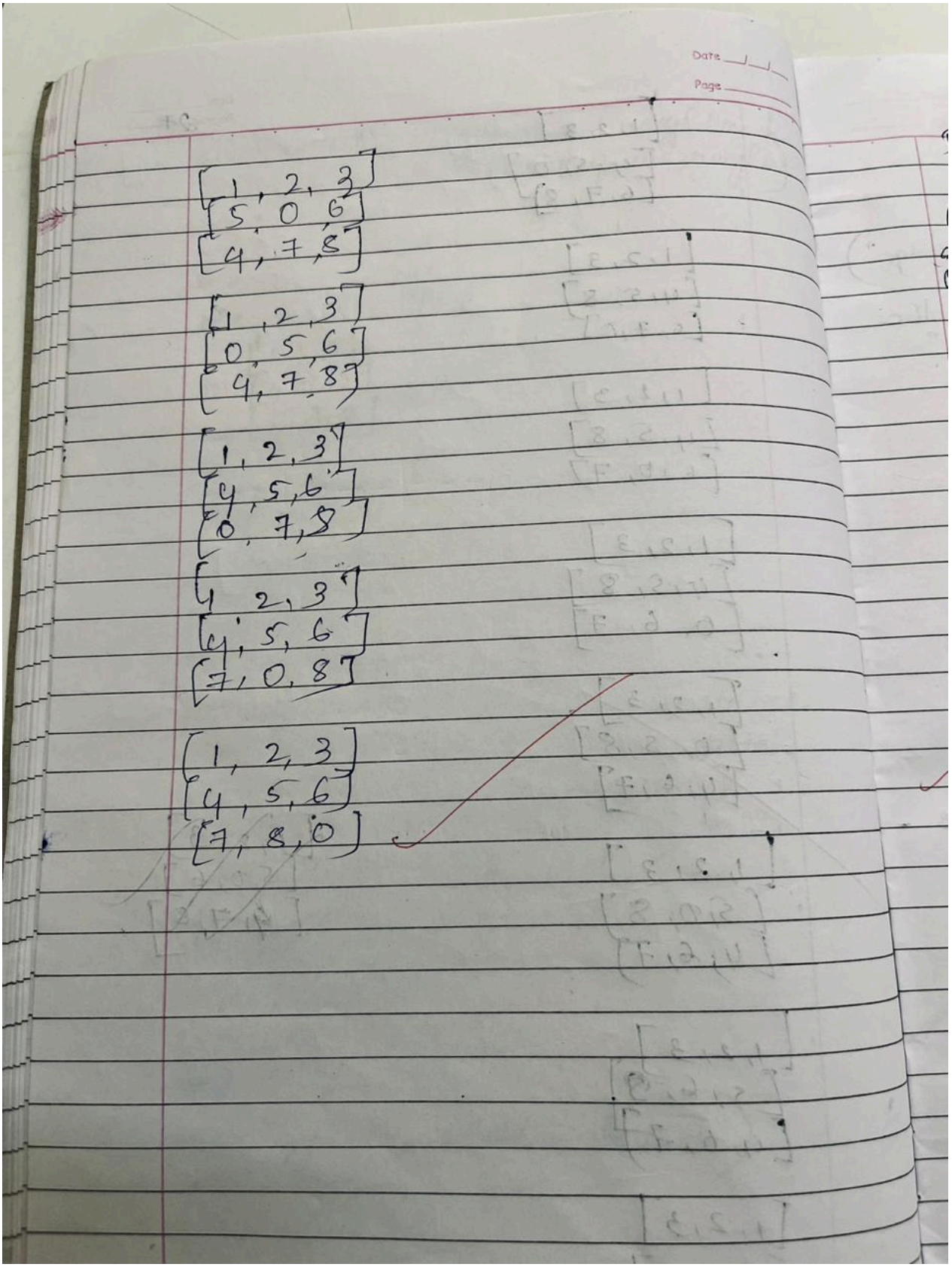
Date / /
Page 25

```
neighbor_state[x][y] = neighbor_state[x][y],
neighbor_state[i][j]
neighbors.append(PuzzleNode(neighbor_state,
parent=node))
return neighbors
def a_star_search(initial_state, goal_state,
heuristic):
initial_node = PuzzleNode(initial_state,
g=0, h=heuristic(initial_state, goal_state))
goal_node = PuzzleNode(goal_state)
if initial_state == goal_state:
    return [initial_state]
priority_queue = [initial_node]
visited_states = set()
while priority_queue:
current_node = heapq.heappop(priority_
queue)
if current_node.state == goal_state:
path = [current_node.state]
while current_node.parent:
current_node = current_node.parent
path.append(current_node.state)
path.reverse()
return path
visited_states.add(tuple(map(tuple,
current_node.state)))
neighbors = get_neighbors(current_node)
for neighbor in neighbors:
if tuple(map(tuple, neighbor.state)) not
in visited_states:
neighbor.g = current_node.g + 1
neighbor.h = heuristic(neighbor.state,
goal_state)
```

```
        neighbor.f = neighbor.g + neighbor.h
        heapq.heappush(priority_queue, neighbor)
    return None

# Example usage
initial_state = [
    [1, 2, 3],
    [4, 0, 5],
    [6, 7, 8]
]
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

heuristic_function = manhattan_distance
path = a_star_search(initial_state, goal_state, heuristic_function)
if path:
print("Solution found:")
for state in path:
for row in state:
print(row)
print()
else:
print("No solution found")
```

OUTPUT:

```
Solution found
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]
```


Date
Page
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
[4, 2, 3]
[4, 5, 6]
[7, 0, 8]
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Program- 6

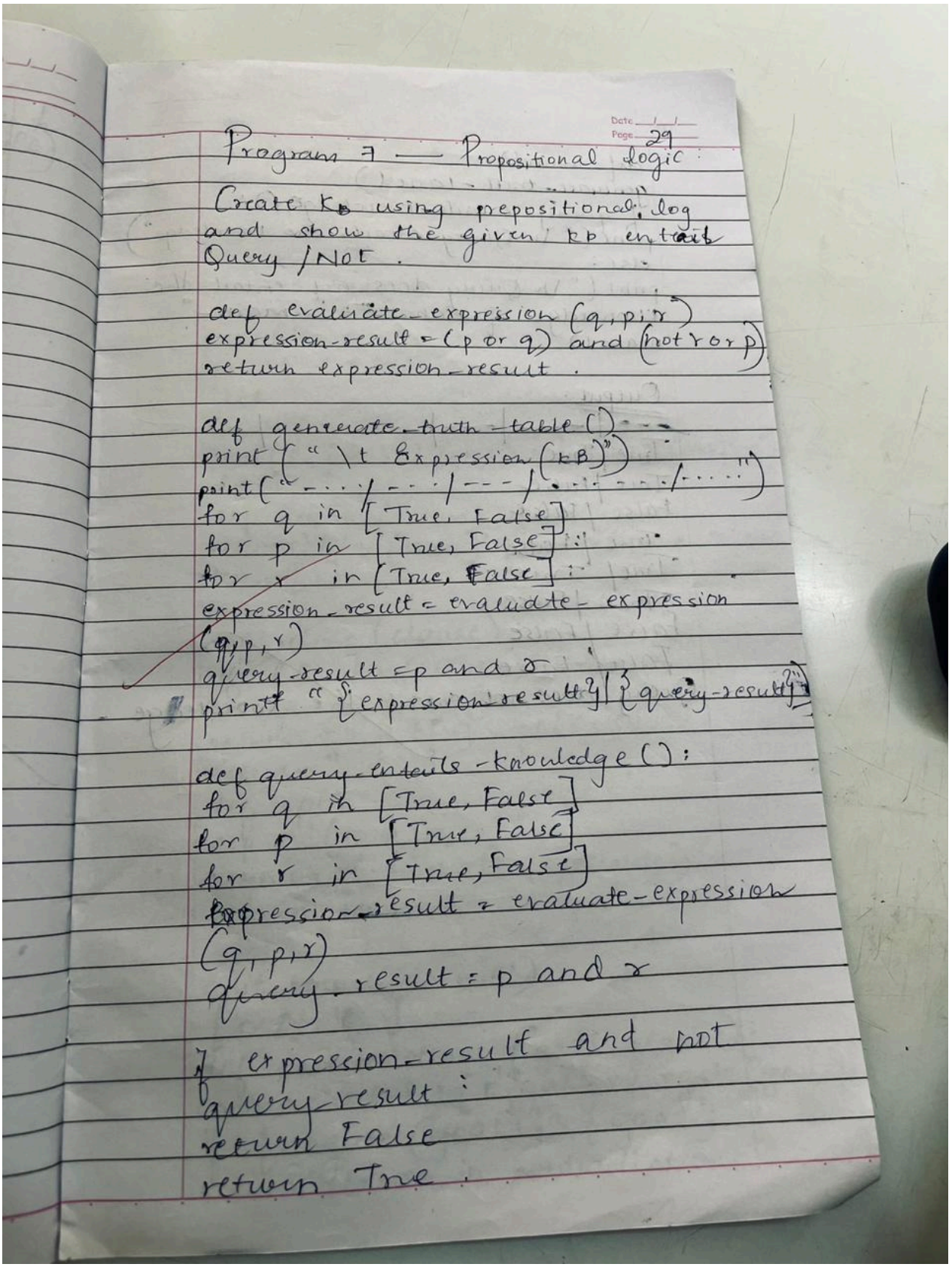
Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not .

Code:

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

def generate_truth_table():
    print(" p | q | r | Expression (KB) | Query (p^r)")
    print("---|---|---|-----------------|------------")

    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p, q, r)
                query_result = p and r

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

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

                if expression_result and not query_result:
                    return False

    return True
```

```
def main():
generate_truth_table()

if query_entails_knowledge():
print("\nQuery entails the knowledge.")
else:
print("\nQuery does not entail the knowledge.")

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

Notes:



Program 7 — Propositional logic:

Create KB using propositional log and show the given KB entails Query / Not.

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

def generate_truth_table():
print("\t Expression (KB)")
print("-.../---/---/.../.../...")
for q in [True, False]
for p in [True, False]:
for r in [True, False]:
expression_result = evaluate_expression
(q, p, r)
query_result = p and r
print "{expression_result}\t\t{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 = p and r

if expression_result and not
query_result:
return False
return True.
```

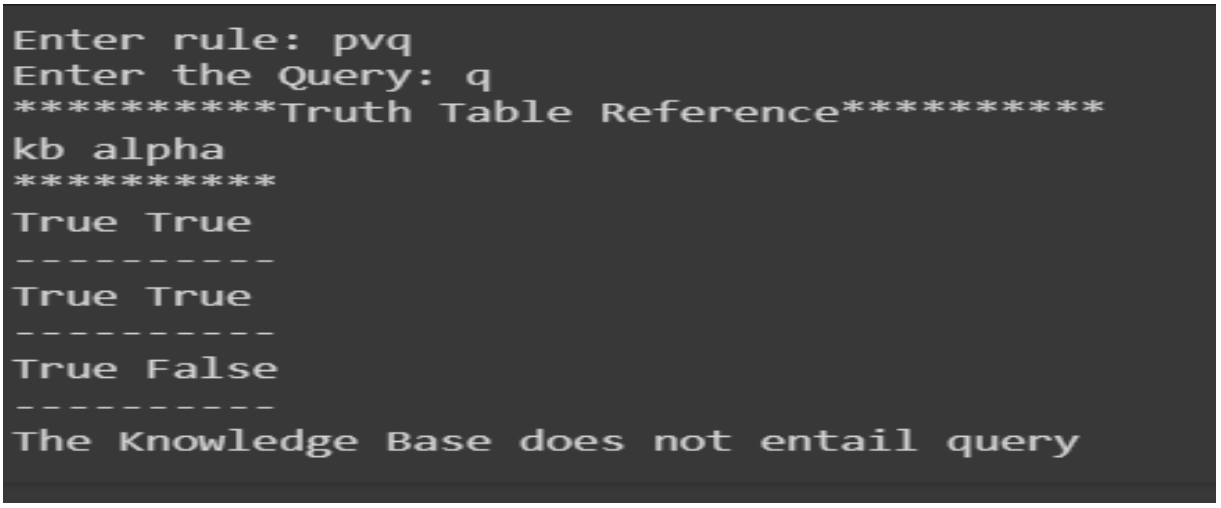
```
def main():
generate_truth_table()
if query_entails_knowledge():
print("\n Query entails the knowledge")
else:
print("\n Query does not entail the knowledge")
if __name__ == "__main__":
main()
```

Output:

True	True		
True	False		
False	False		
True	False		
True	True		
True	False		
False	False		
False	False		

Query does not entail the knowledge.

Output:


Enter rule: pvq
Enter the Query: q
**********Truth Table Reference**********
kb alpha
****
True True

True True

True False

The Knowledge Base does not entail query

Program- 7

Create a knowledge base using prepositional logic and prove the given query using resolution

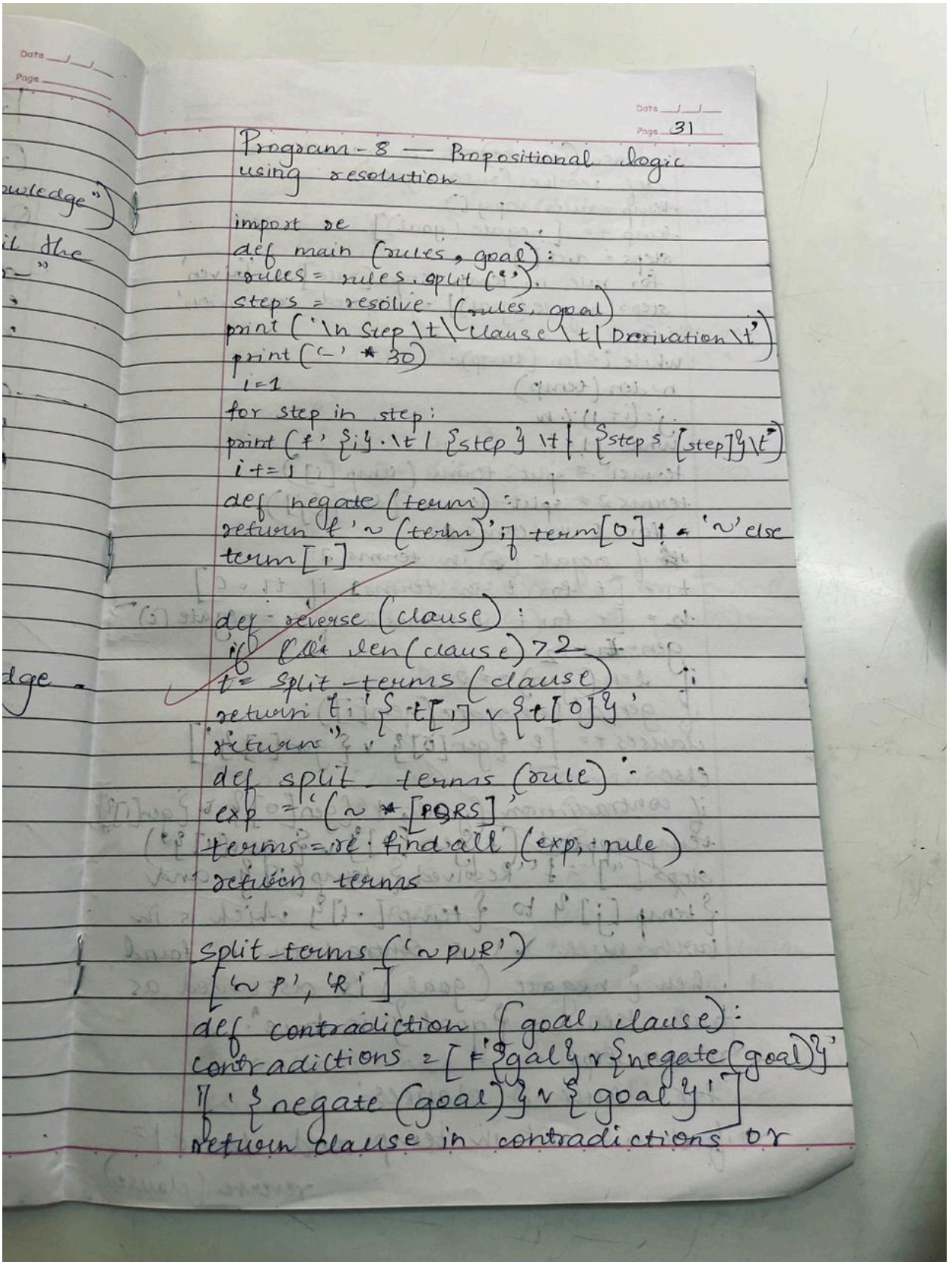
Code:

```
import re


def main(rules, 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 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')
def contradiction(goal, clause):
    contradictions = [ 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) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            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 and clause != reverse(clause) and
                    reverse(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
rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R :
(Rv~P)v(Rv~Q)^(~RvP)^(~RvQ) goal = 'R'
print('Rules: ',rules)
print("Goal: ",goal)
main(rules, goal)

rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
print('Rules: ',rules)
print("Goal: ",goal)
main(rules, goal)

rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q,
(P=>P)=>R, (R=>S)=>~(S=>Q) goal = 'R'
print('Rules: ',rules)

print("Goal: ",goal)
main(rules, goal)
```

Notes:



Program - 8 — Propositional logic using resolution

```
import re
def main(rules, goal):
    rules = rules.split(',')
    steps = resolve(rules, goal)
    print('\n Step \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 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):
    contradictions = [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
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]}']
else:
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
for clause in clauses:
if clause not in temp and clause!=reverse(clause)
```

and reverse (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
rules = 'RV~P PV~Q ~RVP ~RVQ' # (P^Q) <==> R : (RV~P) ^ (RV~Q) ^ (~RVP) ^ (~RVQ)
goal = 'R'
main(rules, goal)
rules = 'PVQ ~PVR ~QVR' # P=>Q, P=>Q= ~PVQ, Q=>R ~QVR
goal = 'R'
main(rules, goal)
rules = 'PVQ PV~R ~PVR RVS RV~Q ~SV~Q' # (P=>Q) ^ (Q=>Q), (P=>P) => R, (R=>S) => ~(S=>Q)
main(rules, 'R')

Output:

Step	Clause	Derivation
1	RV~P	Given
2	RV~Q	Given
3	~RVP	Given
4	~RVQ	Given
5	~R	Negated conclusion
6		Resolved RV~P to RV~R

A contradiction is found when ~R is assumed as true. Hence, R is true.

Output:

Program- 8

Implement unification in first order logic

Code:

```
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:
```

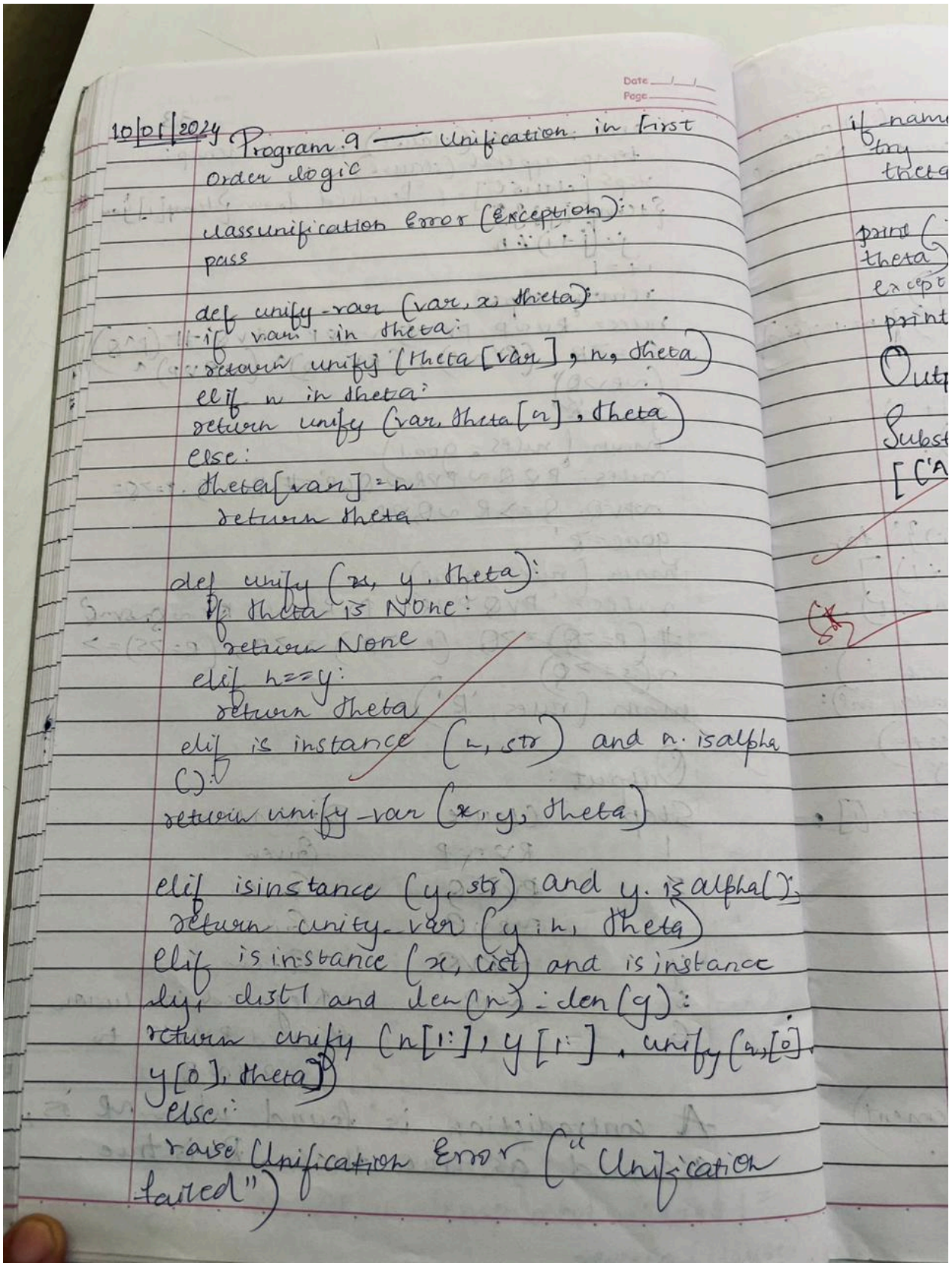
```
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

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:
```

Output:

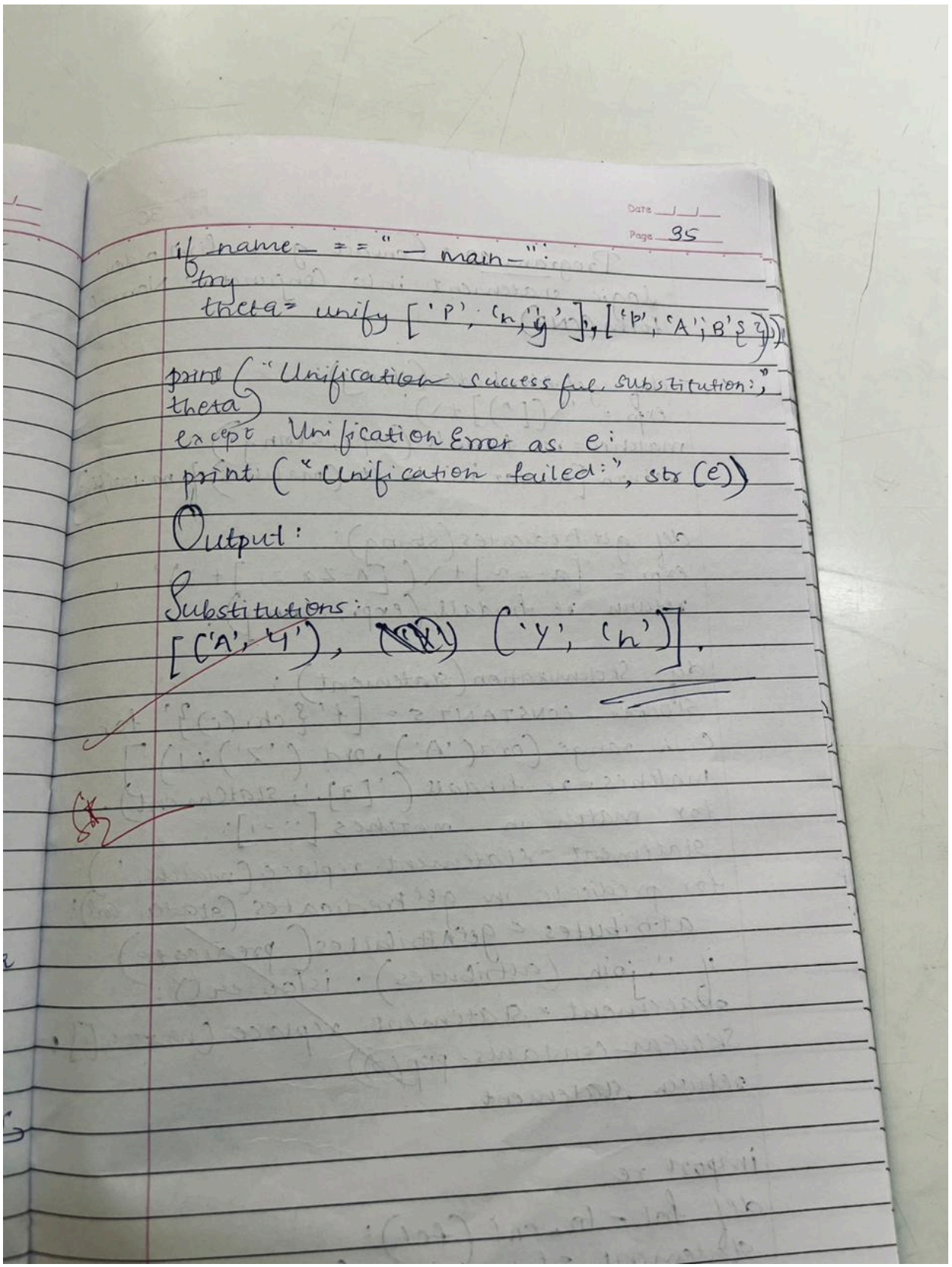
```
Enter the first expression
knows(y,f(x))
Enter the second expression
knows(nithin,N)
The substitutions are:
['nithin / y', 'N / f(x)']
```

Notes :



10/01/2024 Program 9 — Unification in First Order logic

```
class unification Error (exception):
    pass

def unify_var (var, x, theta):
    if var in theta:
        return unify (theta[var], x, theta)
    elif x in theta:
        return unify (var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def unify (x, y, theta):
    if theta is None:
        return None
    elif x==y:
        return theta
    elif is instance (x, str) and x.isalpha
    ():
    return unify_var (x, y, theta)

    elif isinstance (y, str) and y.isalpha():
        return unity_var (y, x, theta)
    elif is instance (x, list) and is instance
    (y, list) and len(x) == len(y):
    return unify (x[1:], y[1:], unify (x[0],
    y[0], theta))
    else:
        raise Unification Error ("Unification
    failed")
```



Date / /
Page 35

```
if __name__ == "__main__":
    try:
        theta = unify ['P', 'x', 'y'], ['P', 'A', B', z']
        print("Unification successful. Substitution:",
        theta)
    except UnificationError as e:
        print("Unification failed:", str(e))
```

Output:

Substitutions:

[('A', 'y'), ('y', 'x')].

Program- 9

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

Code: def getAttributes(string):

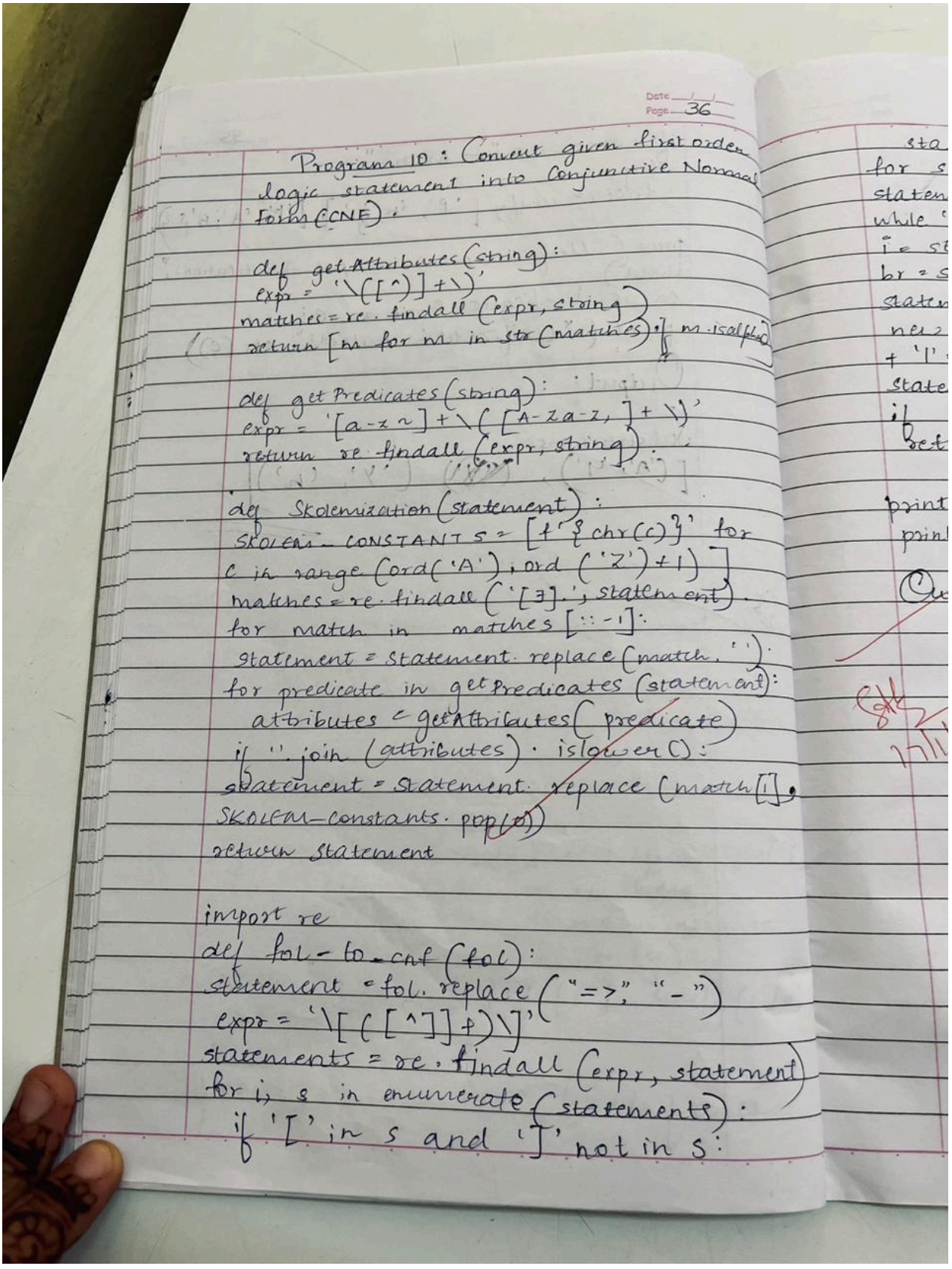
```
 expr = '\([^)]+\)'
 matches = re.findall(expr, string)
 return [m for m in str(matches) if m.isalpha()]
def getPredicates(string):
 expr = '[a-z~]+\([A-Za-z,]+\)'
 return re.findall(expr, string)
def Skolemization(statement):
 SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
 matches = re.findall('[∃].', statement)
 for match in matches[::-1]:
 statement = statement.replace(match, '')
 for predicate in getPredicates(statement):
 attributes = getAttributes(predicate)
 if ''.join(attributes).islower():
 statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
 return statement
import re
def fol_to_cnf(fol):
 statement = fol.replace("=>", "-")
 expr = '\[([^]]+)\]'
 statements = re.findall(expr, statement)
 for i, s in enumerate(statements):
 if '[' in s and ']' not in s:
 statements[i] += ']'
37
 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
 return Skolemization(statement)
print(fol_to_cnf("bird(x)=>~fly(x)"))
print(fol_to_cnf("∃x[bird(x)=>~fly(x)]"))
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:

```
Example 1
FOL: bird(x)=>~fly(x)
CNF:  ~bird(x)|~fly(x)

Example 2
FOL: ∃x[bird(x)=>~fly(x)]
CNF:  [~bird(A)|~fly(A)]

Example 3
FOL: animal(y)<=>loves(x,y)
CNF:  ~animal(y)<|loves(x,y)

Example 4
FOL: ∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]
CNF:  ∀x~[∀y[~animal(y)|loves(x,y)]]|[[loves(A,x)]]

Example 5
FOL: [american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)
CNF:  ~[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]|criminal(x)
```

Notes :



Program 10 : Convert given first order logic statement into Conjunctive Normal Form (CNF).

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

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

def Skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for
    c in range(ord('A'), ord('Z')+1)]
    matches = re.findall('[∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
    for predicate in getPredicates(statement):
        attributes = getAttributes(predicate)
        if ''.join(attributes).islower():
            statement = statement.replace(match[1],
            SKOLEM_constants.pop(0))
    return statement

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

Date / / Page 36

first order
ke Normal

m.isalpha()

for

Date / / Page 37

```
    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
return Skolemization(statement)

print(fol_to_cnf("bird(x) => ~fly(x)"))
print(fol_to_cnf("∃x[bird(x) => ~fly(x)]"))
```

Output.

~bird(x) | ~fly(x)

[~bird(A) | ~fly(A)]

Program-10

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

Code:

```
import re
def isVariable(x):
 return len(x) == 1
    and x.islower()
    and x.isalpha()
def
    getAttributes(str
    ing):
 expr = '\([^)]+\)'
 matches =
    re.findall(expr,
    string)
 return matches
def
    getPredicates(str
    ing):
 expr =
    '([a-z~]+)\([^&|]
    +\)'
 return
    re.findall(expr,
    string)
class Fact:
 def __init__(self,
    expression):
 self.expression =
    expression
 predicate, params =
    self.splitExpress
    ion(expression)
 self.predicate =
    predicate
 self.params = params
 self.result =
    any(self.getConst
    ants())

 def
    splitExpression(s
    elf, expression):
```

```
 predicate =
    getPredicates(exp
    ression)[0]
 params =
    getAttributes(exp
    ression)[0].strip
    ('()').split(',')
 return [predicate,
    params]

 def getResult(self):
39
 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([cons
    tants.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.get
    Variables()):
 if v:
 constants[v] =
    fact.getConstants
    ()[i]
 new_lhs.append(fact)
40
 predicate,
    attributes =
    getPredicates(sel
    f.rhs.expression)
    [0],
str(getAttributes(sel
    f.rhs.expression)
    [0])
 for key in
    constants:
 if constants[key]:
 attributes =
    attributes.replac
    e(key,
    constants[key])
 expr =
    f'{predicate}{att
    ributes}'
 return Fact(expr) if
```

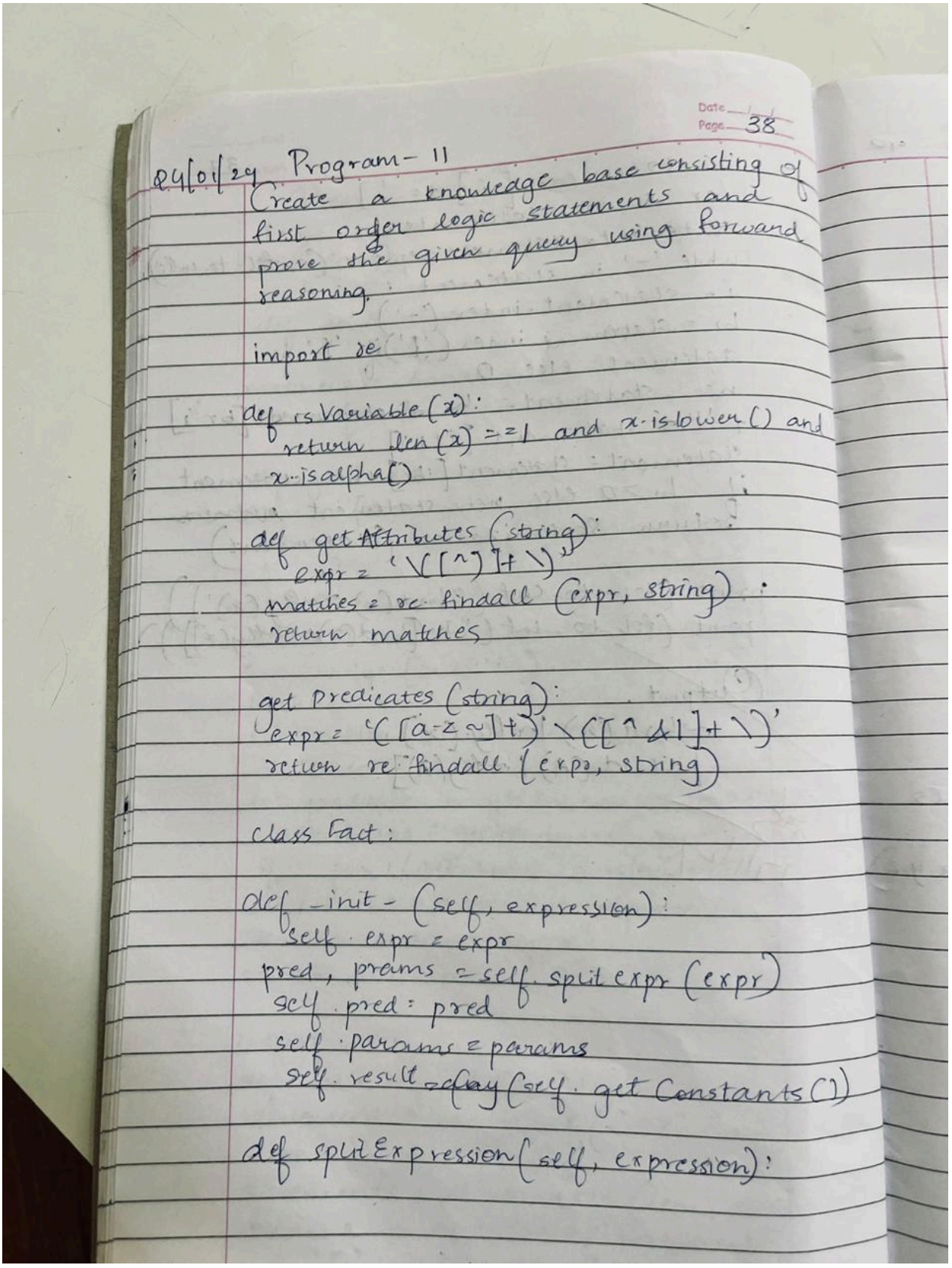
```
    len(new_lhs) and
    all([f.getResult(
    ) for f in
    new_lhs]) else
    None
class KB:
 def __init__(self):
 self.facts = set()
 self.implications =
    set()

 def tell(self, e):
 if '=>' in e:

    self.implications
    .add(Implication(
    e))
 else:

    self.facts.add(Fa
    ct(e))
 for i in
    self.implications
    :
 res =
    i.evaluate(self.f
    acts)
 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
41
```

```
 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,Amer
    ica)=>hostile(x)'
    )
kb.tell('american(Wes
    t)')
kb.tell('enemy(Nono,A
    merica)')
kb.tell('owns(Nono,M1
    )')
kb.tell('missile(x)&o
    wns(Nono,x)=>sell
    s(West,x,Nono)')
kb.tell('american(x)&
    weapon(y)&sells(x
    ,y,z)&hostile(z)=
    >criminal(x)')
kb.query('criminal(x)
    ')
kb.display()
kb_ = KB()
kb_.tell('king(x)&gre
    edy(x)=>evil(x)')
kb_.tell('king(John)'
    )
kb_.tell('greedy(John
    )')
kb_.tell('king(Richar
    d)')
kb_.query('evil(x)')
```

Output:

```
Enter KB: (enter e to exit)
missile(x)=>weapon(x)
missile(m1)
enemy(x,america)=>hostile(x)
american(west)
enemy(china,america)
owns(china,m1)
missile(x)&owns(china,x)=>sells(west,x,china)
american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)
e
Enter Query:
criminal(x)
Querying criminal(x):
        1. criminal(west)
All facts:
        1. criminal(west)
        2. weapon(m1)
        3. owns(china,m1)
        4. enemy(china,america)
        5. sells(west,m1,china)
        6. american(west)
        7. hostile(china)
        8. missile(m1)
```

Notes:

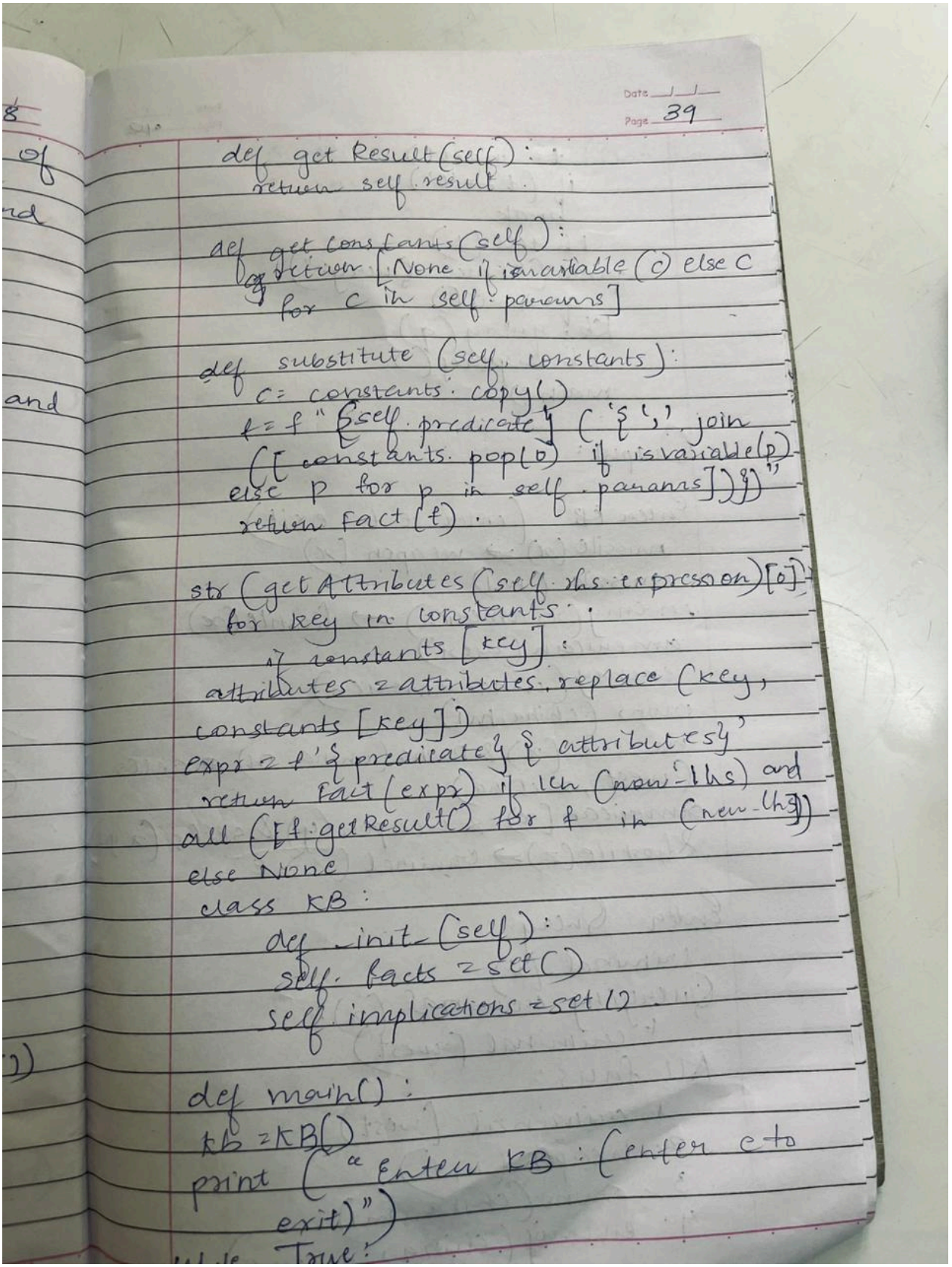


Date
Page 38

24/01/24 Program - 11

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

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

class Fact:

def __init__(self, expression):
    self.expr = expr
    pred, params = self.splitExpr(expr)
    self.pred = pred
    self.params = params
    self.result = any(self.getConstants())

def splitExpression(self, expression):
```



Date / /
Page 39

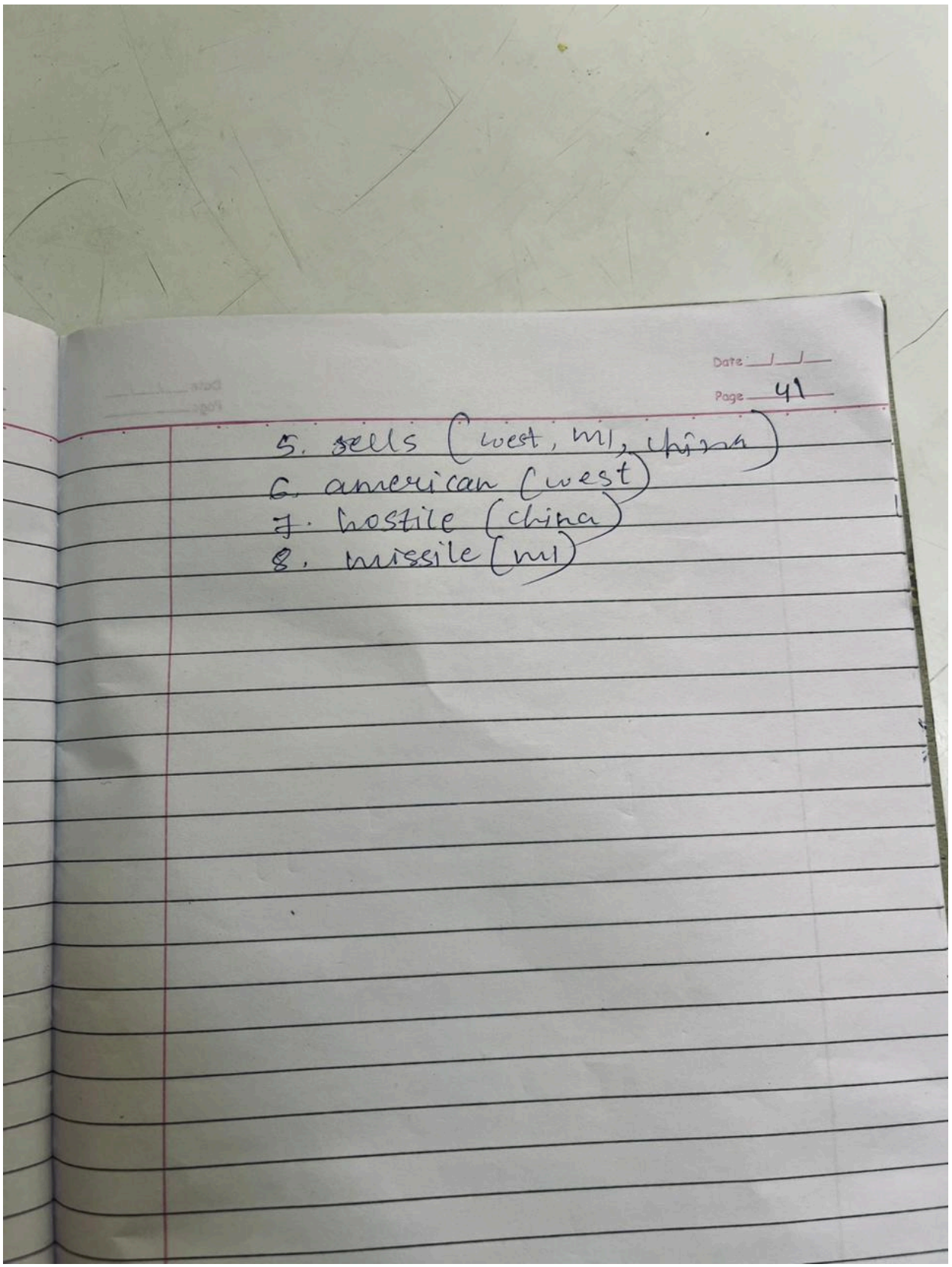
```
def getResult(self):
    return self.result

def getConstants(self):
    return [None if isVariable(c) else c
            for c 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)

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 main():
    kb = KB()
    print("Enter KB (enter e to
        exit)")
while True:
```

```
t = input()
if (t == 'e'):
    break
kb.tell(t)
print("Enter Query:")
q = input()
kb.query(q)
kb.display()
main()
```

Output:

Enter KB: (enter e to exit)
missile(x) ⇒ weapon(x)
missile(m1)
enemy(x, america) ⇒ hostile(x)
american(west)
enemy(china, america)
owns(china, m1)
missile(x) & owns(china, x) ⇒ sells(west, x, china)
american(x) & weapon(y) & sells(x, y, z) & hostile(z) ⇒ criminal(x) e

Enter Query:
criminal(x)
Querying criminal(x):
1. criminal(west)
All facts:
1. criminal(west)
2. weapon(m1)
3. owns(china, m1)
4. enemy(china, america)


Date __/__/__
Page 41
5. sells (west, MI, china)
6. american (west)
7. hostile (china)
8. missile (MI)