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

Lab Record

Artificial Intelligence

Submitted in partial fulfillment for the 5th Semester Laboratory

Bachelor of Technology in
Computer Science and Engineering

Submitted by:

JEEVANTHI KASHYAP
1BM21CS080

Department of Computer Science and Engineering B.M.S.
College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
Oct-Feb 2024

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (20CS5PCAIP) laboratory has been carried out by Jeevanthi Kashyap (1BM21CS080) during the 5th Semester October-February 2024.

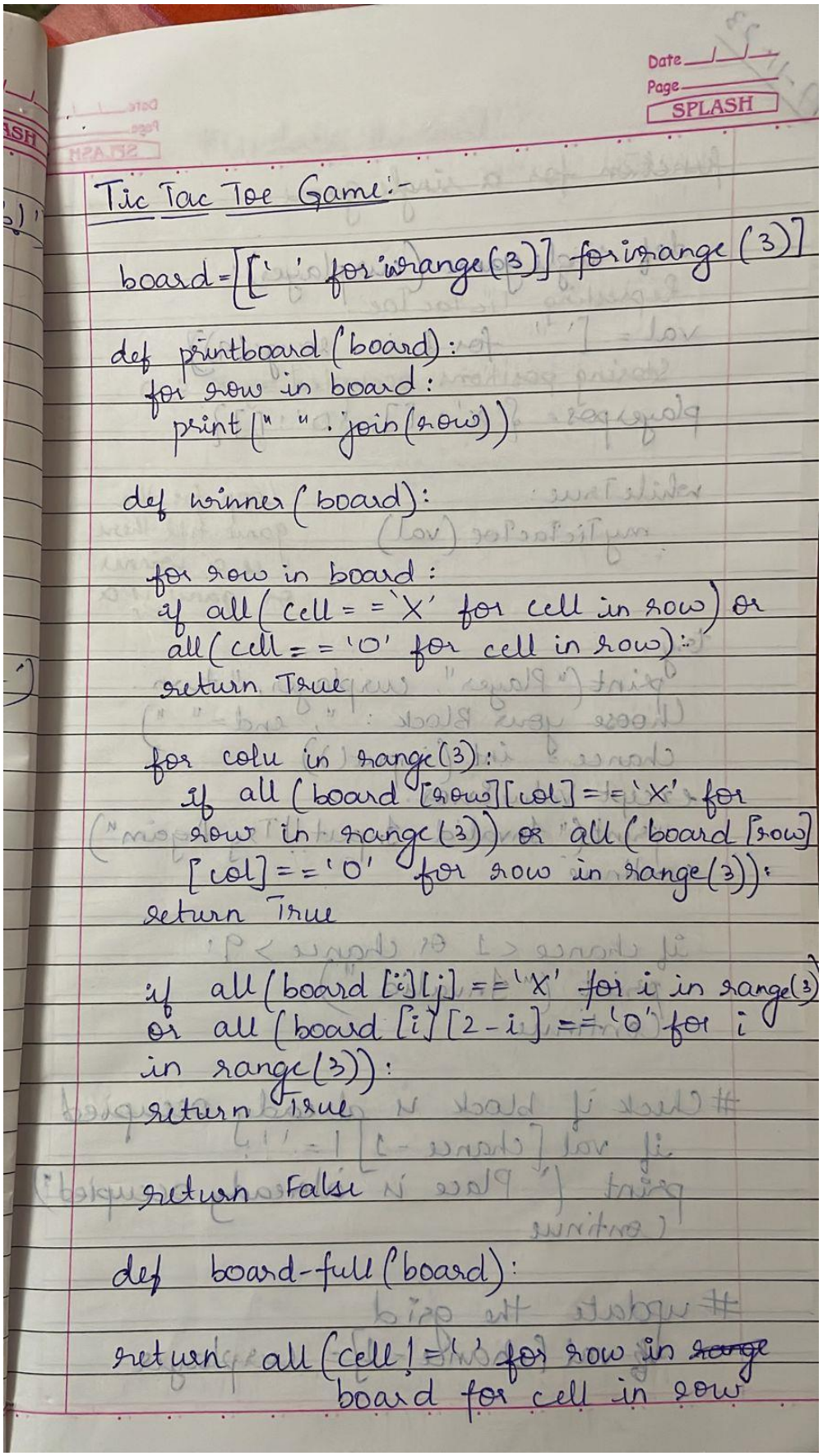
Signature of the Faculty Incharge:

Prof. Asha G.R
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	1 – 7
2.	8 Puzzle Breadth First Search Algorithm	8 - 12
3.	8 Puzzle Iterative Deepening Search Algorithm	13 - 17
4.	8 Puzzle A* Search Algorithm	18 – 21
5.	Vacuum Cleaner	22 – 27
6.	Knowledge Base Entailment	28 – 34
7.	Knowledge Base Resolution	35 –38
8.	Unification	39 – 43
9.	FOL to CNF	44 – 47
10.	Forward reasoning	48– 51

PROGRAM-1



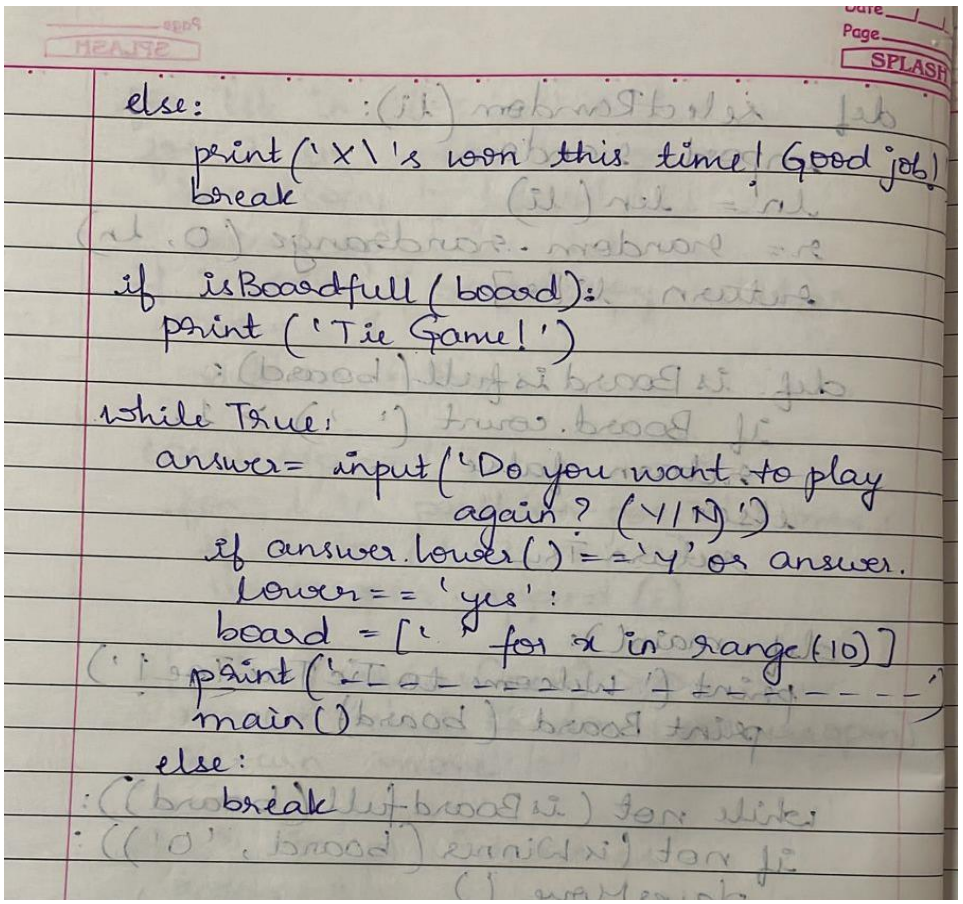
Tic Tac Toe Game:-

```
board = [[' ' for i in range(3)] for i in range(3)]

def printboard(board):
    for row in board:
        print(" ".join(row))

def winner(board):
    for row in board:
        if all(cell == 'X' for cell in row) or
        all(cell == 'O' for cell in row):
        return True

    for colu in range(3):
        if all(board[row][col] == 'X' for
        row in range(3)) or all(board[row]
        [col] == 'O' for row in range(3)):
    return True

    if all(board[i][j] == 'X' for i in range(3)
    or all(board[i][2-i] == 'O' for i
    in range(3)):
    return True

    return False

def board-full(board):

return all(cell != ' ' for row in board for cell in row
```

```
import random
BOARD_SIZE = 9
ROWS = 3
COMPUTER_SYMBOL = 'X'
USER_SYMBOL = 'O'
def board(tic):
    for i in range(0, BOARD_SIZE, ROWS):
    print("+" + "-" * 29 + "+")
    print("|" + " " * 9 + "|" + " " * 9 + "|"
          + " " * 9 + "|")
    print("|" + " " * 3, tic[0+i], " " * 3 + "|"
          + " " * 3, tic[1+i], " " * 3 + "|"
          + " " * 3, tic[2+i], " " * 3 +
          "|")
    print("|" + " " * 9 + "|" + " " * 9 + "|" + " " *
          9 + "|")
    print("+" + "-" * 29 + "+")
def comp_move(tic):
    num = random.radiant(1,9)
    while num not in tic:
        num = random.radiant(1,9)
    tic[num-1] = COMPUTER_SYMBOL
    return num
def user_move(tic):
    num = int(input("Enter a number on the
              board: "))
while num not in tic:
    num = int(input("Enter a number on the
              board: "))
    tic[num-1] = USER_SYMBOL
def check_winner(tic, num):
    if [tic[0] == tic[4] == tic[8] or tic[2] ==
      tic[4] == tic[6]:
```

#update the grid.

```
return True
if tic[num] == tic[num-3] == tic[num-6]:
    return True
if tic[num//3*3] == tic[num//3*3+1]
    == tic[num//3*3+2]:
    return True
return False
try:
    tic = [i for i in range(1, BOARD_SIZE+1)]
    count = 0
    print(tic)
    board(tic)
    while count != BOARD_SIZE:
        if count % 2 == 0:
            print("Computer's turn:")
            num = comp_move(tic)
            board(tic)
            count += 1
        else:
            print("Your turn:")
            user_move(tic)
            board(tic)
            count += 1
        if count >= 5:
            if check_winner(tic, num-1):
                print("Winner is", tic[num-1])
                break
    elif count == BOARD_SIZE:
        print("It is a tie")
except Exception as e:
    print(f"Error: {e}")
```

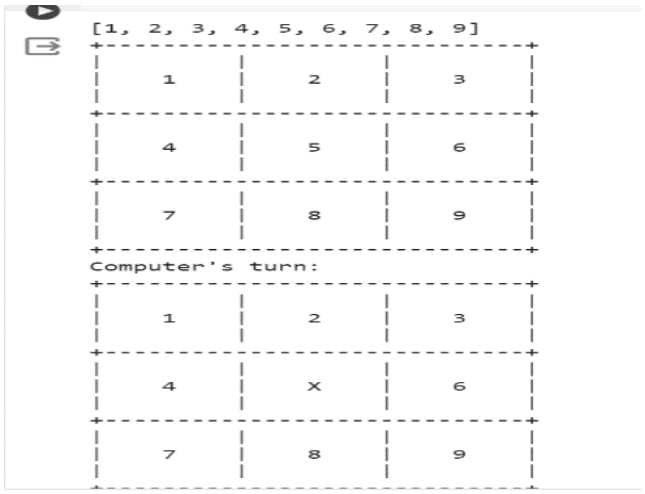


CODE:

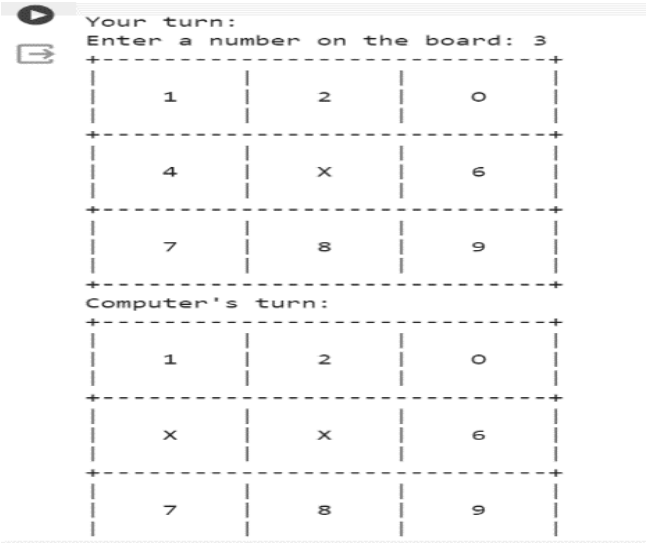
```
import random

# Constants
BOARD_SIZE = 9
ROWS = 3
COMPUTER_SYMBOL = 'X'
USER_SYMBOL = 'O'

def board(tic):
    for i in range(0, BOARD_SIZE, ROWS):
        print("+" + "-" * 29 + "+")
        print("|" + " " * 9 + "|" + " " * 9 + "|" + " " * 9 + "|")
        print(
            "|" + " " * 3, tic[0 + i], " " * 3 + "|" + " " * 3, tic[1 +
i], " " * 3 + "|" + " " * 3, tic[2 + i], " " * 3 + "|")
        print("|" + " " * 9 + "|" + " " * 9 + "|" + " " * 9 + "|")
    print("+" + "-" * 29 + "+")

def computer_move(tic):
    num = random.randint(1, 9)
    while num not in tic:
        num = random.randint(1, 9)
    tic[num - 1] = COMPUTER_SYMBOL
    return num
```

```
def user_move(tic):
    num = int(input("Enter a number on the board: "))
    while num not in tic:
        num = int(input("Enter a number on the board: "))
    tic[num - 1] = USER_SYMBOL

def check_winner(tic, num):
    # Check for a winner based on the updated move
    if tic[0] == tic[4] == tic[8] or tic[2] == tic[4] == tic[6]:
        return True
    if tic[num] == tic[num - 3] == tic[num - 6]:
        return True
    if tic[num // 3 * 3] == tic[num // 3 * 3 + 1] == tic[num // 3 * 3 +
2]:
        return True
    return False

try:
    tic = [i for i in range(1, BOARD_SIZE + 1)]
    count = 0
    print(tic)
    board(tic)

    while count != BOARD_SIZE:
        if count % 2 == 0:
            print("Computer's turn:")
            num = computer_move(tic)
            board(tic)
            count += 1
        else:
            print("Your turn:")
            user_move(tic)
            board(tic)
            count += 1
        if count >= 5:
            if check_winner(tic, num - 1):
                print("Winner is", tic[num - 1])
                break
            elif count == BOARD_SIZE:
                print("It's a tie!")

except Exception as e:
    print(f"Error: {e}")
```

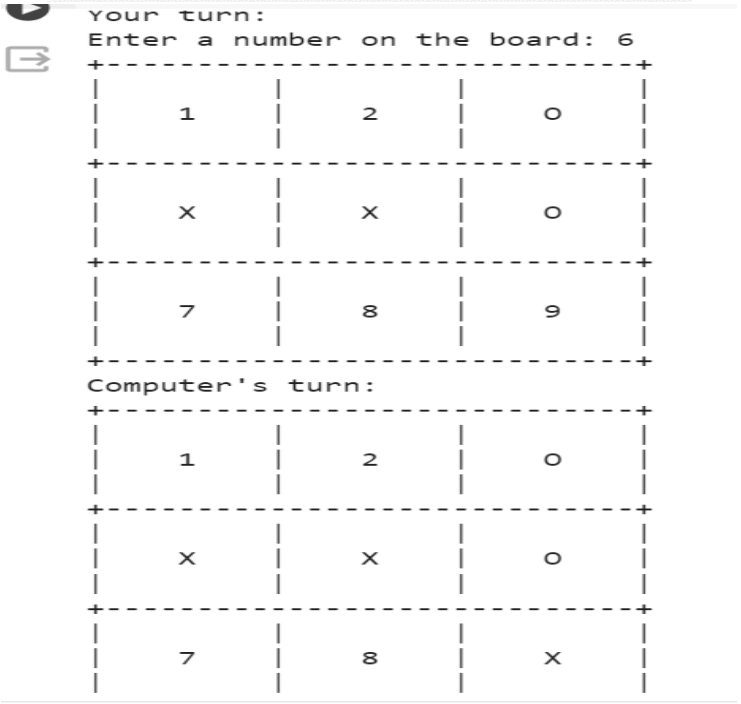
OUTPUT SCREENSHOT



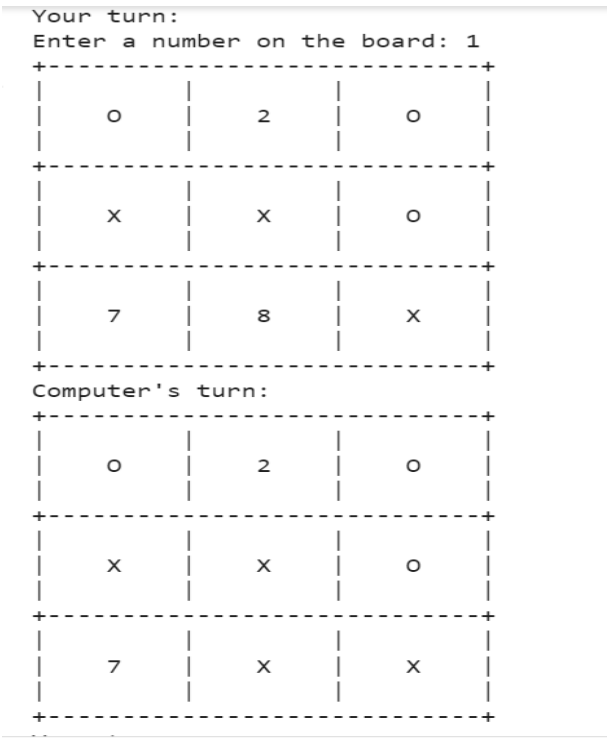
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
+-------+-------+-------+
|   1   |   2   |   3   |
+-------+-------+-------+
|   4   |   5   |   6   |
+-------+-------+-------+
|   7   |   8   |   9   |
+-------+-------+-------+
Computer's turn:
+-------+-------+-------+
|   1   |   2   |   3   |
+-------+-------+-------+
|   4   |   X   |   6   |
+-------+-------+-------+
|   7   |   8   |   9   |
+-------+-------+-------+
```

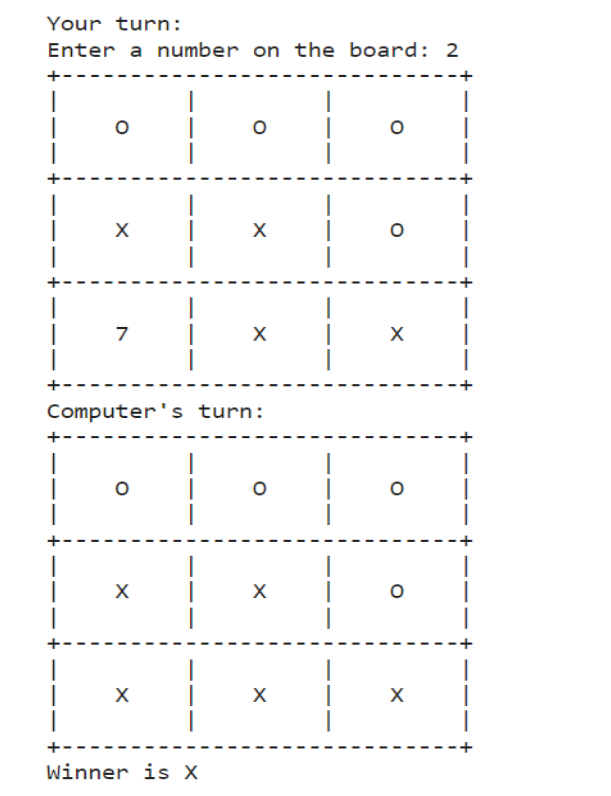


```
Your turn:
Enter a number on the board: 3
+-------+-------+-------+
|   1   |   2   |   O   |
+-------+-------+-------+
|   4   |   X   |   6   |
+-------+-------+-------+
|   7   |   8   |   9   |
+-------+-------+-------+
Computer's turn:
+-------+-------+-------+
|   1   |   2   |   O   |
+-------+-------+-------+
|   X   |   X   |   6   |
+-------+-------+-------+
|   7   |   8   |   9   |
+-------+-------+-------+
```

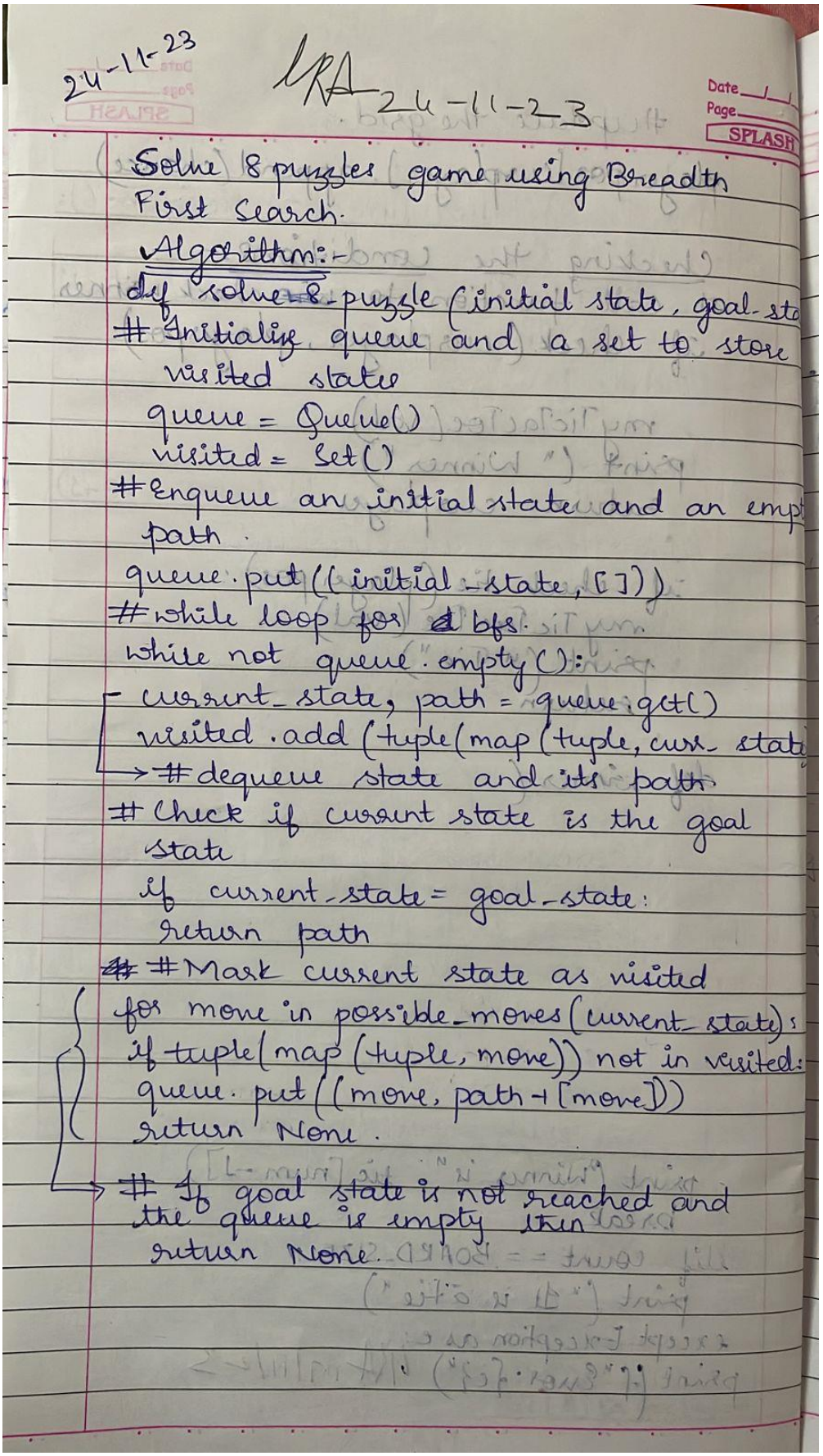


```
Your turn:
Enter a number on the board: 6
+-------+-------+-------+
|   1   |   2   |   O   |
+-------+-------+-------+
|   X   |   X   |   O   |
+-------+-------+-------+
|   7   |   8   |   9   |
+-------+-------+-------+
Computer's turn:
+-------+-------+-------+
|   1   |   2   |   O   |
+-------+-------+-------+
|   X   |   X   |   O   |
+-------+-------+-------+
|   7   |   8   |   X   |
+-------+-------+-------+
```


Your turn:
Enter a number on the board: 1
O 2 O
X X O
7 8 X
Computer's turn:
O 2 O
X X O
7 X X


Your turn:
Enter a number on the board: 2
O O O
X X O
7 X X
Computer's turn:
O O O
X X O
X X X
Winner is X

PROGRAM-2



24-11-23 LRA 24-11-23

Solve 8 puzzles game using Breadth First Search.

Algorithm:-

```
def solve_8_puzzle(initial_state, goal_sta
#Initialize queue and a set to store
  visited states
  queue = Queue()
  visited = Set()
#Enqueue an initial state and an empt
  path.
  queue.put((initial_state, []))
#while loop for bfs.
while not queue.empty():
  current_state, path = queue.get()
  visited.add(tuple(map(tuple, curr_state
  #dequeue state and its path.
#Check if current state is the goal
  state
  if current_state = goal_state:
    return path
#Mark current state as visited
for move in possible_moves(current_state):
  if tuple(map(tuple, move)) not in visited:
  queue.put((move, path + [move]))
  return None.
# If goal state is not reached and
  the queue is empty then
  return None.
```

Date / /
Page
SPLASH

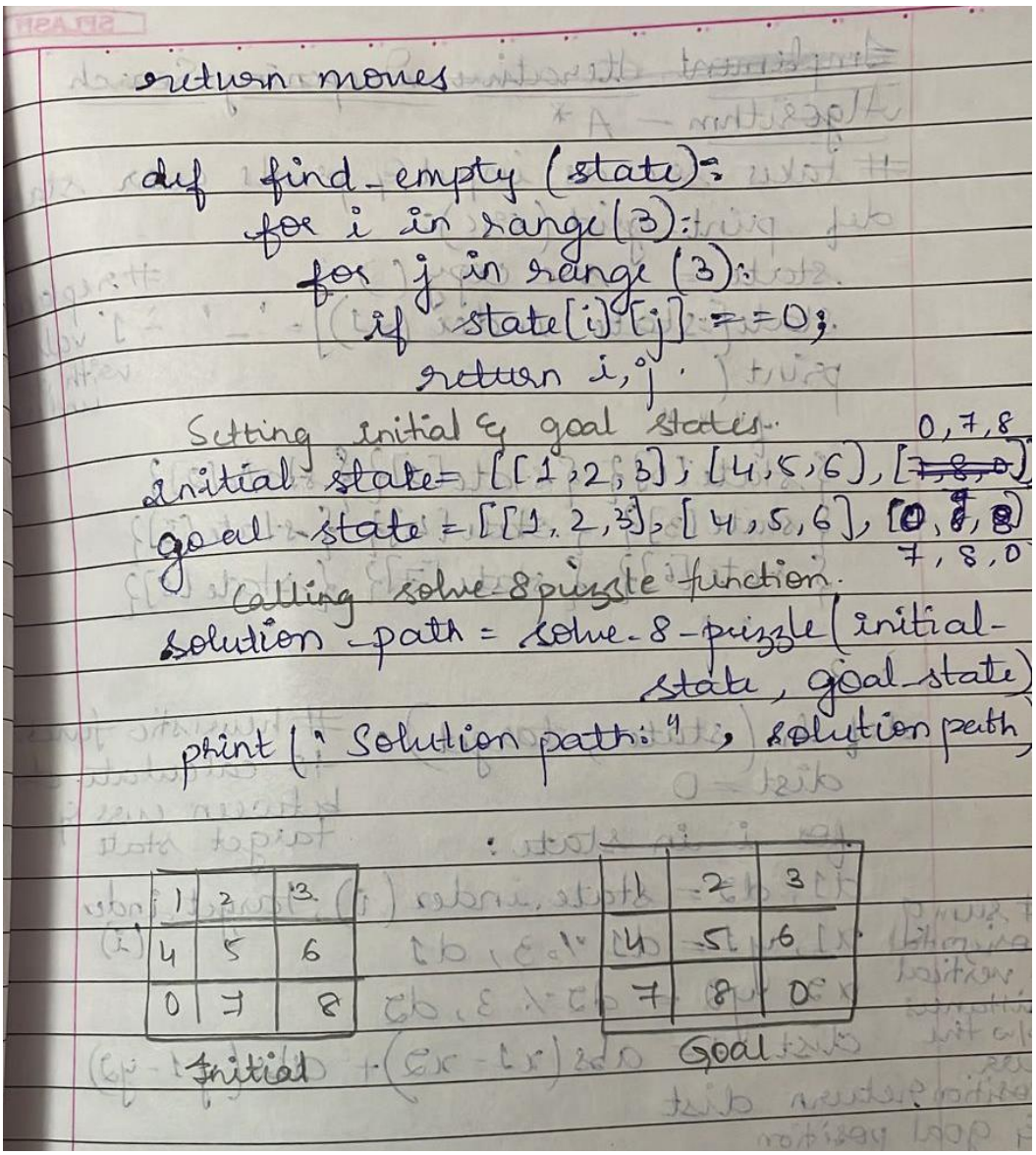
import Queue class from queue module

```
from queue import Queue
def solve_8_puzzle(initial_state, goal_state):
    queue = Queue()
    visited = set()
    queue.put((initial_state, []))
    while not queue.empty():
        current_state, path = queue.get()
        visited.add(tuple(map(tuple, current_state)))
        if current_state == goal_state:
            return path
        for move in possible_moves(current_state):
            if tuple(map(tuple, move)) not in visited:
                queue.put((move, path + [move]))
    return None
def possible_moves(state):
    empty_row, empty_col = find_empty_tile(state)
    moves = []
    # Move empty space up
    if empty_row > 0:
        new_state = [row[:] for row in state]
        new_state[empty_row][empty_col],
        new_state[empty_row-1][empty_col]
        = new_state[empty_row-1][empty_col],
        new_state[empty_row][empty_col]
        moves.append(new_state)
```

Identify the position of the empty space

Generate valid moves based on the current state

```
# Move empty spaces left
if empty_col > 0:
    new_state = [row[:] for row in
                 state]
    new_state[empty_row][empty_col]
        new_state[empty_row][empty_col-1]
    = new_state[empty_row][empty_col-1],
    new_state[empty_row][empty_col]
    moves.append(new_state)

# Move empty spaces down
if empty_row < 2:
    new_state = [row[:] for row in
                 state]
    new_state[empty_row][empty_col],
    new_state[empty_row+1]
        [empty_col] = new_state[empty_
                    row+1][empty_col],
    new_state[empty_row][empty_col]
    moves.append(new_state)

# Move empty space right
if empty_col < 2:
    new_state = [row[:] for row in
                 state]
    new_state[empty_row][empty_col],
    new_state[empty_row][empty_col+1]
    = new_state[empty_row][empty_col],
      new_state[empty_row][empty_col]
    moves.append(new_state)
```



CODE AND OUPUT

JEEVANTHI KASHYAP 1BM21CS080

1]:

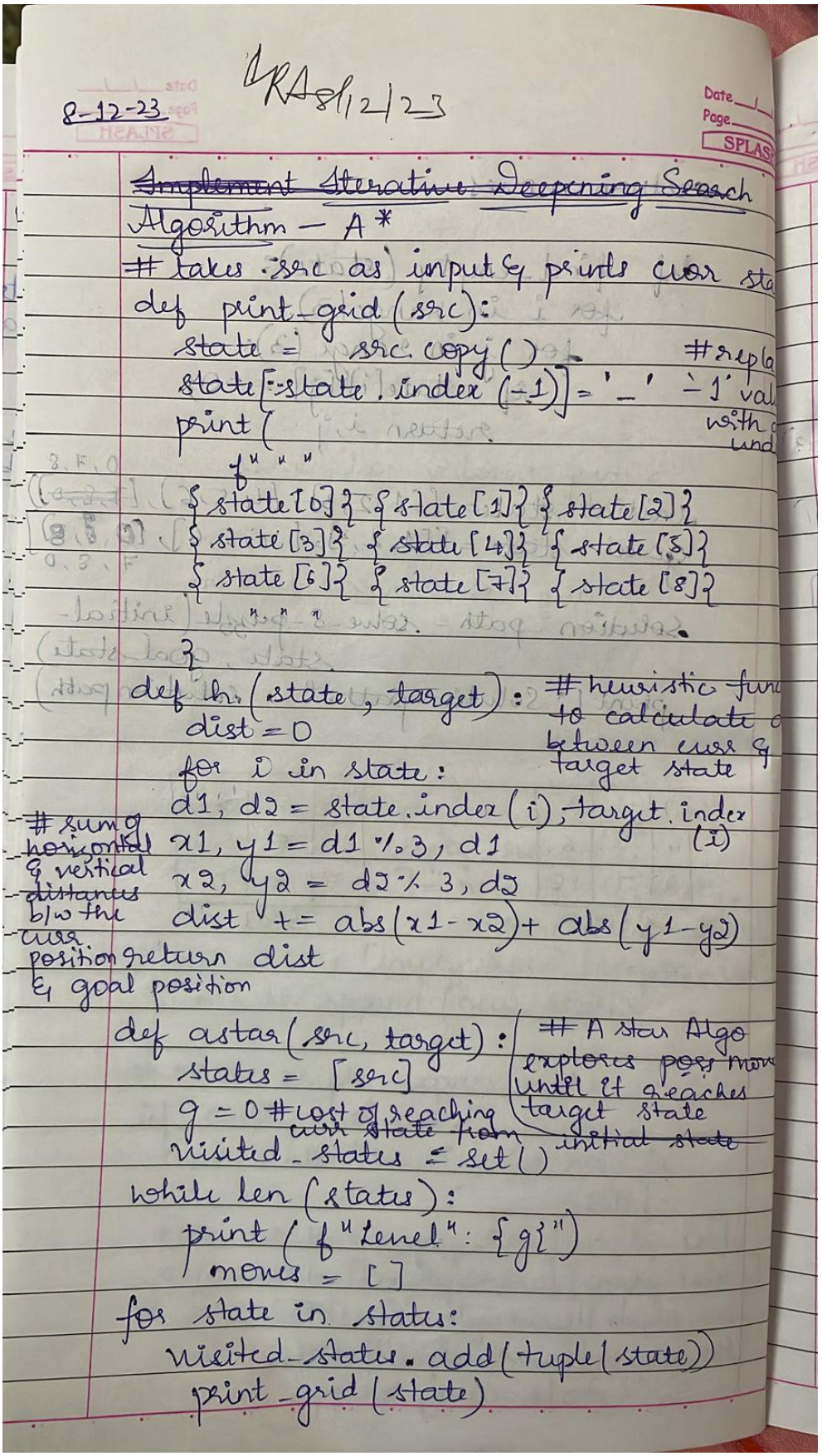
```
def solve_8_puzzle(initial, goal):
    queue = []
    queue.append(initial)
    visited = []

    while len(queue) > 0:
        current_state = queue.pop(0)
        display(current_state)

        visited.append(current_state)

        if current_state == goal:
            print("Success")
            return

        moves = possible_moves(current_state, visited)

        for move in moves:
            if move not in visited and move not in queue:
                queue.append(move)

    print("Failure: Target state not reached")

def display(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])
    print("-----------")
```

```python
def possible_moves(state, visited_states):
    blank_space = state.index(0)
    directions = []

    if blank_space not in [0, 1, 2]:
        directions.append('u')
    if blank_space not in [6, 7, 8]:
        directions.append('d')
    if blank_space not in [0, 3, 6]:
        directions.append('l')
    if blank_space not in [2, 5, 8]:
        directions.append('r')

    poss_moves = []

    for direction in directions:
        poss_moves.append(generate(state, direction, blank_space))

    return [move for move in poss_moves if move not in visited_states]

def generate(state, direction, blank_space):
    temp = state.copy()

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

```python
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
solve_8_puzzle(initial, goal)
```

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

PROGRAM-3



8-12-23 KRA 8/12/23

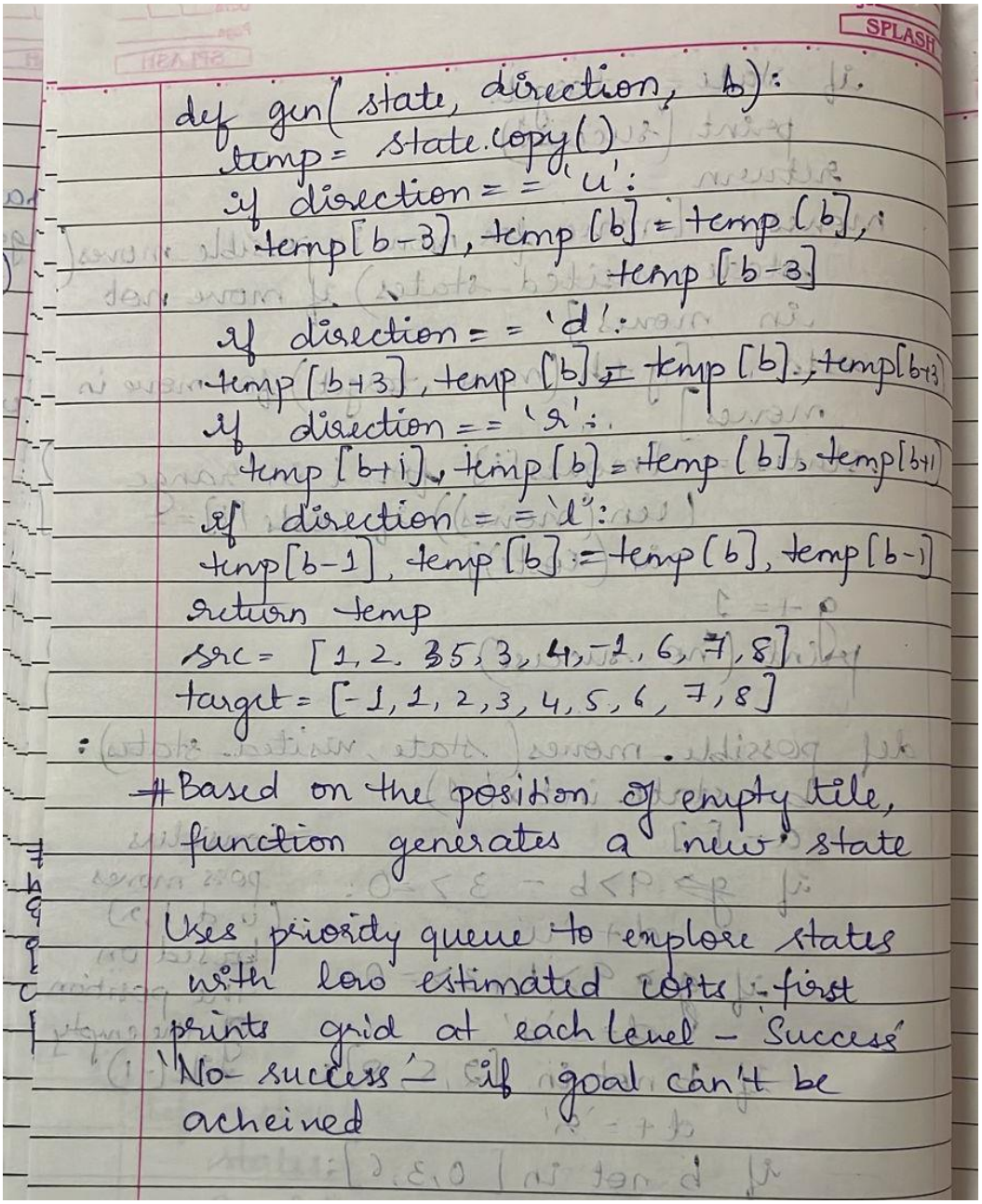
~~Implement Iterative Deepening Search~~
Algorithm – A*

```
# takes 'src' as input & prints corr sta
def print_grid(src):
    state = src.copy()                    #repla
    state[:state.index(-1)] = '_'         -1' val
    print(                                with
        f"""                              und
{state[0]} {state[1]} {state[2]}
{state[3]} {state[4]} {state[5]}
{state[6]} {state[7]} {state[8]}
""")
def h(state, target):                     # heuristic fun
    dist = 0                              to calculate d
    for i in state:                       between curr & target state
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3          # sum of horizontal & vertical distances b/w the curr position & goal position
        x2, y2 = d2 % 3, d2 // 3
        dist += abs(x1-x2) + abs(y1-y2)
    return dist
def astar(src, target):                   # A star Algo explores pos moves until it reaches target state
    states = [src]
    g = 0 # cost of reaching curr state from initial state
    visited_states = set()
    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.add(tuple(state))
            print_grid(state)
```

Date / /
Page
SPLASH

```
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("no success")

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if 0 > b - 3 >= 0:
        d += 'u'
    if 0 > 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]
```

Considers poss moves (u, d, l, r) based on the position of the empty tile (-1)



CODE

```
def print_grid(src):  # print the grid
    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):  # a* algo
    states = [src]
    g = 0
    visited_states = set()
    while len(states):
        print(f"Level: {g}")
        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]  # fn=gn+hn
        states = [moves[i]
                  for i in range(len(moves)) if costs[i] == min(costs)]  #
min cost
        g += 1
    print("No success")


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


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

astar(src, target)
```

OUTPUT:

```
Level: 0

1 2 5
3 4 _
6 7 8

Level: 1

1 2 _
3 4 5
6 7 8

Level: 2

1 _ 2
3 4 5
6 7 8

Level: 3

_ 1 2
3 4 5
6 7 8

Success
```

PROGRAM-4

Date / /
Page
SPLASH

Iterative Deepening Search Algorithm :-

dfs (src, target, limit, visited_states)

function performs dfs with a limit.
possible moves from the current state (src) to the target state (target).
→ If current state is the target state it returns true
→ If depth limit is reached, it returns false
→ Appends the current state to the list of visited states (visited_states)
→ generates possible moves using poss_moves function
→ Calls itself for each valid move.
Recursive call returns true
If none of the moves lead to target within the depth, it returns false.

possible_moves(state, visited_states)
Returns a list of possible moves from the given state except the ones that are visited.

gen(state, move, blank):
this function will generate a new state based on performing the specified move
l, r, d, u - possible moves from a given state

iddfs (src, target, depth)
function calls dfs with ~~iterative~~ increasing depth limits until goal state is reached

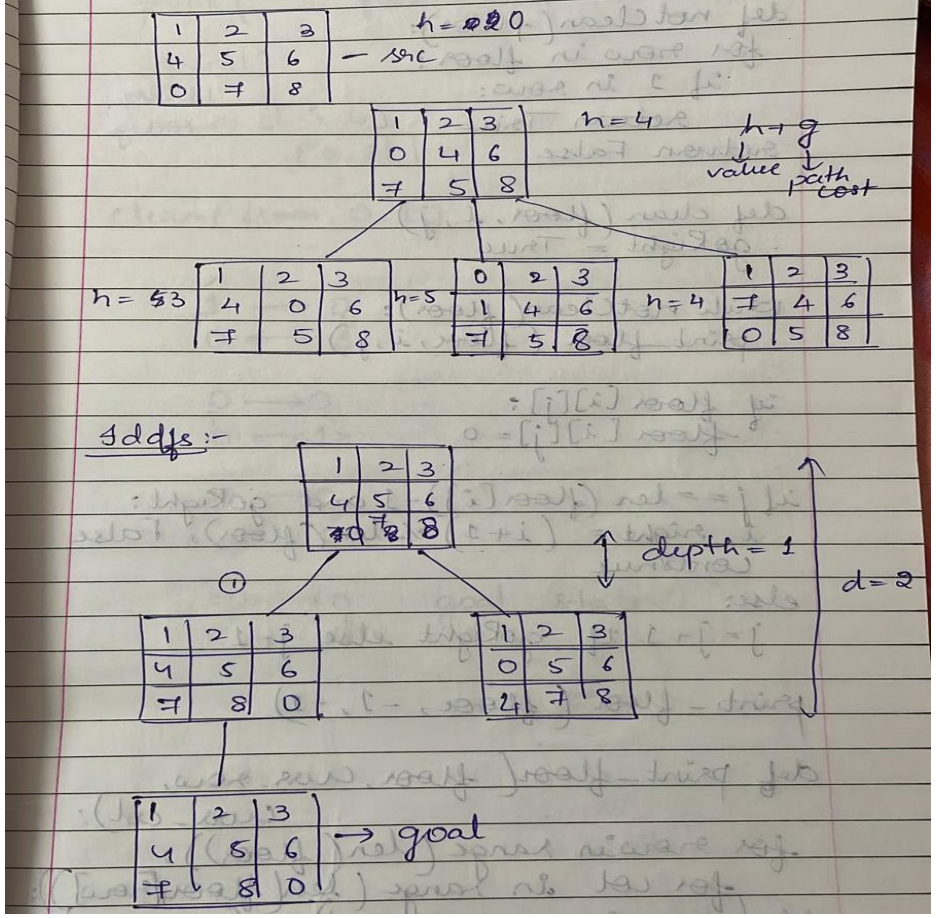
initialises empty list - visited - states for each depth limit ~~(i++)~~

calls dfs with the current depth limit (i+1)

if dfs returns true, prints true and exits
If none of the searches reach the goal state, returns false.

deepens its search until it finds a solution or reaches the specified depth limit

A* :-



CODE:

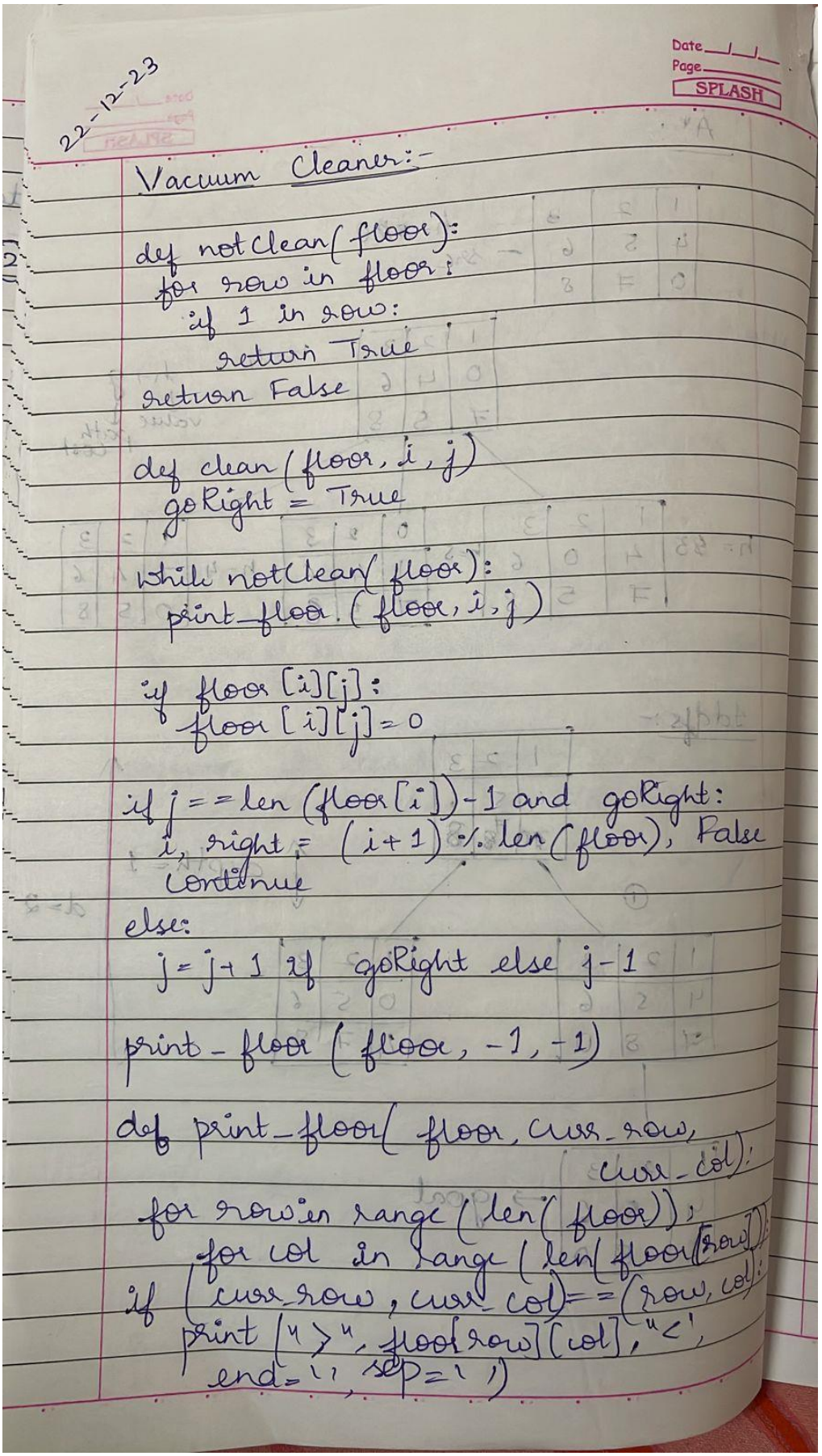
```
def iddfs(puzzle, goal, get_moves):
    import itertools
    def dfs(route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves(route[-1]):
            if move not in route:
                next_route = dfs(route + [move], depth - 1)
                if next_route:
                    return next_route

    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route


def possible_moves(state):
    b = state.index(0)  # ) indicates White space -> so b has index of it.
    d = []  # direction

    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')

    poss_moves = []
    for i in d:
        poss_moves.append(gen(state, i, b))
    return poss_moves


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


# calling IDDFS
initial = [1, 2, 3, 0, 4, 6, 7, 5, 8]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

route = iddfs(initial, goal, possible_moves)

if route:
    print("Success! 8 puzzle problem solved")
    print("Path:", route)
else:
    print("Failed to find a solution")
```

OUTPUT:

```
Success! 8 puzzle problem solved
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

PROGRAM-5

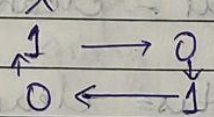


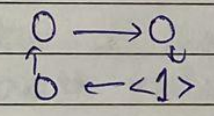
22-12-23

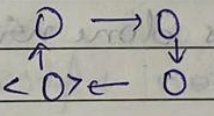
Vacuum Cleaner:-

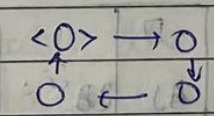
```
def notClean(floor):
    for row in floor:
        if 1 in row:
            return True
    return False

def clean(floor, i, j)
    goRight = True

while notClean(floor):
    print_floor(floor, i, j)

    if floor[i][j]:
        floor[i][j] = 0

    if j == len(floor[i]) - 1 and goRight:
        i, right = (i+1) % len(floor), False
        continue
    else:
        j = j+1 if goRight else j-1

print_floor(floor, -1, -1)

def print_floor(floor, curr_row,
                curr_col):
    for row in range(len(floor)):
        for col in range(len(floor[row])):
            if (curr_row, curr_col) == (row, col):
                print(">", floor[row][col], "<",
                      end='', sep='')
```

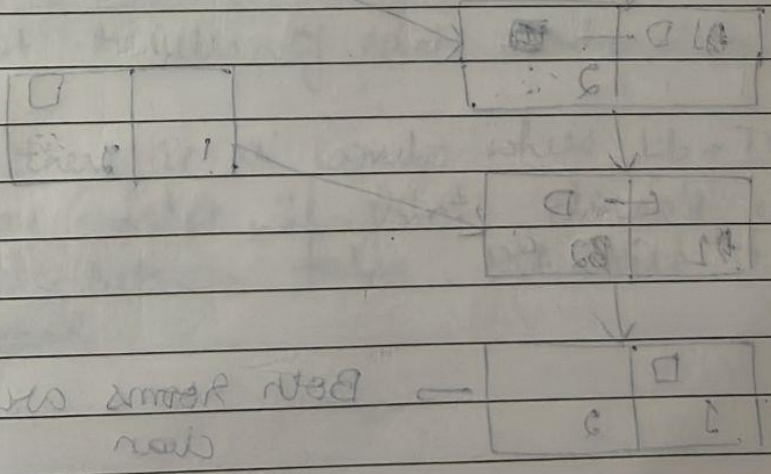
```
else:
    print("", floor[row][col], "", end=' ',
          sep=' ')
print()
print()
floor = [[1,0],
         [0,1]]

clean(floor, 0, 0)
```











22-12-23

22-12-23

Date / /
Page
SPLASH

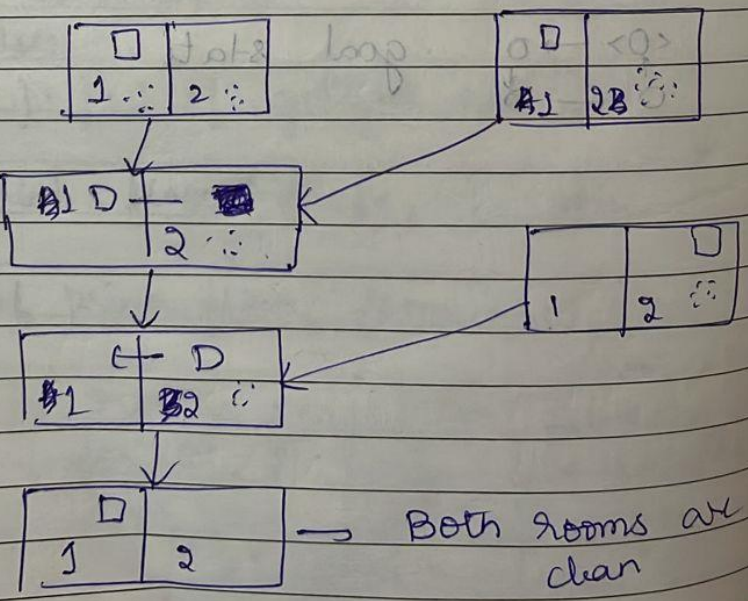
Algorithm :-

Vacuum Cleaner Agent :-

Initialize starting and goal state
Goal state → clean the 2 rooms

If status = dirty
then clean
else if
location = A1 and status = clean,
then return to right
location = B2 and status = clean,
then return to left

else exit
If both rooms are clean, then
the vacuum cleaner is done with
its task.



CODE:

```
def clean():
        # 0 indicates Clean and 1 indicates Dirty
    goal = {'A': '0', 'B': '0'}
    cost = 0

    loc_input = input("Enter Location of Vacuum")
    status_input1 = input("Enter status of " + loc_input)
    status_input2 = input("Enter status of other room")
    print("Initial Location Condition" + str(goal))

    if loc_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input1 == '1':
            print("Location A is Dirty.")
            goal['A'] = '0'
            cost += 1 #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input2 == '1':

                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1
                print("COST for moving RIGHT" + str(cost))
                goal['B'] = '0'
                cost += 1 #cost for suck
                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_input1 == '0':
            print("Location A is already clean ")
            if status_input2 == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))

                goal['B'] = '0'
                cost += 1
                print("Cost for SUCK" + str(cost))
                print("Location B has been Cleaned. ")
            else:
```

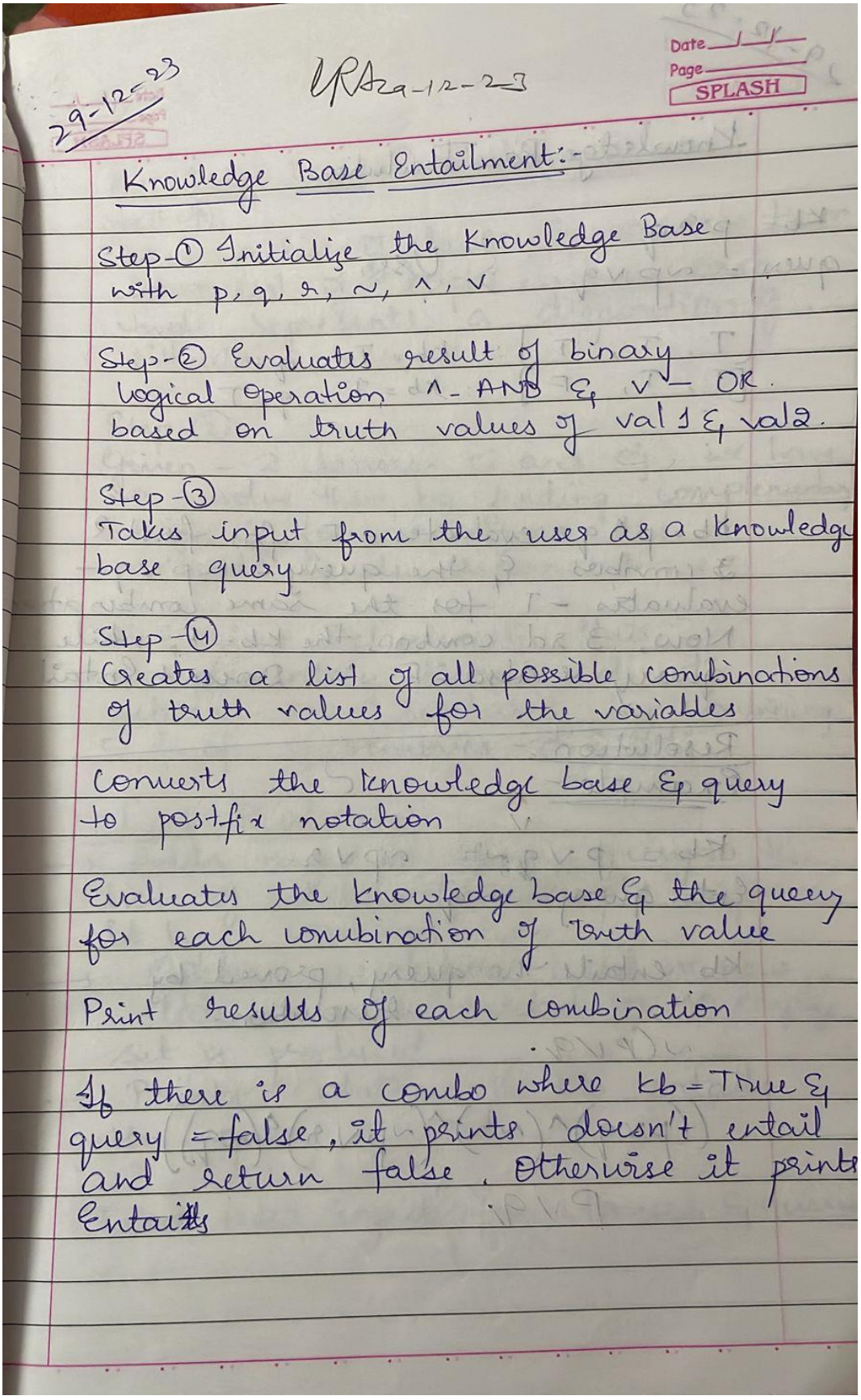
```
            print("No action " + str(cost))
            print(cost)

            print("Location B is already clean.")

    else:
        print("Vacuum is placed in location B")

        if status_input1 == '1':
            print("Location B is Dirty.")

            goal['B'] = '0'
            cost += 1
            print("COST for CLEANING " + str(cost))
            print("Location B has been Cleaned.")

            if status_input2 == '1':

                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1
                print("COST for moving LEFT" + str(cost))

                goal['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_input2 == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1
                print("COST for moving LEFT " + str(cost))

                goal['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)
    print("Performance Measurement: " + str(cost))


clean()
```

OUTPUT:

```
Enter Location of Vacuumb
Enter status of b1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

PROGRAM-6



29-12-23

URA 29-12-23

Date / /
Page
SPLASH

Knowledge Base Entailment:-

Step-① Initialize the Knowledge Base with p, q, r, ~, ^, v

Step-② Evaluates result of binary logical operation ^ - AND & v - OR. based on truth values of val1 & val2.

Step-③
Takes input from the user as a Knowledge base query

Step-④
Creates a list of all possible combinations of truth values for the variables

Converts the knowledge base & query to postfix notation

Evaluates the knowledge base & the query for each combination of truth value

Print results of each combination

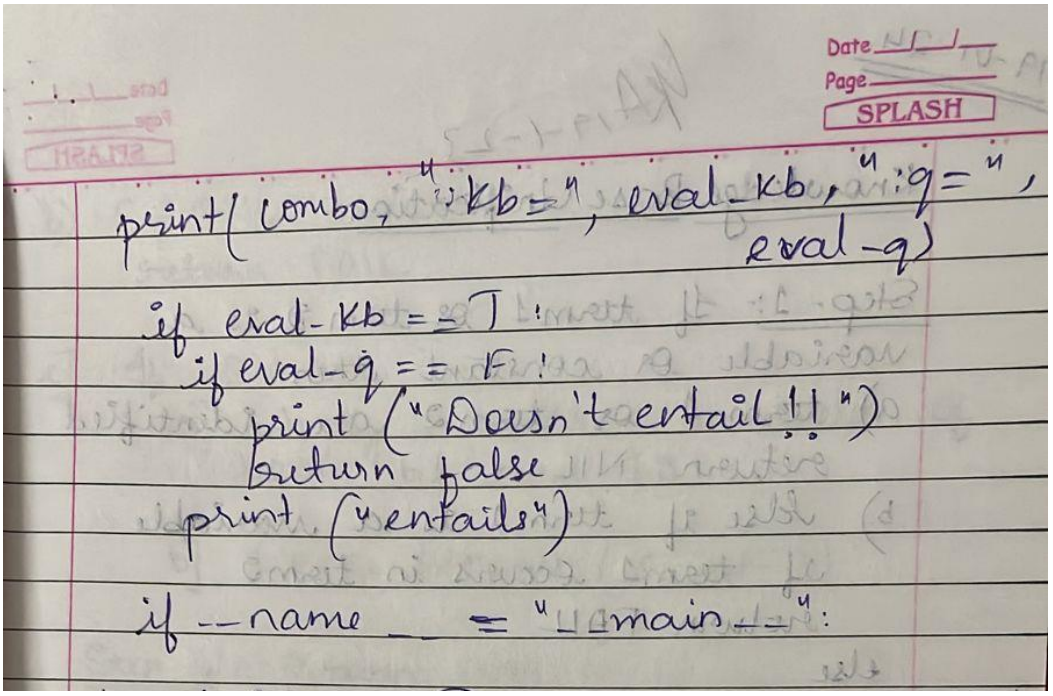
If there is a combo where kb = True & query = false, it prints doesn't entail and return false. Otherwise it prints Entails

Knowledge Base Entailment :-

```
variable = {"p":0, "q":1, "r":2}
priority = {"~":3, "v":1, "^":2}

def eval(i, val1, val2):
    if i == "^":
        return val2 and val1
    return val2 or val1

def isOperand(c):
    return c.isalpha() and c != "v"

def isLeftParenthesis(c):
    return c == "("

def isRightParenthesis(c):
    return c == ")"

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

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

def hasLessOrEqualPrior(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 isLeftParenthesis(c):
            stack.append(c)
        elif isRightParenthesis(c):
            operator = stack.pop()
            while not isLeftParenthesis(operator):
                postfix += operator
                operator = stack.pop()
        else:
            while (not isEmpty(stack)) and
            hasLessOrEqualPrior(c, peek(stack)):
                postfix += stack.pop()
            stack.append(c)
    while not isEmpty(stack):
        postfix += stack.pop()
    return postfix

def evaluatePost(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:
```

```
else:
    val1 = stack.pop()
    val2 = stack.pop()
stack.append(!eval(i, val2, val1))
return stack.pop()
stack.append(not val1)
else:
    val1 = stack.pop()
    val2 = stack.pop()
    stack.append(eval(i, val2, val1)
return stack.pop()

def CheckEntail():
kb = input("Enter kb:")
query = input("Enter query")
combos = [
[T, T, T],
[T, T, F],
[T, F, T],
[T, F, F],
[F, T, T],
[F, T, F],
[F, F, T],
[F, F, F],
]
postfix_kb = toPostfix(kb)
postfix_q = toPostfix(query)

for combo in combo:
eval_kb = evaluatePostfix(postfix_kb,
                          combo)
eval_q = evaluatePostfix(postfix_q,
                         combo)
```



CODE:

```
variable = {"p": 0, "q": 1, "r": 2}
priority = {"~": 3, "v": 1, "^": 2}


def _eval(i, val1, val2):
    if i == "^":
        return val2 and val1
    return val2 or val1


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]
```

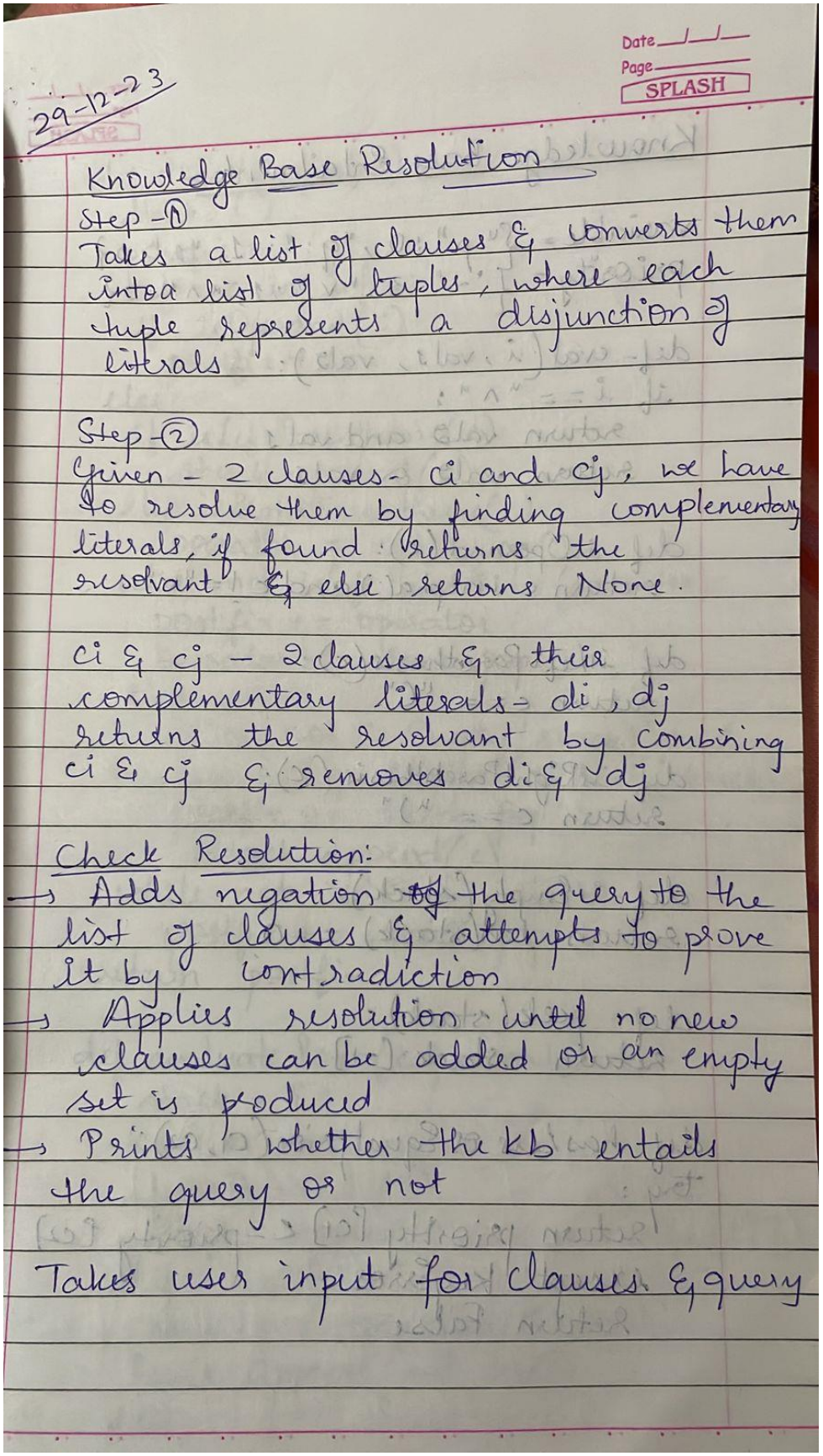
```
def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False


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 CheckEntailment():
    kb = input("Enter the knowledge base: ")
```

```
    query = input("Enter the query: ")
    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],
    ]
    postfix_kb = toPostfix(kb)
    postfix_q = toPostfix(query)
    for combination in combinations:
        eval_kb = evaluatePostfix(postfix_kb, combination)
        eval_q = evaluatePostfix(postfix_q, combination)
        print(combination, ":kb=", eval_kb, ":q=", eval_q)
        if eval_kb == True:
            if eval_q == False:
                print("Doesn't entail!! \U0001F92A")
                return False
    print("Entails")


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

OUTPUT:

```
Enter the knowledge base: (p^q)v(~pvq)
Enter the query: pvq
[True, True, True] :kb= True :q= True
[True, True, False] :kb= True :q= True
[True, False, True] :kb= False :q= True
[True, False, False] :kb= False :q= True
[False, True, True] :kb= True :q= True
[False, True, False] :kb= True :q= True
[False, False, True] :kb= True :q= False
Doesn't entail!!
```

PROGRAM-7



29-12-23

Date / /
Page
SPLASH

Knowledge Base Resolution

Step-①
Takes a list of clauses & converts them into a list of tuples, where each tuple represents a disjunction of literals

Step-②
Given - 2 clauses - ci and cj, we have to resolve them by finding complementary literals, if found returns the resolvant & else returns None.

ci & cj - 2 clauses & their complementary literals - di, dj
returns the resolvant by combining ci & cj & removes di & dj

Check Resolution:
→ Adds negation of the query to the list of clauses & attempts to prove it by contradiction
→ Applies resolution until no new clauses can be added or an empty set is produced
→ Prints whether the kb entails the query or not

Takes user input for clauses & query

29-12-23

Knowledge Base Resolution.

Kb = p ^ q
query = ~p v q } User.

[T, T, T] : Kb = T, q = T
[T, T, F] : Kb = T, q = T
:

Kb = p ^ q evaluates - T for first 2 combos & the query - ~p v q - evaluates - T for the same combination. Now, 3 rd combo, the Kb = F, while query is also - F ∴ Doesn't Entail

Resolution :-

Example :-

Kb = p v q ^ ~r ^ ~p v r

Entis query = ~q

Kb entails the query, proved by Resolution.

~(p v q.

((p v q) ^ (~r) ^ (~p v r) ^ (~q))

P v q

CODE:

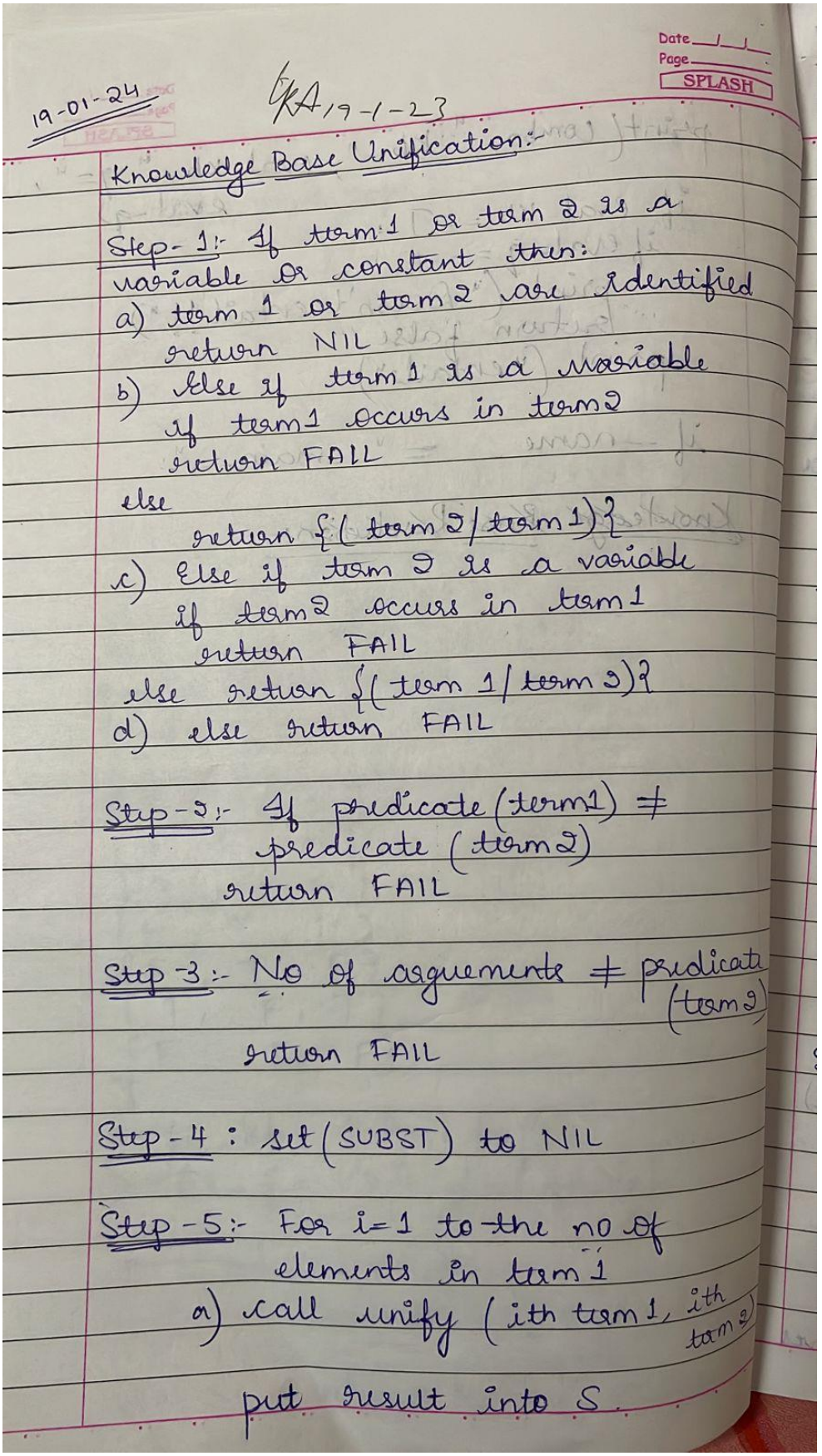
```
def disjunctify(clauses):
    disjuncts = []
    for clause in clauses:
        disjuncts.append(tuple(clause.split('v')))
    return disjuncts

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(resolvant))
        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 separated by a space: ').split()
query = input('Enter the query: ')

checkResolution(clauses, query)
```

OUTPUT:

```
Enter the clauses separated by a space: p v ~q ~r v p ~q
Enter the query: ~p
Trying to prove (p)^(v)^(~q)^(~r)^(v)^(p)^(~q)^(~p) by contradiction....
Knowledge Base entails the query, proved by resolution
```

PROGRAM-8



19-01-24

19-1-23

Knowledge Base Unification:-

Step-1:- If term 1 or term 2 is a variable or constant then:
a) term 1 or term 2 are identified return NIL
b) Else if term 1 is a variable if term1 occurs in term2 return FAIL
else
return {(term 2/term 1)}
c) Else if term 2 is a variable if term2 occurs in term 1 return FAIL
else return {(term 1/term 2)}
d) else return FAIL

Step-2:- If predicate(term1) ≠ predicate (term2)
return FAIL

Step-3:- No of arguements ≠ predicate (term2)
return FAIL

Step-4 : set(SUBST) to NIL

Step-5:- For i=1 to the no of elements in term 1
a) call unify (ith term 1, ith term 2)
put result into S.

b) S = FAIL
return FAIL

c) If S ≠ NIL
   a] Apply S to the remainder of both b1 and b2
   b] SUBST = APPEND (S, SUBJT)

Step-6: Return SUBST.

Example:- Knows (Ram x) Knows (Ram, Sham)

{x/Sham}

Knows (Ram x) → term 1
Knows (Ram, Sham) → term 2.

Any word with capital letters are constants
Here x → variable, Knows, Ram, Sham are constants.

So x takes the value of Sham.
∴ Both the terms are Same
∴ They are unified.

CODE:

```
import re


def getAttributes(expr):
    expr = expr.split("(")[1:]
    expr = "(".join(expr)
    expr = expr[:-1]
    expr = re.split("(?<!\(.),(?!.\))", expr)
    return expr

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

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

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

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

def apply(expr, subs):
    for sub in subs:
        new, old = sub  #substitution is a tuple of 2 values (new, old)
        expr = replaceAttributes(expr, old, new)
    return expr
def checkOccurs(var, expr):
    if expr.find(var) == -1:
        return False
    return True

def getFirstPart(expr):
    attr = getAttributes(expr)
    return attr[0]

def getRemainingPart(expr):
    predicate = getInitialPredicate(expr)
    attr = getAttributes(expr)
    newExpr = predicate + "(" + ",".join(attr[1:]) + ")"
    return newExpr
```

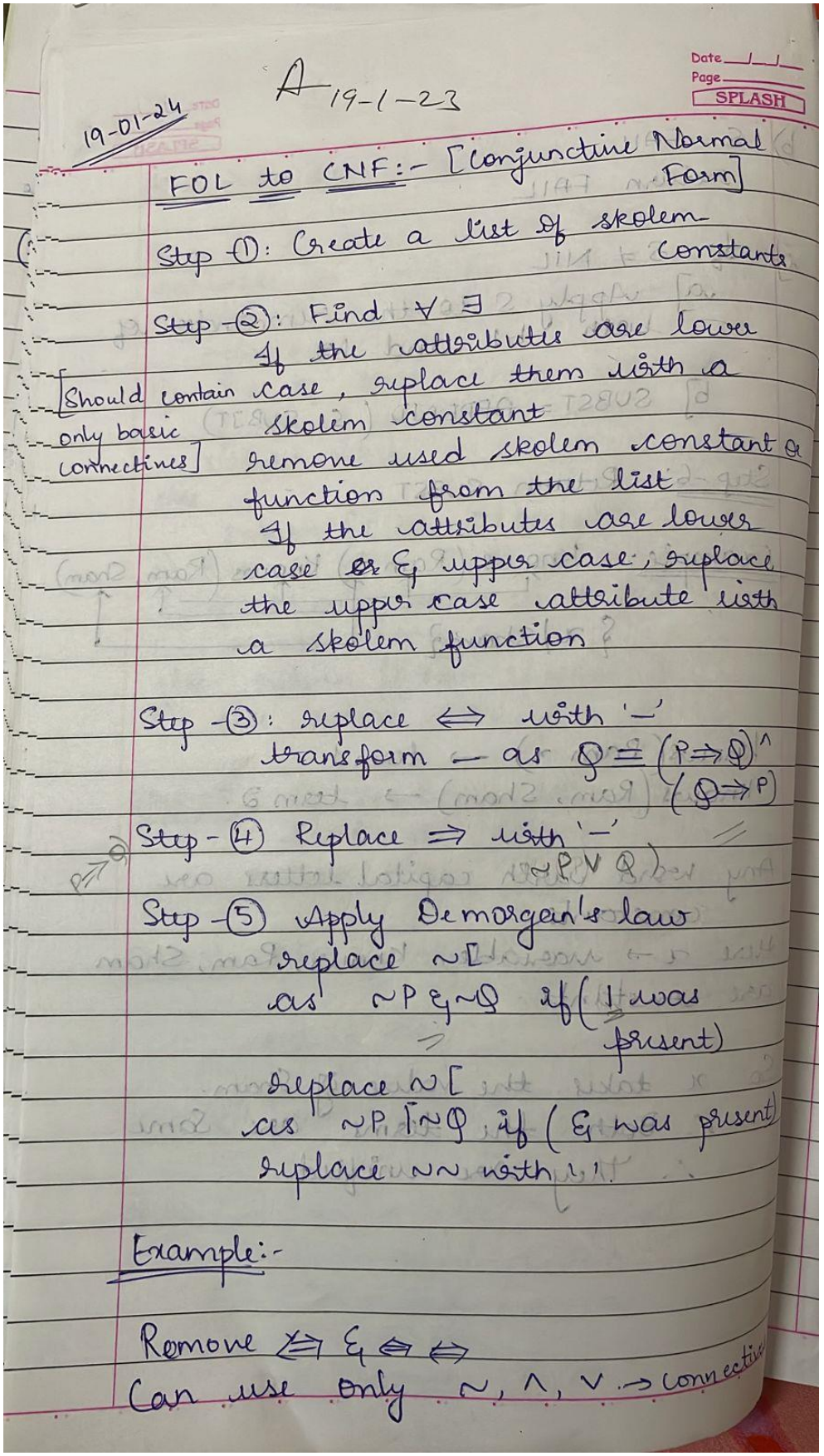
```
def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    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)
    initialSub = unify(head1, head2)
    if not initialSub:
        return False
    if attributeCount1 == 1:
        return initialSub

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

    if initialSub != []:
        tail1 = apply(tail1, initialSub)
```

```
        tail2 = apply(tail2, initialSub)

    remainingSub = unify(tail1, tail2)
    if not remainingSub:
        return False

    initialSub.extend(remainingSub)
    return initialSub

exp1 = input("Expression1: ")
exp2 = input("Expression2: ")
subs = unify(exp1, exp2)
print("Substitutions: ")
print(subs)
```

OUTPUT:

```
Expression1: parent(x, y)
Expression2: parent(john, mary)
Predicates do not match. Cannot be unified
Substitutions:
False
```

PROGRAM-9



19-01-24

A 19-1-23

FOL to CNF:- [Conjunctive Normal Form]

Step ①: Create a list of skolem constants

Step ②: Find ∀ ∃
If the attributes are lower case, replace them with a skolem constant
remove used skolem constant or function from the list
If the attributes are lower case & upper case, replace the upper case attribute with a skolem function

[Should contain only basic connectives]

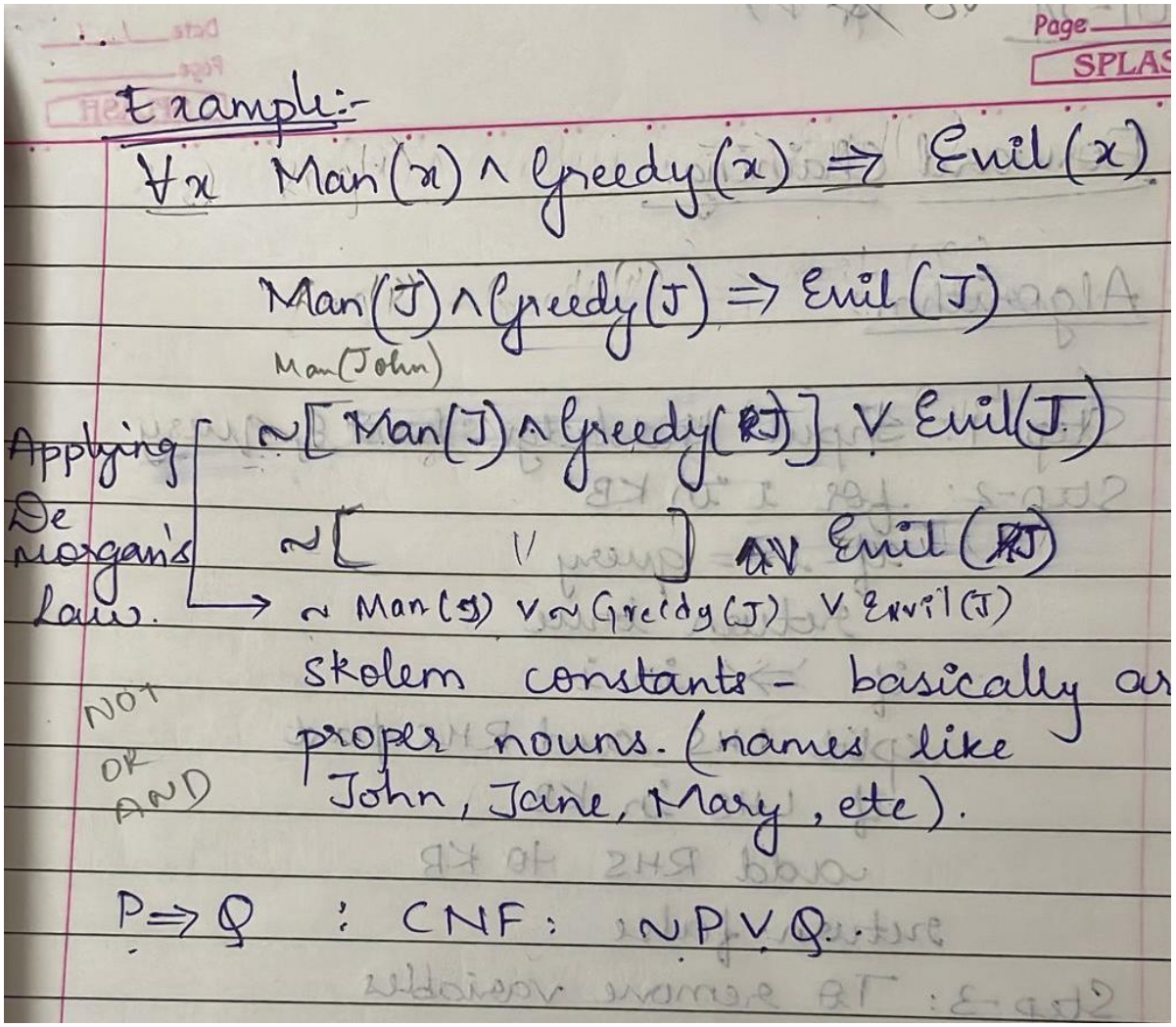
Step ③: replace ⇔ with '→'
transform → as P ⇔ Q = (P ⇒ Q) ∧ (Q → P)

Step ④ Replace ⇒ with '→'
(~P ∨ Q)

Step ⑤ Apply Demorgan's law
replace ~[
as ~P & ~Q if (⇒ was present)
replace ~[
as ~P [~Q if (& was present)
replace ~~ with ' '

Example:-

Remove ⇔ & ⇒ ⇔
Can use only ~, ∧, ∨ → connectives



CODE:

```
import re


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

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

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):
```

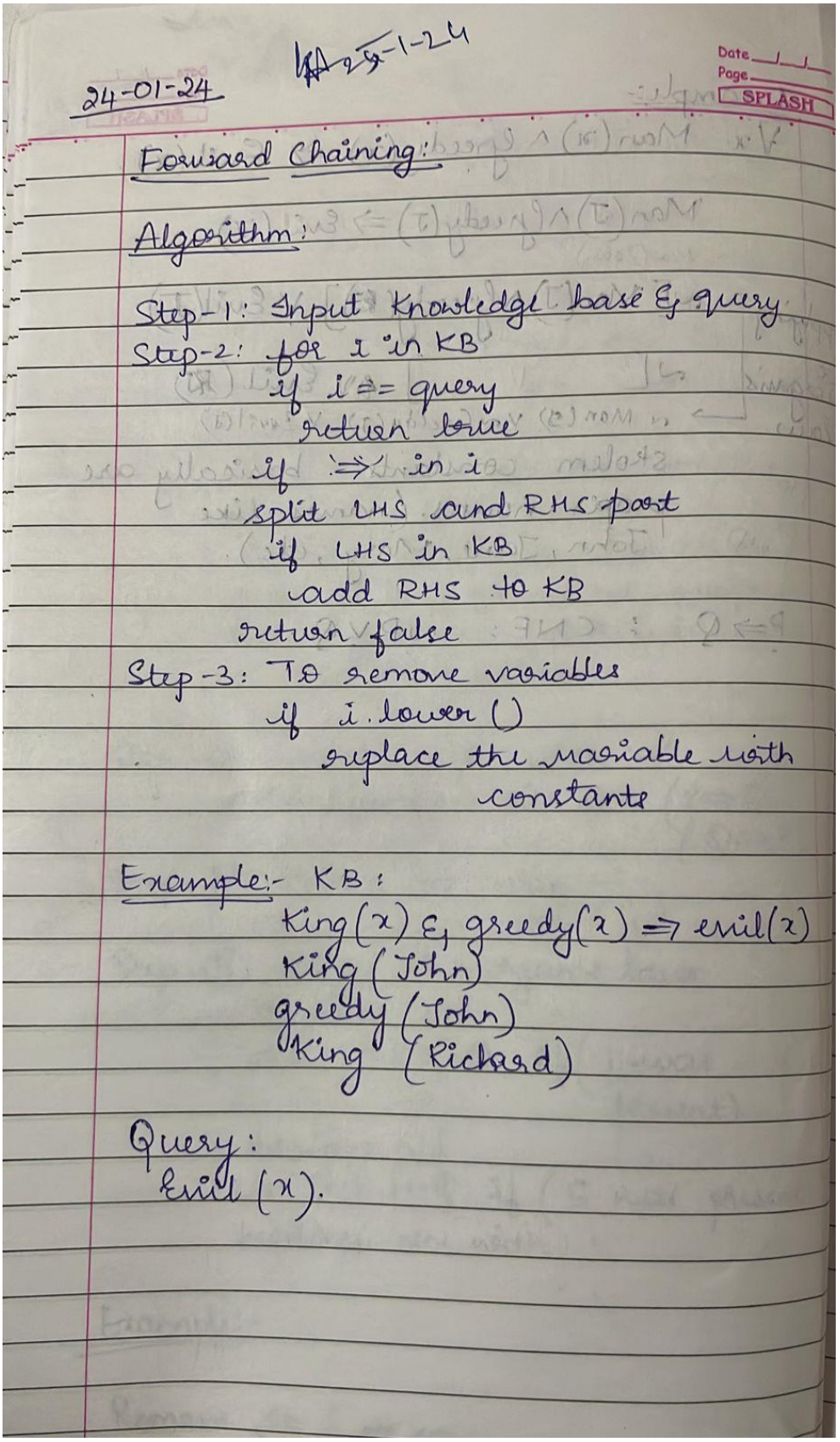
```
            string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if 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:
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU,
f'{SKOLEM_CONSTANTS.pop(0)}({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

def main():
    statement = input("Enter FOL statement: ")

    print(f"FOL converted to CNF: {Skolemization(fol_to_cnf(statement))}")

main()
```

OUTPUT:

```
Enter FOL statement: x+y_z*s
FOL converted to CNF: [~x+y|z*s]&[~z*s|x+y]
```

PROGRAM-10



24-01-24

25-1-24

Forward Chaining:

Algorithm:

Step-1: Input Knowledge base & query
Step-2: for i in KB
    if i == query
        return true
    if ⇒ in i
        split LHS and RHS part
        if LHS in KB
            add RHS to KB
        return false
Step-3: To remove variables
    if i.lower()
        replace the variable with constants

Example:- KB:
King(x) & greedy(x) ⇒ evil(x)
King(John)
greedy(John)
King(Richard)

Query:
Evil(x).

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 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}')

def main():
    kb = KB()
    n = int(input("Enter number of statements in Knowledge Base: "))

    for _ in range(n):
        kb.tell(input())

    print("Enter Query:")
    query = input()
    kb.query(query)

    kb.display()

main()
```

OUTPUT:

```
Enter number of statements in Knowledge Base: 4
Elephant(x) => Mammal(x)
Lion(Mufasa)
Mammal(x) => Animal(x)
Animal(Simba)
Enter Query:
Mammal(x)
Querying Mammal(x):
All facts:
        1. Lion(Mufasa)
        2. Animal(Simba)
```