VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

AARUSH GARG (1BM23CS004)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug-2025 to Dec-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Mohit Kumar Verma (1BM23CS198),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Prof.Swati Sridharan Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Aarush004/AI

Infosys®
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aarush Garg

for successfully completing the course

OpenAI Generative Pre-trained Transformer 3 (GPT-3) for developers

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys®
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Aarush Garg

for successfully completing

Principles of Generative AI Certification

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys®
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aarush Garg

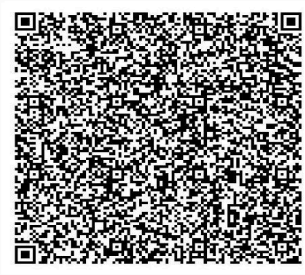
for successfully completing the course

Generative models for developers

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!



Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Infosys®
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aarush Garg

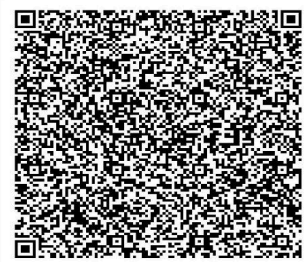
for successfully completing the course

Introduction to OpenAI GPT Models

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!



Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

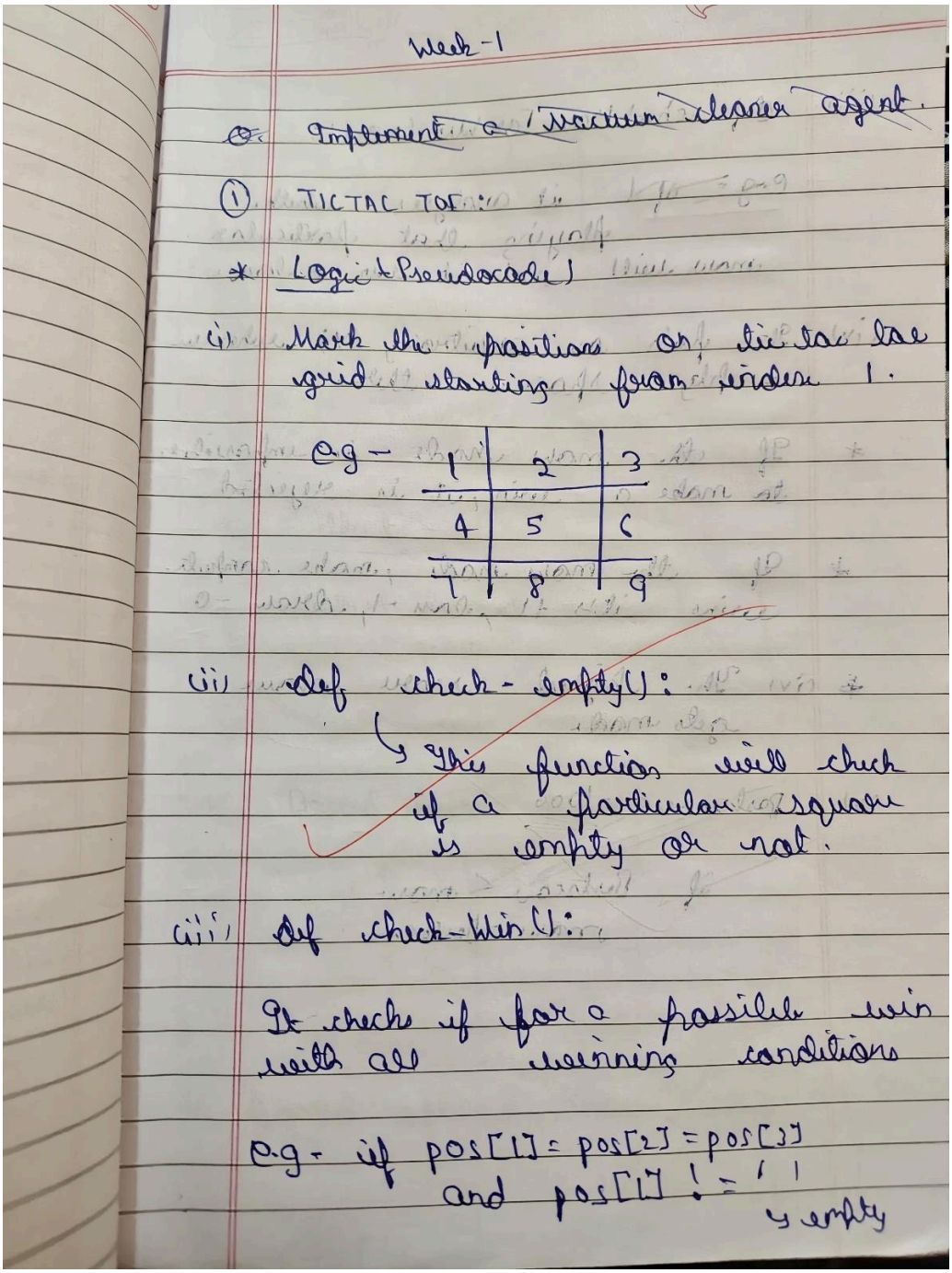
Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Week-1

Implement a vacuum cleaner agent.

(1) TICTAC TOE :

* Logic + Pseudocode)

(i) Mark the positions on tic tac toe grid starting from index 1.

e.g. –

1	2	3
4	5	6
7	8	9

(ii) def check-empty() :

↳ this function will check if a particular square is empty or not.

(iii) def check-win() :

It checks if for a possible win with all winning conditions

e.g. - if pos[1] = pos[2] = pos[3]
and pos[1] != ' '
↳ empty

(iii) def checkMoveForWin(move):

e.g:- if it analysis whether playing that particular move will make it win or loose.

(iv) It finds an arbitrary (x) where empty space is there.

* If the move made is impossible to make a win, it is rejected

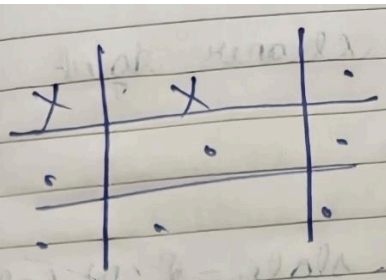
* If the move made, make computer wins it is +1, loss -1, draw -0

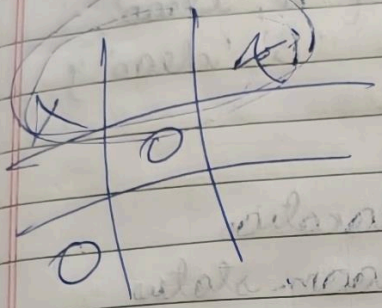
* (iv) The highest score move gets made

(v) bestScore = -1000

if bestScore < move

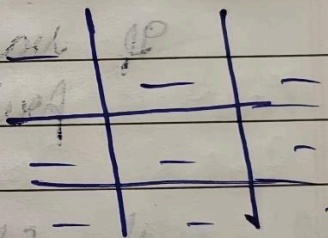
move = bestScore.



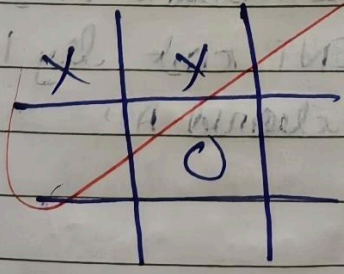


* Output -

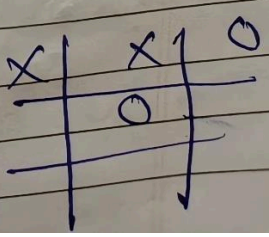
* Initial board -



* Board before AI move -



* Board after AI move :



CODE:

TIC TAC TOE

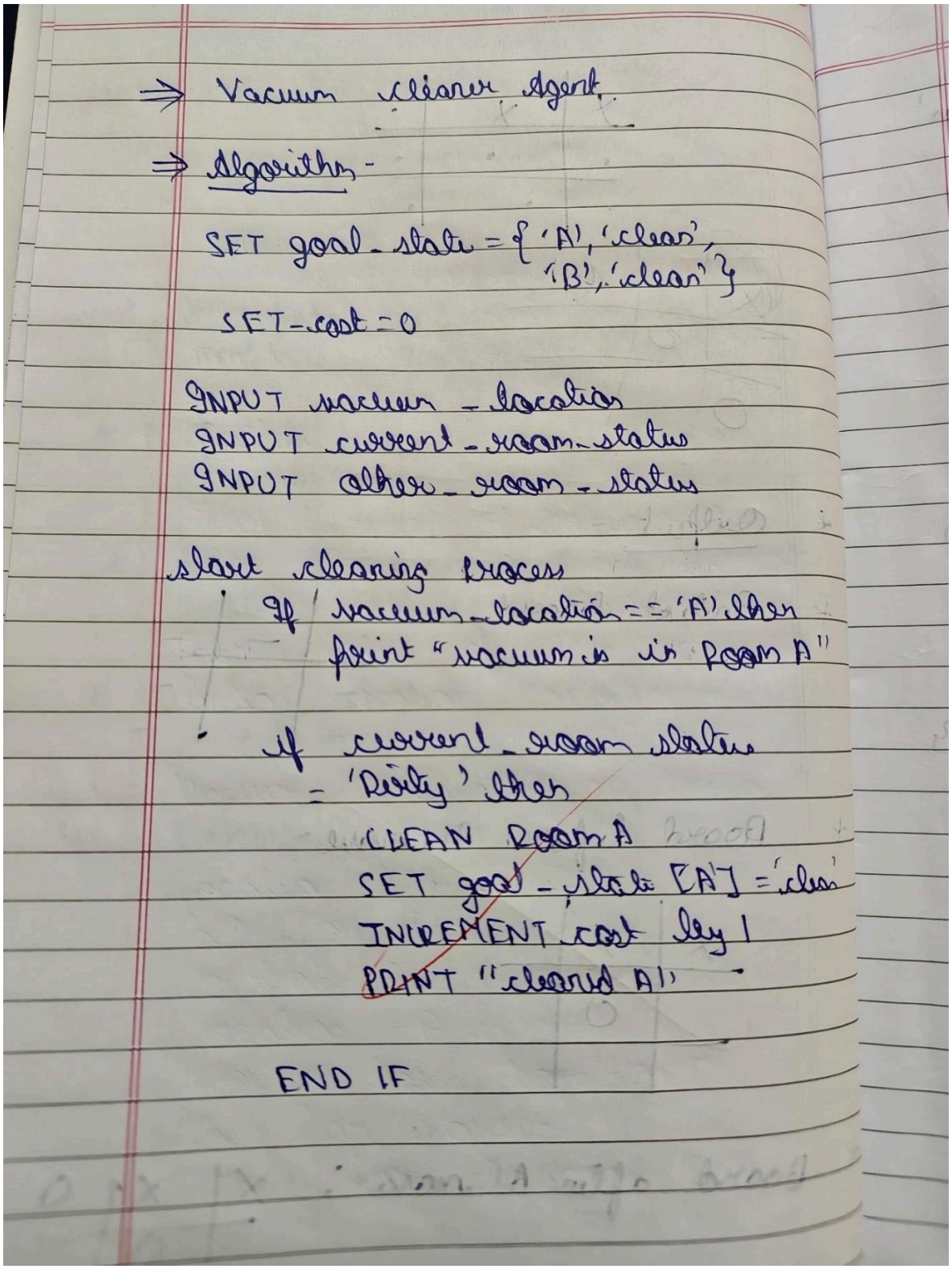
```
def print_board(board):
for row in board:
print(' | '.join(row))
print()

def is_winner(board, player):
wins = [
[board[0][0], board[0][1], board[0][2]],
[board[1][0], board[1][1], board[1][2]],
[board[2][0], board[2][1], board[2][2]],
[board[0][0], board[1][0], board[2][0]],
[board[0][1], board[1][1], board[2][1]],
[board[0][2], board[1][2], board[2][2]],
[board[0][0], board[1][1], board[2][2]],
[board[0][2], board[1][1], board[2][0]],
]
return [player, player, player] in wins

def is_board_full(board):
return all(cell != ' ' for row in board for cell in row)

def minimax(board, is_max):
if is_winner(board, 'O'):
return 1
if is_winner(board, 'X'): return
-1
if is_board_full(board):
return 0
if is_max:
best = -float('inf') for i in
range(3):
for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
val = minimax(board, False) board[i][j] =
' '
best = max(best, val) return best
else:
best = float('inf') for i
in range(3):
for j in range(3):
```

```
if board[i][j] == ' ':
board[i][j] = 'X'
val = minimax(board, True) board[i][j] = '
'
best = min(best, val) return best

def bot_move(board):
best_score = -float('inf') move
= None
for i in range(3): for j
in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O'
score      =      minimax(board,      False)
board[i][j] = ' '
if      score      >      best_score:
best_score = score move = (i,j)
if move:
board[move[0]][move[1]] = 'O' board =

[[' ']*3 for _ in range(3)]

while True:
print_board(board)
if is_winner(board, 'O'):
print("Bot wins!") break
if is_winner(board, 'X'):
print("You win!") break
if is_board_full(board):
print("Tie!")
break

try:
row = int(input("Your move row (0-2): ")) col =
int(input("Your move col (0-2): "))
if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == ' ':
board[row][col] = 'X'
else:
print("Invalid move. Try again.")
continue
except:
print("Invalid input. Try again.") continue
if is_winner(board, 'X') or is_board_full(board): continue

bot_move(board
```

OUTPUT:

```
Welcome to Tic Tac Toe!
| |
 ------
| |
 ------
| |
 ------
Enter your move (row and column: 1 1 for top-left): 1 1
X | |
 ------
| |
 ------
| |
 ------
Computer placed an O at position 3 3
X | |
 ------
| |
 ------
| | O
 ------
Enter your move (row and column: 1 1 for top-left): 1 0
Invalid position! Try again.
Enter your move (row and column: 1 1 for top-left): 1 2
X | X |
 ------
| |
 ------
| | O
 ------
Computer placed an O at position 1 3
X | X | O
 ------
| |
 ------
| | O
 ------
Enter your move (row and column: 1 1 for top-left): 1 3
That spot is already taken
```

VACUUM CLEANER AGENT:



⇒ Vacuum cleaner Agent

⇒ Algorithm -

SET goal_state = {'A', 'clean',
'B', 'clean'}

SET_cost = 0

INPUT vacuum_location
INPUT current_room_status
INPUT other_room_status

start cleaning process
If vacuum_location == 'A' then
print "vacuum is in Room A"

If current_room status
= 'Dirty' then
CLEAN Room A
SET goal_state['A'] = 'clean'
INCREMENT cost by 1
PRINT "cleaned A"

END IF

```
        SET goal = state[A] = 'clean'
        INCREMENT cost by 1
        PRINT "cleaned A"
    ELSE
        PRINT "Room A is clean"
    END IF
END IF
```

* Output

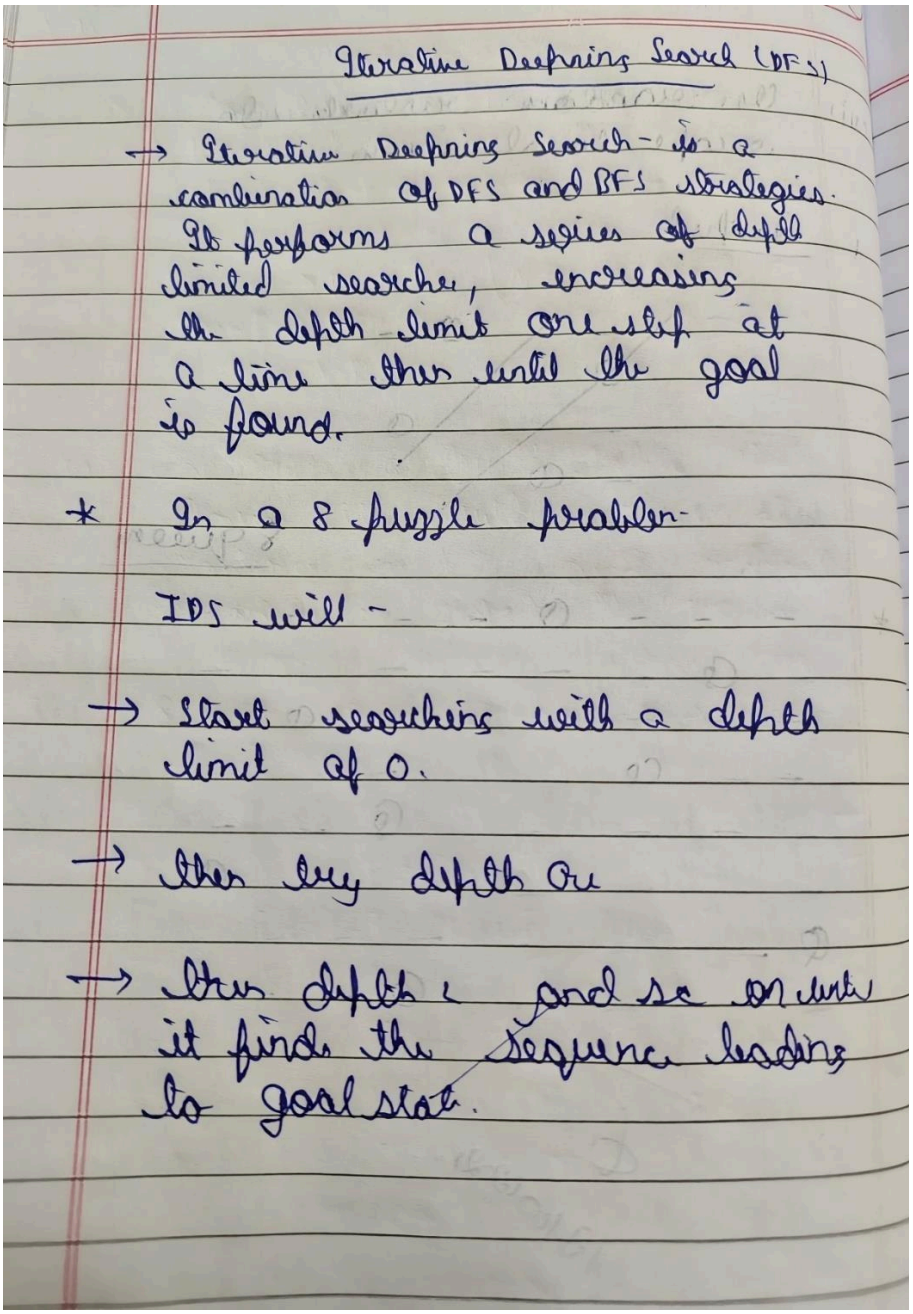
Enter location of vacuum (A/B): A
Enter status of A (0 for clean / 1 for dirty) : 1
Enter status of other room : 1
Initial location condition : {'A', '0', 'B', '0'}
Vacuum is placed in location B

Location A is dirty
Cost for cleaning A: 1
Location A has been cleaned
Location B is dirty
Moving to location B
Cost for moving : 1
Cost for cleaning B: 1

```
return f"Time={self.time}, Perf={self.performance}, State={state}"
def simple_reflex_agent(percept):
loc, status = percept
if status == "dirty":
return SUCK
return RIGHT if loc == 0 else LEFT

class ModelBasedReflexAgent:
def _init_(self, n_rooms):
self.n = n_rooms
self.model = {i: "unknown" for i in range(n_rooms)}

def program(self, percept):
loc, status = percept
self.model[loc] = status if
status == "dirty":
return SUCK
dirty_rooms = [i for i, s in self.model.items() if s == "dirty"] if
dirty_rooms:
distances = [(abs(i - loc), i) for i in dirty_rooms]
_, target = min(distances) if target
< loc:
return LEFT elif target >
loc:
return RIGHT else:
return SUCK
unknown_rooms = [i for i, s in self.model.items() if s == "unknown"] if
unknown_rooms:
distances = [(abs(i - loc), i) for i in unknown_rooms]
_, target = min(distances) if target
< loc:
return LEFT elif target >
loc:
return RIGHT else:
return NO_OP return
NO_OP

def run_agent(env, agent_program, max_steps=100, verbose=True):
program = agent_program.program if hasattr(agent_program, "program") else agent_program
trace = []
for step in range(max_steps):
percept = env.percept() action =
program(percept)
trace.append((step, percept, action, env.performance))
env.execute(action)
```

```
if verbose:
print(f"Step {step}: Percept={percept} -> Action={action}; Perf(before)={trace[-1][3]}") print(env)
if env.is_all_clean() and action == NO_OP: if
verbose:
print("All clean and agent chose NO_OP -> Halting.") break
return trace, env.performance

if __name__ == "__main__":
print("=== Demo: 2-room world ===")
env1 = Environment(n_rooms=2, dirt_probability=0.5, seed=42)
print("Initial:", env1)
print("\nSimple Reflex Agent run:")
env_copy = Environment(n_rooms=2, dirt_probability=0.5, seed=42)
trace, perf = run_agent(env_copy, simple_reflex_agent, max_steps=20, verbose=True)
print("Final performance:", perf)
print("\nModel-Based Reflex Agent run:")
env_copy2 = Environment(n_rooms=2, dirt_probability=0.5, seed=42)
model_agent = ModelBasedReflexAgent(n_rooms=2)
trace2, perf2 = run_agent(env_copy2, model_agent, max_steps=20, verbose=True) print("Final
performance:", perf2)
print("\n=== Demo: 5-room world random dirt ===")
env5 = Environment(n_rooms=5, dirt_probability=0.4, seed=123)
print("Initial:", env5)
model_agent5 = ModelBasedReflexAgent(n_rooms=5)
_, perf5 = run_agent(env5, model_agent5, max_steps=200, verbose=False)
print("Final perf (silent run):", perf5)
```

2. Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm



Iterative Deepning Search (IDFS)

→ Iterative Deepning Search - is a combination of DFS and BFS strategies. It performs a series of depth limited searches, increasing the depth limit one step at a time then until the goal is found.

* In a 8 puzzle problem-

IDS will -

→ Start searching with a depth limit of 0.

→ Then try depth one

→ then depth 2 and so on until it finds the sequence leading to goal state.

e.g. Initial State

1	2	3
0	4	6
7	5	8

Goal state

1	2	3
4	5	6
7	8	0

Depth = 0 ⇒

1	2	3
0	4	6
7	5	8

Depth = 1 ⇒

1	2	3	1	2	3	0	2	3
4	0	6	7	4	6	1	4	6
7	5	8	0	5	8	7	5	8

Depth = 2 ⇒

1	0	3	1	2	3	1	2	3	1	2	3	2	0	3
4	2	6	4	5	6	4	6	0	7	4	6	1	4	6
7	5	8	7	0	8	7	5	8	5	0	8	7	5	8

Depth = 3

0	1	3	1	3	0	1	2	3	1	2	8	1	2	3
4	2	6	4	2	6	4	5	6	4	5	6	4	6	7
7	5	8	7	8	0	7	8	0	7	0	8	5	7	8

1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	0	7	8	0	7	8	0	7	8	0	7	8	0

```
from collections import deque

GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

MOVES = {
'up': -3,
'down': 3,
'left': -1,
'right': 1
}

def is_valid_move(blank_idx, move):
if move == 'left' and blank_idx % 3 == 0: return
False
if move == 'right' and blank_idx % 3 == 2: return
False
if move == 'up' and blank_idx < 3: return
False
if move == 'down' and blank_idx > 5:
return False
return True

def get_successors(state):
successors = []
blank_idx = state.index(0)

for move in MOVES:
if is_valid_move(blank_idx, move): swap_idx =
blank_idx + MOVES[move] new_state =
list(state)
new_state[blank_idx], new_state[swap_idx] = new_state[swap_idx], new_state[blank_idx]
successors.append(tuple(new_state))
return successors

def DLS(state, goal, limit, path, visited, last_state_holder):
last_state_holder[0] = state

if state == goal: return
path
if limit == 0:
return None

visited.add(state)

for successor in get_successors(state): if
successor not in visited:
```

```
result = DLS(successor, goal, limit - 1, path + [successor], visited, last_state_holder)
if result is not None: return
result

visited.remove(state)
return None

def IDDFS(start, goal):
depth = 0
iteration = 1
while True:
visited = set() last_state_holder =
[start]
path = DLS(start, goal, depth, [start], visited, last_state_holder)

print(f"Iteration {iteration} completed at Depth limit = {depth}") print("Last
visited puzzle state in this iteration:") print_puzzle(last_state_holder[0])

if path is not None:
return path
depth += 1
iteration += 1

def print_puzzle(state):
for i in range(0, 9, 3):
print(state[i:i+3]) print()

if _name____== "_main_":
start_state = (1, 2, 3,
4, 0, 6,
7, 5, 8)

print("Starting state:")
print_puzzle(start_state)

solution_path = IDDFS(start_state, GOAL_STATE)

print(f"Solution found in {len(solution_path) - 1} moves:") for
step in solution_path:
print_puzzle(step)
```

OUTPUT:

```
Starting state:
(1, 2, 3)
(4, 0, 6)
```

(7, 5, 8)

Iteration 1 completed at Depth limit = 0
Last visited puzzle state in this iteration:
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)

Iteration 2 completed at Depth limit = 1
Last visited puzzle state in this iteration:
(1, 2, 3)
(4, 6, 0)
(7, 5, 8)

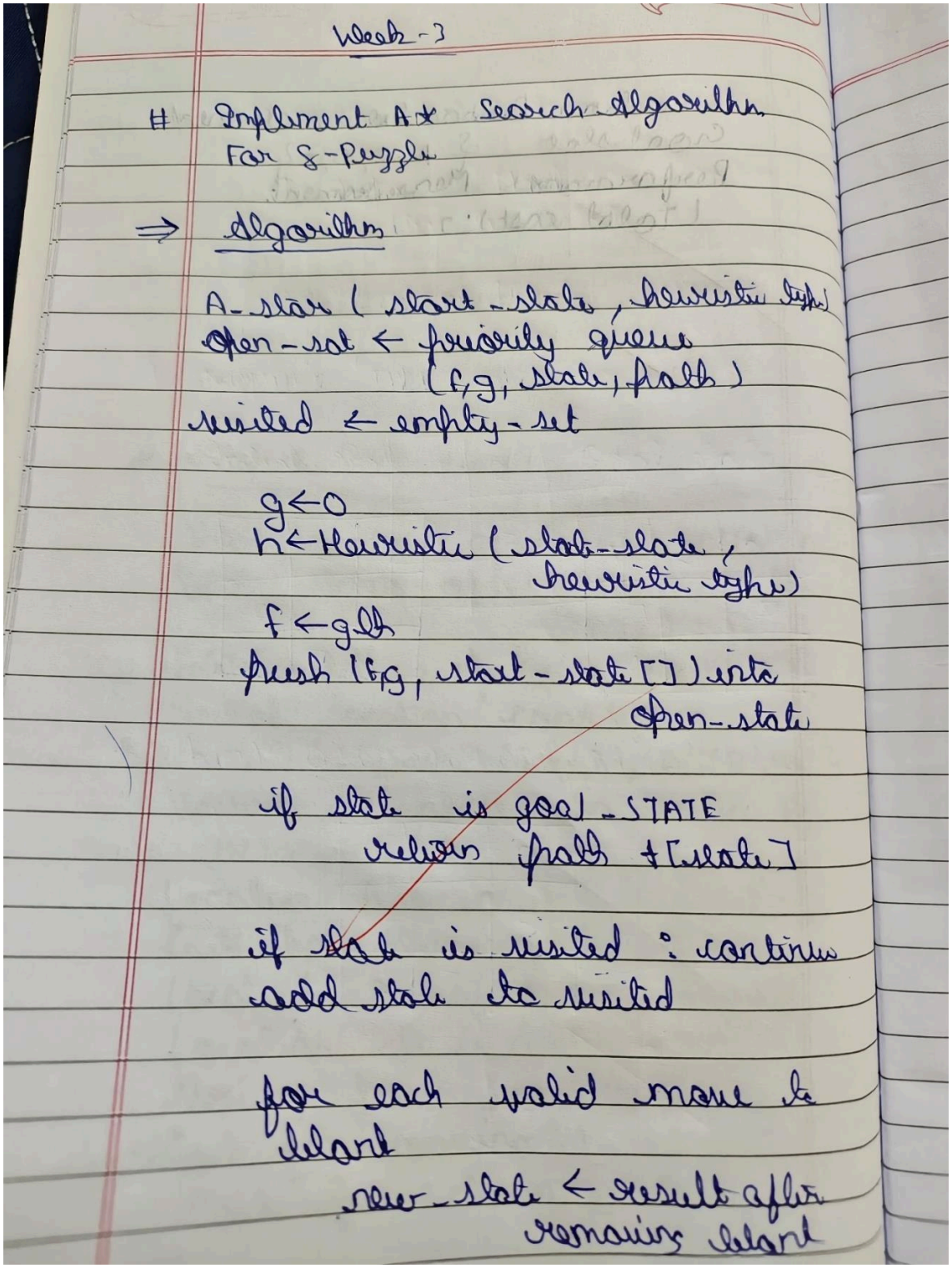
Iteration 3 completed at Depth limit = 2
Last visited puzzle state in this iteration:
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

Solution found in 2 moves:
(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)

3. Implement A* search algorithm



Week-3

Implement A* Search Algorithm For 8-Puzzle

⇒ Algorithm

A-star (start-state, heuristic type)
open-set ← priority queue
(f, g, state, path)
visited ← empty-set

g ← 0
h ← Heuristic (state-state, heuristic type)
f ← g+h
push (f, g, start-state[],) into open-state

if state is goal-STATE
return path + [state]

if state is visited : continue
add state to visited

for each valid move in legal
new-state ← result after removing legal

```
if new_state not in visited
    new_g ← g+1
    new_h ← Heuristic (new_state,
            heuristic, depth)
    new_f ← new_g + new_h
    push (new_f, new_g, new_
          state, path)
    into open set

return "No solution"

Heuristic 1 (state, depth)
    sum ← 0
    for each tile ≠ 0
        sum += abs (current goal -
                    goal row)
             + abs (current col - goal col)

    return sum

Heuristic 2 (state)
    count ← number of tiles ≠ goal
                (excluding 0)
    return count
```

Heuristic (state_type)

if type == "Manhattan" return Heuristic (state)

else : return Heuristic 2 (state)

* NUMBER TILES OUT OF PLACE:

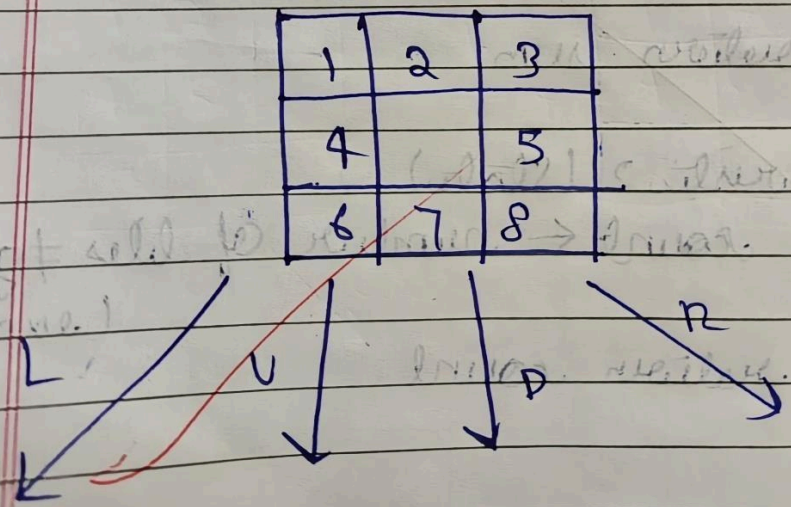
Initial state

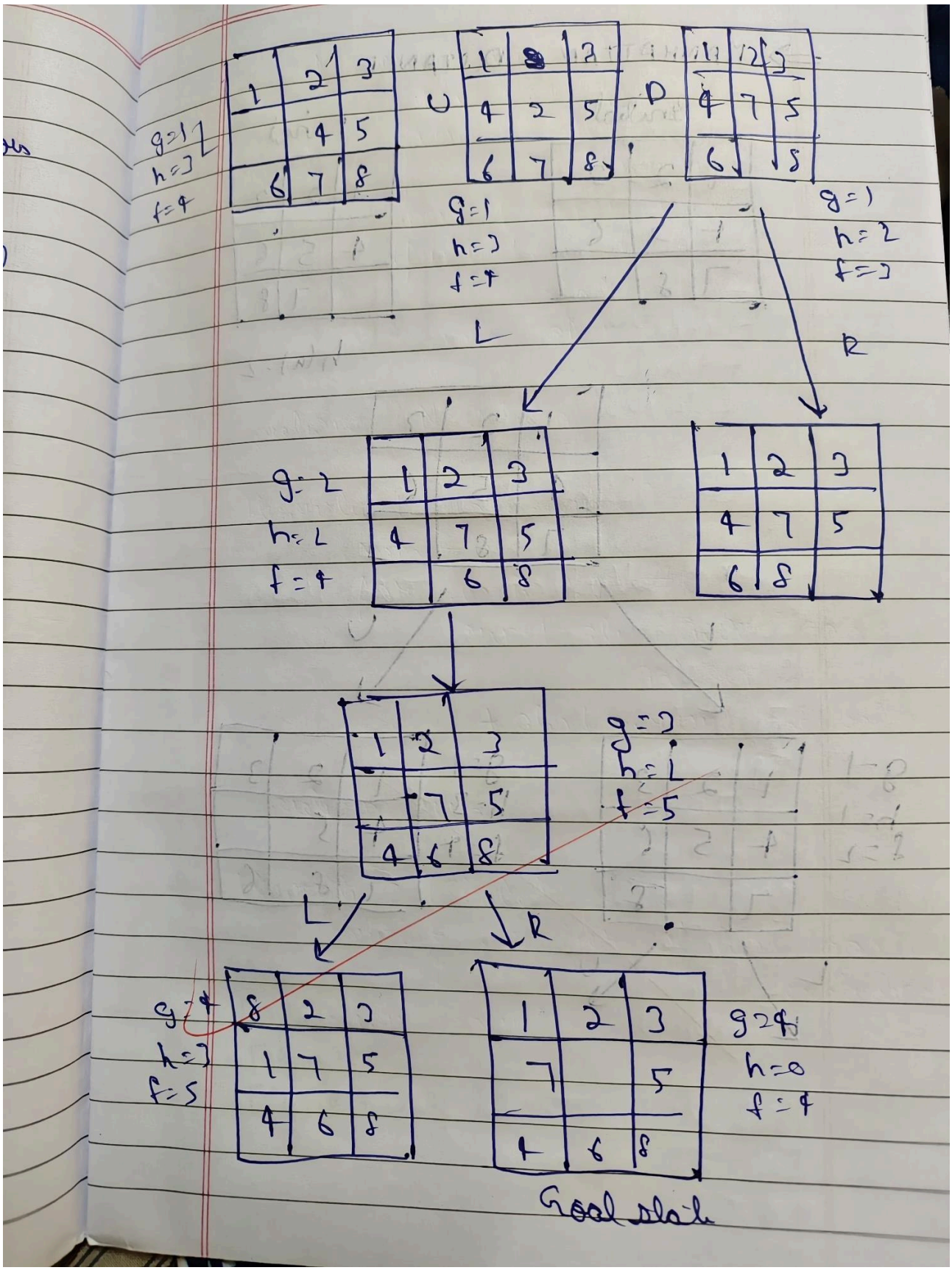
1	2	3
4		5
6	7	8

Goal state

1	2	3
7		5
4	6	8

* h(n) = 0+0+0+0+0+1+1+1 = 3




g=1
h=3
f=4
U
g=1
h=3
f=4
D
g=1
h=2
f=3
L
R
g=2
h=2
f=4
g=2
h=2
f=5
L
R
g=4
h=3
f=5
g=4
h=0
f=4
Goal state

⇒ MANHATTAN DISTANCE

Initial

1	2	3
4	5	6
7	8	

Final

1	2	3
4	5	6
	7	8

h(a) = 2

1	2	3
4	5	6
7	8	

L U

g = 1
h = 1
f = 2

1	2	3
4	5	6
7		8

g = 1
h = 3
f = 4

1	2	3
4	5	
7	8	6

L U

L

1	2	3
4	5	6
	7	8

g=2
h=0
f=2

U

1	2	3
4		6
7	5	8

Goal State

* Output , Initial State = [[1,2,3]
[4,0,6]
[7,5,8]]

Goal State = [[1,2,3]
[4,5,6]
[7,8,0]]

* Using Manhattan Distance Heuristic:

Step 0:
[1,2,3]
[4,0,6]
[7,5,8]

```
import heapq

start = ((1,2,3),(4,5,6),(7,0,8))
goal = ((1,2,3),(4,5,6),(7,8,0))

def heuristic(state):
dist = 0
for i in range(3): for j
in range(3):
val = state[i][j] if val !=
0:
for x in range(3): for y in
range(3):
if goal[x][y] == val:
dist += abs(i - x) + abs(j - y) break
return dist

def neighbors(state):
board = [list(row) for row in state] x
= y = -1
for i in range(3): for j
in range(3):
if board[i][j] == 0: x, y =
i, j
break if x != -1:
break moves
= []
for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
nx, ny = x + dx, y + dy
if 0 <= nx < 3 and 0 <= ny < 3: new_board =
[row[:] for row in board]
new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
moves.append(tuple(tuple(row) for row in new_board))
return moves

def a_star(start):
open_set = []
heapq.heappush(open_set, (heuristic(start), 0, start, [])) visited
= set()
while open_set:
f, g, state, path = heapq.heappop(open_set) if
state == goal:
return path + [state] if state
in visited:
continue
```

```
visited.add(state)
for n in neighbors(state): if n
not in visited:
heapq.heappush(open_set, (g+1+heuristic(n), g+1, n, path+[state])) return
None

result = a_star(start)
if result is None:
print("No solution found")
else:
for step in result: for
row in step:
print(row) print()
```

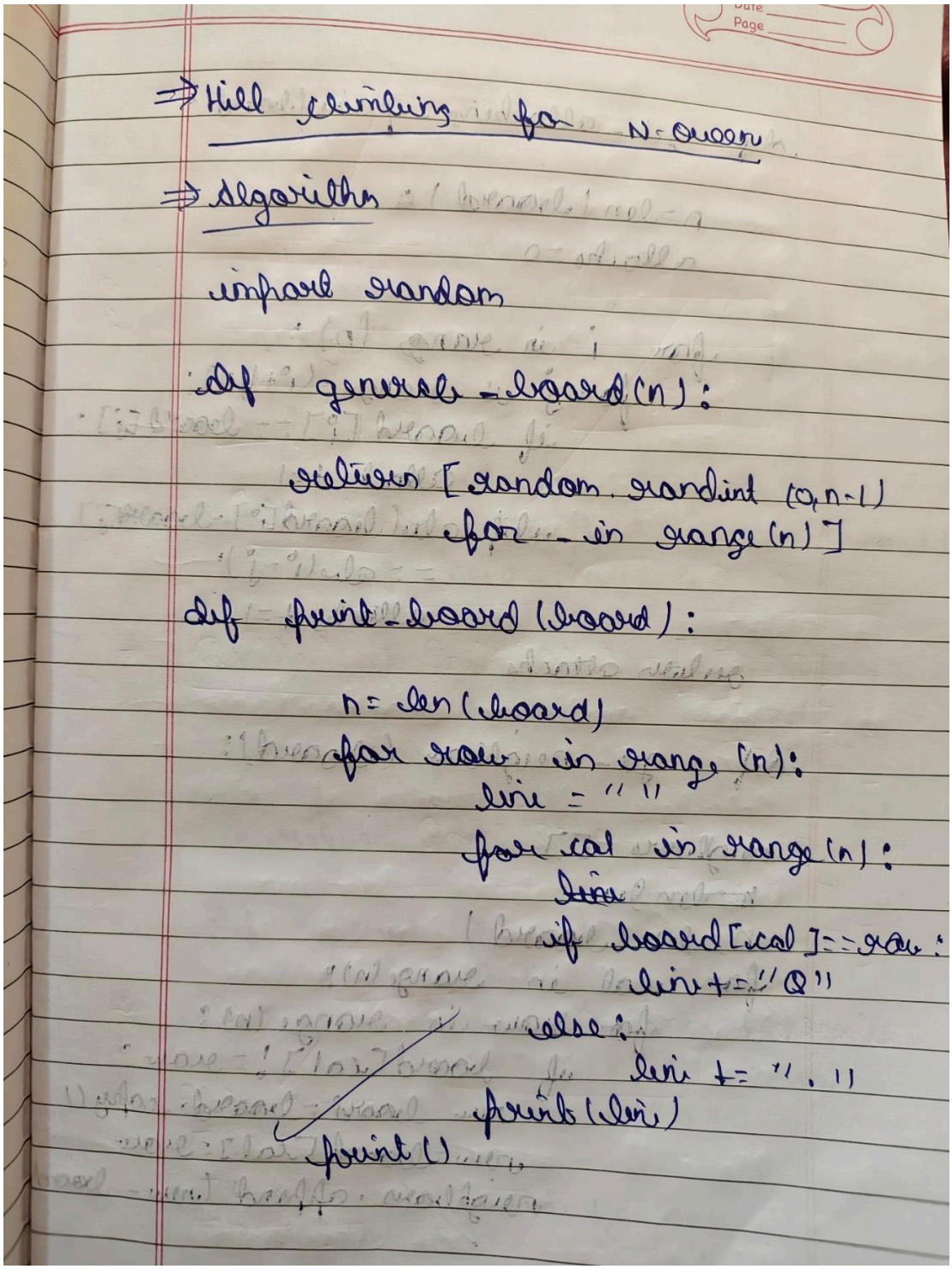
OUPUT:

Solution found in 2 moves:

(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

4. Implement Hill Climbing search algorithm to solve N-Queens problem



⇒ Hill climbing for N-queen

⇒ Algorithm

```
import random

def generate_board(n):
    return [random.randint(0,n-1)
            for _ in range(n)]

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += "Q"
            else:
                line += "."
        print(line)
    print()
```

```
def get_attacking_pairs(board):

    n = len(board)
    attacks = 0

    for i in range(n):
        for j in range(i+1, n):
            if board[i] == board[j]:
                attacks += 1
            elif abs(board[i] - board[j])
                == abs(i - j):
                attacks += 1

    return attacks

def get_neighbors(board):

    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if board[col] != row:
                new_board = board.copy()
                new_board[col] = row
                neighbors.append(new_board
```

```
def hill_climb(n, max_restarts=10):
    for restart in range(max_
                         restarts):
        board = generate_board(n)
        conflicts = calculate_conflicts(board)

        while True:
            board, new_conflicts =
                get_best_neighbor(board)
            if new_conflicts == 0:
                return board
            if new_conflicts >= conflicts:
                break
            conflicts = new_conflicts

    return None

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
```

```
if __name__ == "__main__":
    N=8
    solution = hill_climb(N)
    if solution:
        print("Solution found:")
        print_board(solution)
    else:
        print("No solution found:")
```

* Steps

(i) Start with a random board

(ii) Calculate the number of conflicts.

(iii) For each column, try moving queen to every possible row and find the position that reduces conflicts the most.

(iv) Move the queen there.

(v) Repeat until no improvement can be found.

```
import random

def generate_board(n):
return [random.randint(0, n - 1) for _ in range(n)]

def calculate_conflicts(board):
conflicts = 0
n = len(board) for i
in range(n):
for j in range(i + 1, n):
if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j): conflicts += 1
return conflicts

def get_best_neighbor(board):
n = len(board)
current_conflicts = calculate_conflicts(board)
best_board = list(board)
best_conflicts = current_conflicts

for col in range(n):
original_row = board[col] for
row in range(n):
if row == original_row: continue
new_board = list(board) new_board[col]
= row
new_conflicts = calculate_conflicts(new_board) if
new_conflicts < best_conflicts:
best_conflicts = new_conflicts best_board
= new_board
return best_board, best_conflicts def

hill_climb(n, max_restarts=1000):

    for _ in range(max_restarts):
        board = generate_board(n)
conflicts = calculate_conflicts(board) while
True:
next_board, next_conflicts = get_best_neighbor(board) if
next_conflicts == 0:
return next_board
if next_conflicts >= conflicts: break
board, conflicts = next_board, next_conflicts
```

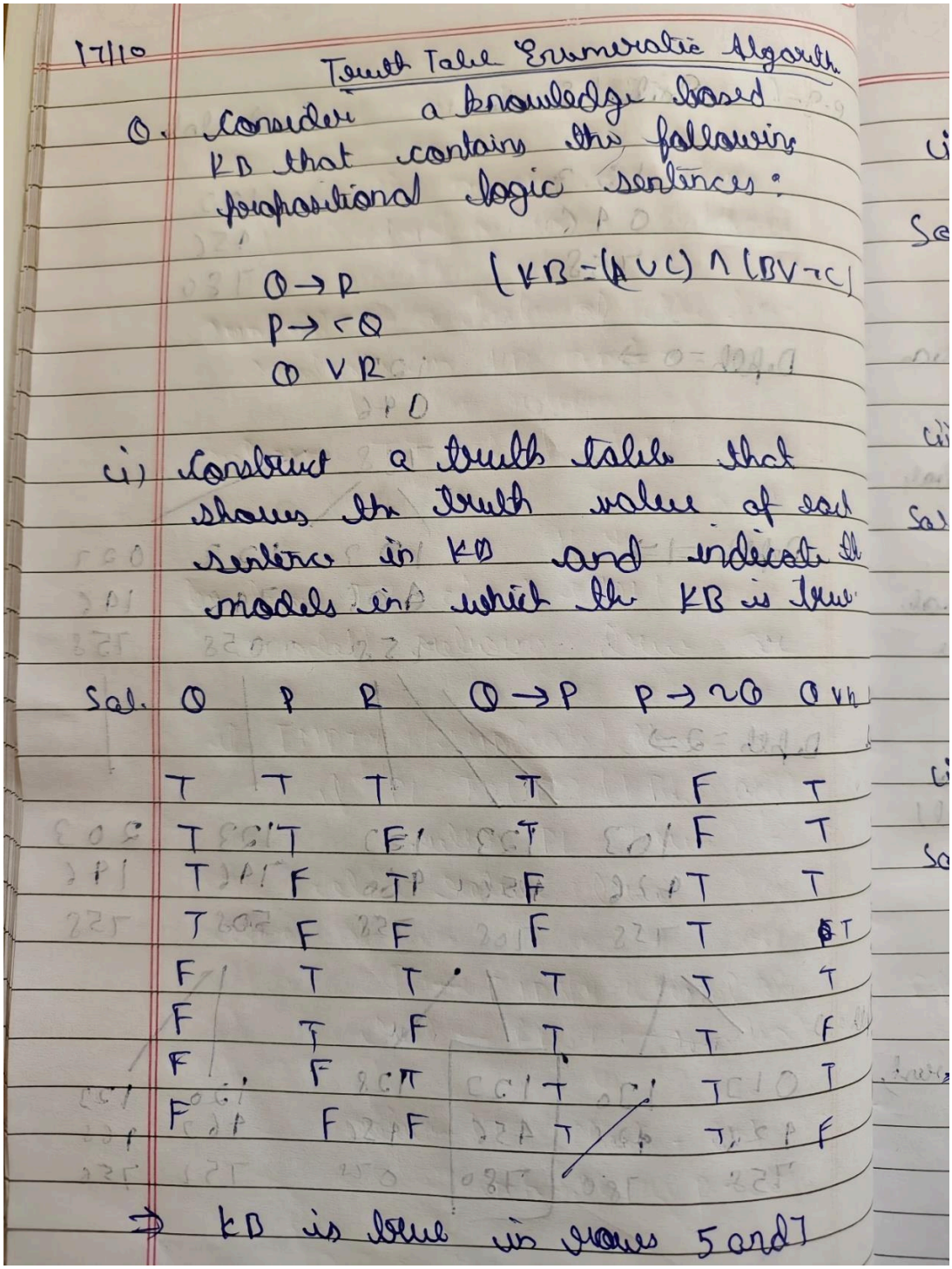
```
return None

def print_board(board):
n = len(board)
for row in range(n): line
= ""
for col in range(n):
line += "Q " if board[col] == row else ". " print(line)

if _name____== "_main_":
N = 8
solution = hill_climb(N) if
solution:
print_board(solution) else:
print("No solution found.")
```

OUTPUT:

Solution found for 8-Queens:

```
. . . Q . . . .
.         Q . .
.             Q
. Q . . . . . .
.           Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
```

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



17/10

Truth Table Enumeration Algorithm

Q. Consider a knowledge based KB that contains the following propositional logic sentences:

$Q \rightarrow P$ (KB = $(A \vee C) \wedge (B \vee \neg C)$)

$P \rightarrow \neg Q$

$Q \vee R$

i) Construct a truth table that shows the truth value of each sentence in KB and indicate the models in which the KB is true.

Sol. Q	P	R	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$
T	T	T	T	F	T
T	T	F	T	F	T
T	F	T	F	T	T
T	F	F	F	T	T
F	T	T	T	T	T
F	T	F	T	T	F
F	F	T	T	T	T
F	F	F	T	T	F

⇒ KB is true in rows 5 and 7

(ii) Does KB entail R?

Sol. KB is true in row is 5 (F, T, T) and 7 (F, F, T). In both of them P is true. Therefore, KB entails R.

(iii) Does KB entail $R \rightarrow P$?

Sol Row 5 : R=T, P=T $\Rightarrow$ $R \rightarrow P$ is T
Row 7 : R=T, P=F $\Rightarrow$ $R \rightarrow P$ is F

No, KB doesn't entail $R \rightarrow P$ because in row 7, $R \rightarrow P$ is false.

(iv) Does KB entail $Q \rightarrow R$

Sol. In row 5: Q=F, R=T $\Rightarrow$ $Q \rightarrow R$ is T
row 7 : Q=F, R=T $\Rightarrow$ $Q \rightarrow R$ is T

Therefore, KB entails $Q \rightarrow R$

* Algorithm

function TT-ENTAILS? (KB, α) returns true or false

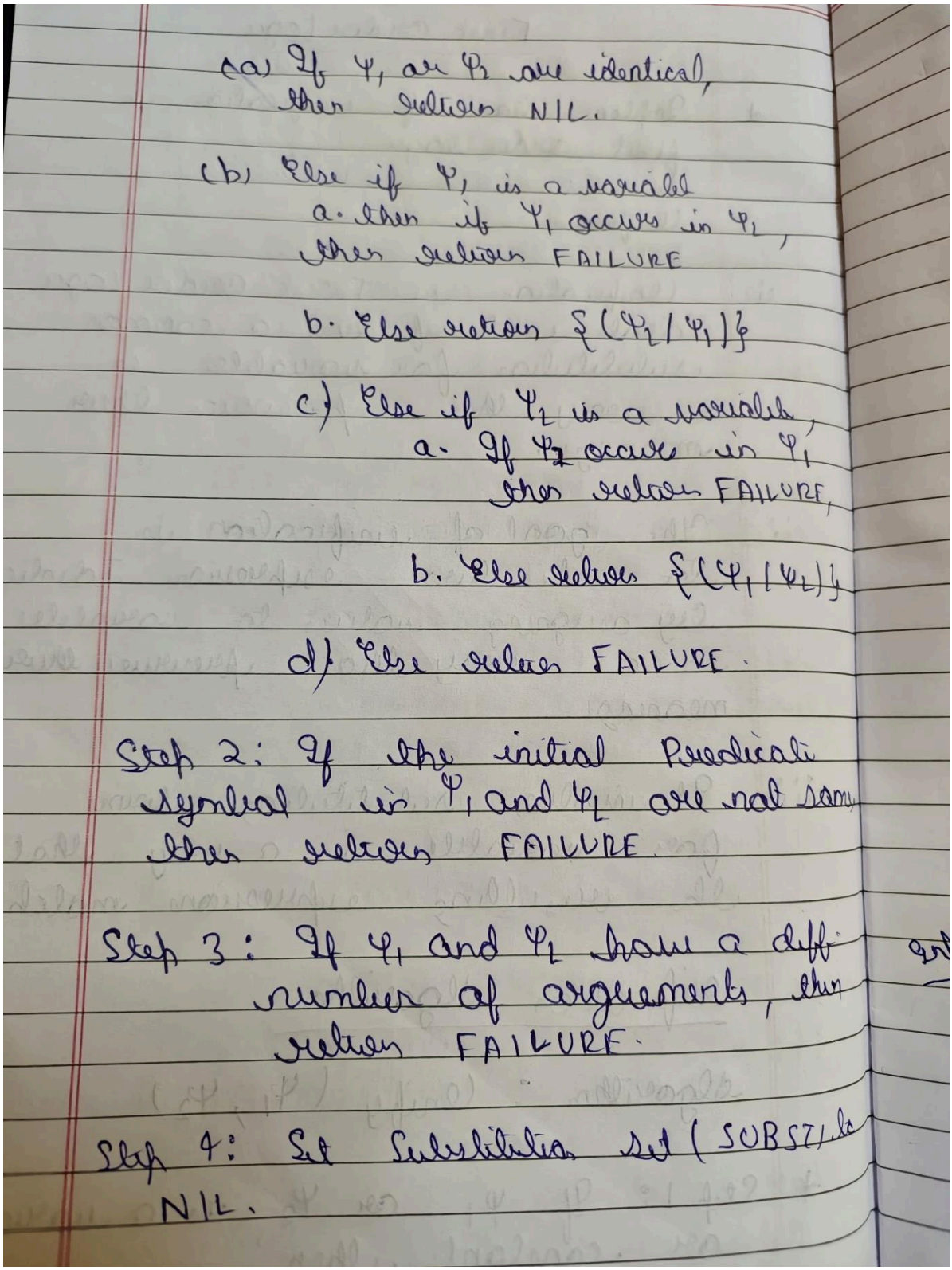
```
from itertools import product

def tt_entails(kb, alpha):
# Get all unique symbols in KB and alpha
symbols = list(set(kb_symbols(kb) | kb_symbols(alpha))) return
tt_check_all(kb, alpha, symbols, {})

def tt_check_all(kb, alpha, symbols, model):
if not symbols:
if pl_true(kb, model):
return pl_true(alpha, model) else:
return True
else:
P = symbols[0] rest =
symbols[1:]
model_true = model.copy()
model_true[P] = True model_false =
model.copy() model_false[P] = False
return (tt_check_all(kb, alpha, rest, model_true) and
tt_check_all(kb, alpha, rest, model_false))

def pl_true(expr, model):
# Evaluate the expression given the model of symbol truth values #
expr can be:
# - symbol string (e.g. 'P')
# - tuple ('AND', expr1, expr2) #
- tuple ('OR', expr1, expr2)
# - tuple ('NOT', expr)
# - tuple ('IMPLIES', expr1, expr2)

if isinstance(expr, str): #
Base case: symbol
return model.get(expr, False) op =
expr[0]
if op == 'AND':
return pl_true(expr[1], model) and pl_true(expr[2], model) elif op
== 'OR':
return pl_true(expr[1], model) or pl_true(expr[2], model) elif
op == 'NOT':
return not pl_true(expr[1], model) elif op
== 'IMPLIES':
return (not pl_true(expr[1], model)) or pl_true(expr[2], model) else:
```

```
raise ValueError(f"Unknown operator {op}")

def kb_symbols(expr):
# Recursively collect all symbols in the expression if
isinstance(expr, str):
return {expr} op =
expr[0]
if op in ('AND', 'OR', 'IMPLIES'):
return kb_symbols(expr[1]) | kb_symbols(expr[2]) elif
op == 'NOT':
return kb_symbols(expr[1]) else:
return set() #

Example usage:

# KB = (A OR C) AND (B OR NOT C)
KB = ('AND',
('OR', 'A', 'C'),
('OR', 'B', ('NOT', 'C'))
)

# Query α = A OR B
alpha = ('OR', 'A', 'B')

result = tt_entails(KB, alpha)
print("Does KB entail alpha?", result)
```

OUTPUT:

```
Does KB entail alpha? True
```

6. Implement unification in first order logic.



(a) If Ψ_1 or Ψ_2 are identical, then return NIL.

(b) Else if Ψ_1 is a variable
a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
b. Else return $\{(\Psi_2 / \Psi_1)\}$

c) Else if Ψ_2 is a variable,
a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
b. Else return $\{(\Psi_1 / \Psi_2)\}$

d) Else return FAILURE.

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3: If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step 4: Set Substitution set (SUBST) to NIL.

Step 5: For i=1 to number of elements in Ψ_1.

(a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

(b) If S= failure then returns FAILURE.

(c) If S ≠ NIL then do,
a. Apply S to the remainder of both L_1 and L_2

b. SUBST = APPEND(S, SUBST)

Step 6 : Return SUBST

* Example

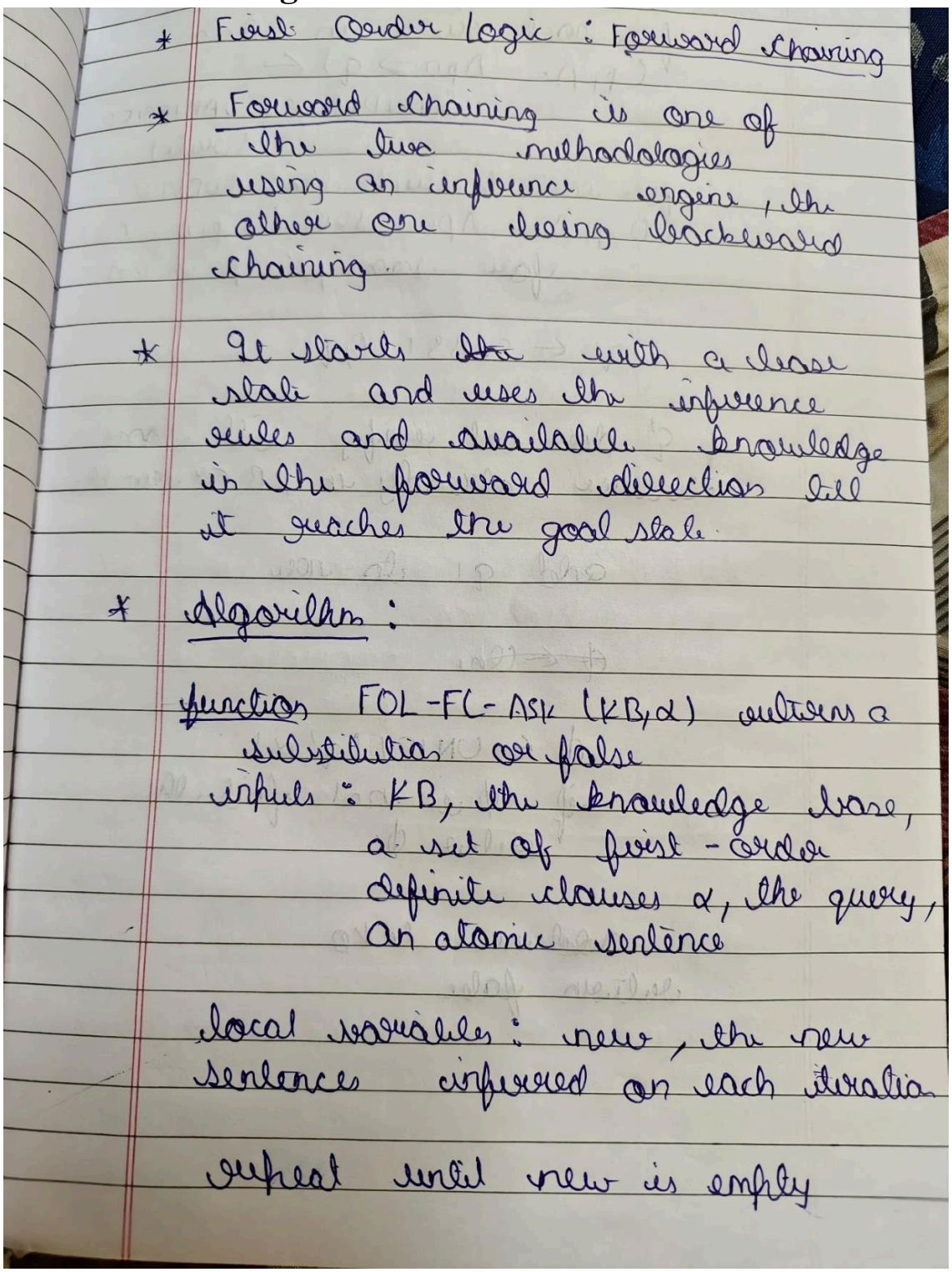
output

```
expr1 = make_predicate("Eats", "n", "Apple")
expr2 = make_predicate("Eats", "Ria", "y")

result = unify(expr1, expr2)
```

```
def unify(x, y, subs=None):
if subs is None:
subs = {}

if x == y: return
subs

# If x is a variable
if isinstance(x, str) and x.islower(): if
x in y:
return None # Occurs check subs[x] = y
return apply_substitution(subs)

# If y is a variable
if isinstance(y, str) and y.islower(): if y
in x:
return None # Occurs check subs[y] = x
return apply_substitution(subs)

# If both are predicates (e.g., Eats(x, Apple)) if
isinstance(x, tuple) and isinstance(y, tuple):
if x[0] != y[0] or len(x[1]) != len(y[1]):
return None # Predicate name or argument count mismatch for a, b
in zip(x[1], y[1]):
subs = unify(a, b, subs) if subs
is None:
return None return
subs

# Otherwise fail
return None


def apply_substitution(subs):
# Apply substitution recursively for
var, val in subs.items():
for v in list(subs.keys()):
if var in str(subs[v]) and v != var:
subs[v] = str(subs[v]).replace(var, str(val)) return subs


# Helper to create predicate terms
def make_predicate(name, *args):
```

```
return (name, list(args))


# Example
expr1 = make_predicate("Eats", "x", "Apple")
expr2 = make_predicate("Eats", "Riya", "y")
result = unify(expr1, expr2)
if result:
print("Unification Successful!")
print("Substitutions:", result)
else:
print("Unification Failed.")
```

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



* First Order Logic : Forward Chaining

* Forward Chaining is one of the two methodologies using an inference engine, the other one being backward chaining.

* It starts with a base state and uses the inference rules and available knowledge in the forward direction till it reaches the goal state.

* Algorithm :

function FOL-FC-ASK (KB, α) returns a substitution or false
inputs : KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence

local variables : new, the new sentences inferred on each iteration

repeat until new is empty

new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES (rule)
for each θ such that SUBST θ
(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p1' ∧ ... ∧ pn')
for some p1' ... pn' in KB

q' ← SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

~~ϕ ← fail~~

ϕ ← UNIFY(q', α)
if ϕ is not fail then return ϕ

add new to KB
return false

* Example knowledge base and query

```
KB = {
    forgenset(["P(x)", "Q(x)"]),
    forgenset(["P(x)", "R(x)"]),
    (forgenset(["P(x)", "Q(x)"]), "R(x)"),
    (forgenset(["P(x)", "R(x)"]), "Q(x)")
```

query = "R(y)"

* Output

Query is enforced : R(y)

"""
Simple Forward Chaining for Horn Clauses (definite clauses).

Supports:
- Facts: ground predicates like parent(alice, bob)
- Rules: body (list of predicates) => head (predicate), with variables
- Standardizing variables apart, unification, substitution
- Iterative forward-chaining: apply rules to known facts until no new facts
- Example included at
bottom """

from collections import namedtuple
import itertools
import copy

# --- Data types ---
Variable = namedtuple("Variable", ["name"]) Constant
= namedtuple("Constant", ["name"]) Predicate =
namedtuple("Predicate", ["name", "args"])

def var(name): return Variable(name) def
const(name): return Constant(name)
def pred(name, *args): return Predicate(name, list(args))

def is_var(x): return isinstance(x, Variable) def
is_const(x): return isinstance(x, Constant)

# --- Pretty printing ---
def term_str(t):
return f"?{t.name}" if is_var(t) else t.name

def pred_str(p):
return f"{p.name}({', '.join(term_str(a) for a in p.args)})"

# --- Substitution utility ---
def apply_subst_term(subst, t):
if is_var(t):
return subst.get(t.name, t) return t

def apply_subst_pred(subst, p):
return Predicate(p.name, [apply_subst_term(subst, a) for a in p.args])

# --- Standardize variables apart (unique renaming per rule application) ---
_unique_id = 0
def fresh_var_name(base):

```
global _unique_id
_unique_id += 1
return f"{base}_{_unique_id}"

def standardize(rule):
"""Given (body_list, head), return copy with fresh variable names."""
body, head = rule
    mapping = {}
  def rename(t): if
          is_var(t):
if t.name not in mapping:
mapping[t.name] = Variable(fresh_var_name(t.name)) return
mapping[t.name]
return t
new_body = [Predicate(b.name, [rename(a) for a in b.args]) for b in body]
new_head = Predicate(head.name, [rename(a) for a in head.args])
return new_body, new_head

# --- Unification (simple) ---
def unify_terms(t1, t2, subst):
# apply current substitutions if
is_var(t1):
t1 = subst.get(t1.name, t1) if
is_var(t2):
t2 = subst.get(t2.name, t2)

# variable cases
if is_var(t1) and is_var(t2) and t1.name == t2.name:
return subst
if is_var(t1):
new    =    subst.copy()
new[t1.name]    =    t2
return new
if is_var(t2):
new    =    subst.copy()
new[t2.name]    =    t1
return new
# constants
if is_const(t1) and is_const(t2) and t1.name == t2.name: return
subst
return None

def unify_preds(p1, p2, subst=None):
if p1.name != p2.name or len(p1.args) != len(p2.args):
return None
s = {} if subst is None else subst.copy() for
```

```
a1, a2 in zip(p1.args, p2.args):
s = unify_terms(a1, a2, s)
if s is None: return
None
return s

# --- Forward chaining algorithm ---
def forward_chain(kb_rules, kb_facts, query=None, verbose=True):
"""
kb_rules: list of (body_list, head_pred)
kb_facts: list of ground Predicate
query: optional Predicate to check entailment
Returns: (inferred_facts_list, substitution_if_query_proved_or_None) """
known = list(kb_facts) # growing list of facts if
verbose:
print("Initial facts:")
for f in known: print(" ", pred_str(f))
if query: print("Query:", pred_str(query))
print("-" * 50)

while True: new_facts
= []
for i, rule in enumerate(kb_rules, start=1):
body, head = standardize(rule) # fresh copy with unique vars if
verbose:
print(f"Trying Rule {i}: {' ∧ '.join(pred_str(b) for b in body)} => {pred_str(head)}") # find
candidate known facts for each literal by matching predicate name and arity candidates_lists =
[
[f for f in known if f.name == lit.name and len(f.args) == len(lit.args)] for lit in
body
]
if not candidates_lists: # rule with empty body (fact) theta = {}
inferred = apply_subst_pred(theta, head)
if inferred not in known and inferred not in new_facts:
new_facts.append(inferred)
continue

# try all combinations
for combo in itertools.product(*candidates_lists): theta = {}
ok = True
for lit, fact in zip(body, combo): theta =
unify_preds(lit, fact, theta) if theta is None:
ok = False break
if not ok:
```

```
continue
qprime = apply_subst_pred(theta, head)
# check groundness (only store ground facts here) if
any(is_var(arg) for arg in qprime.args):
# skip storing non-ground inferred atoms in this simple implementation if verbose:
print(" Derived non-ground atom (skipped storing):", pred_str(qprime))
else:
if all(not (qprime.name == f.name and [a.name for a in qprime.args] == [b.name for b in f.args]) for f in
known+new_facts):
if verbose:
print(" Inferred:", pred_str(qprime), "using", [pred_str(c) for c in combo], "with θ
=", theta)


if not new_facts:
if verbose: print("No new facts inferred; stopping.") return
known, None
# add new facts to known and iterate again
known.extend(new_facts)
if verbose:
print("Added facts:", ", ".join(pred_str(f) for f in new_facts))
print("-" * 50)

# --- Example usage ---
if _name____ == "_main_": #
Facts (ground)
facts = [
pred("parent", const("alice"), const("bob")),
pred("parent", const("bob"), const("charlie")),
pred("mother", const("diana"), const("bob")), # extra fact to show mother=>parent rule
]

# Rules
# parent(x,y) ∧ parent(y,z) => grandparent(x,z)
r1_body = [ pred("parent", var("x"), var("y")), pred("parent", var("y"), var("z")) ]
r1_head = pred("grandparent", var("x"), var("z"))

# mother(x,y) => parent(x,y)
r2_body = [ pred("mother", var("x"), var("y")) ]
```

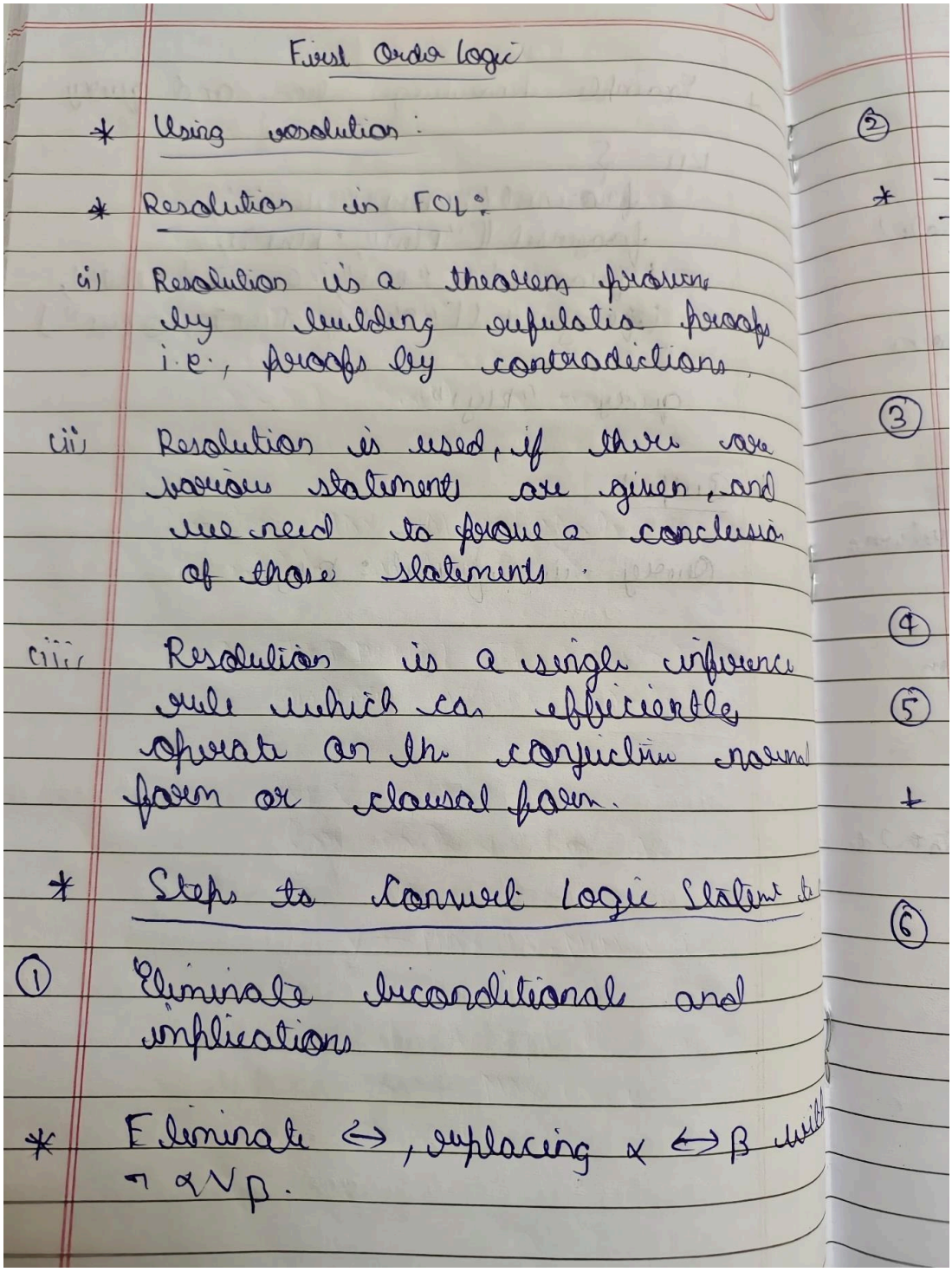
```
r2_head = pred("parent", var("x"), var("y"))
rules = [ (r1_body, r1_head), (r2_body, r2_head) ] #

Query to check

query = pred("grandparent", const("alice"), const("charlie"))
inferred, subst = forward_chain(rules, facts, query=query, verbose=True)
print("\nFinal KB facts:")
for f in inferred:
print(" ", pred_str(f))

if subst:
print("\nQuery succeeded with substitution:", subst) else:
print("\nQuery not proved by forward chaining.")
```

OUTPUT:

```
Initial facts:
parent(alice, bob)
parent(bob, charlie)
mother(diana, bob)
Query: grandparent(alice, charlie)
```

```
Trying Rule 1: parent(?x_1, ?y_2) ∧ parent(?y_2, ?z_3) => grandparent(?x_1, ?z_3)
Inferred: grandparent(alice, charlie) using ['parent(alice, bo...), 'y_2': Constant(name='bob'), 'z_3':
Constant(name='charlie')]
*** Query proved by: grandparent(alice, charlie) with substitution {}
Final KB facts:
parent(alice, bob) parent(bob,
charlie) mother(diana, bob)
grandparent(alice, charlie)

Query not proved by forward chaining.
```

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



First Order logic

* Using resolution:

* Resolution in FOL:

(i) Resolution is a theorem proving by refuting statements proofs i.e., proofs by contradictions.

(ii) Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements.

(iii) Resolution is a single inference rule which can efficiently operate on the conjuction normal form or clausal form.

* Steps to Convert Logic Statements

(1) Eliminate biconditionals and implications

* Eliminate $\leftrightarrow$, replacing $\alpha \leftrightarrow \beta$ with $\neg\alpha \vee \beta$.

② Move ¬ inwards

* $\neg(\forall x p) \equiv \exists x \neg p$
$\neg(\exists x p) \equiv \forall x \neg p$
$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
$\neg\neg\alpha \equiv \alpha$

③ Standardize variables ~~apart~~ apart by a Skolem constant or Skolem function of the enclosing universally quantified variables.

④ Skolemize : each existential variable

⑤ Drop universal quantifier.

* For instance, $\forall x$ Person(x) becomes Person(x).

⑥ Distribute $\wedge$ over $\vee$

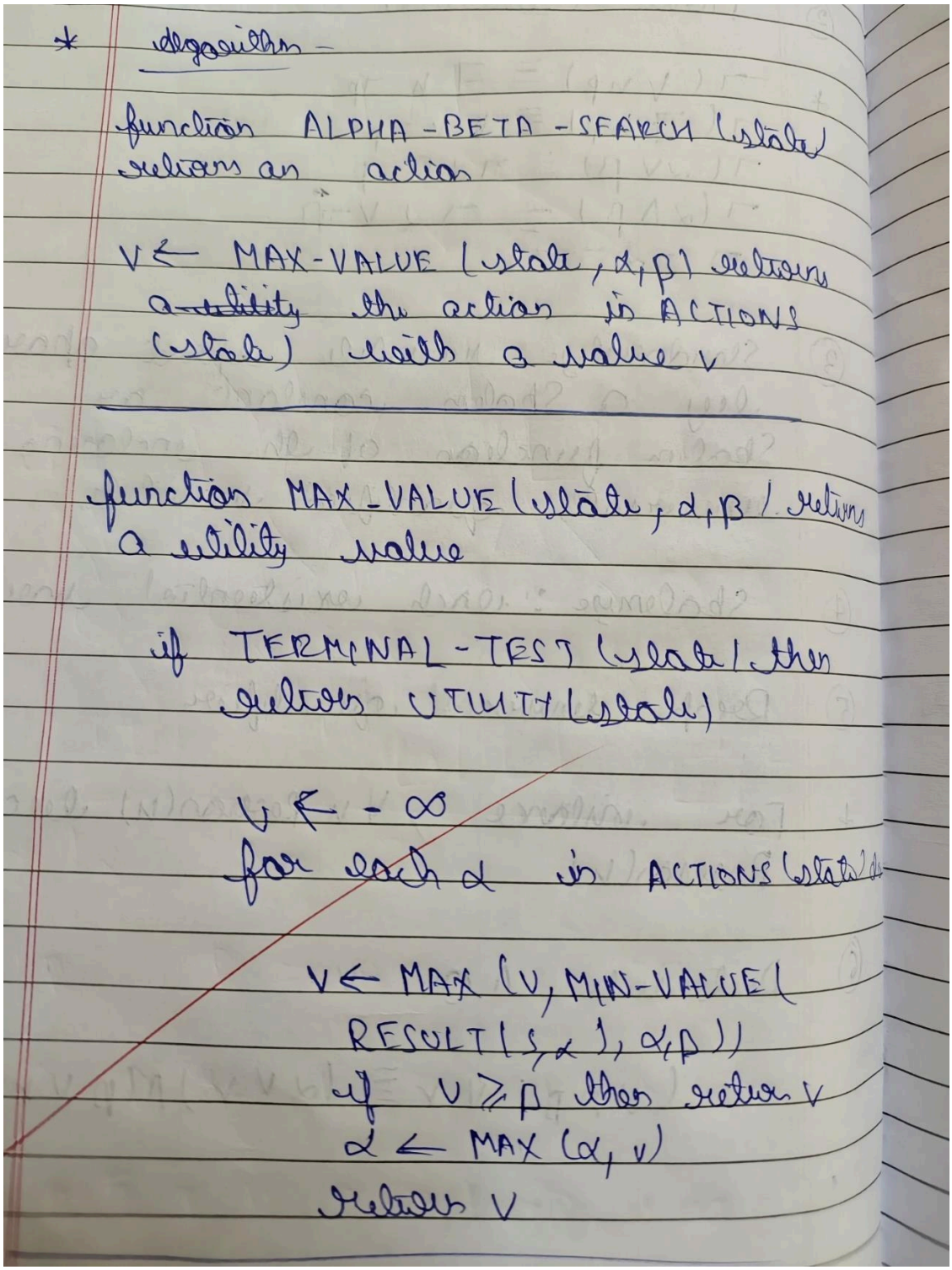
• $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

```
import copy

def unify(x, y, theta={}):
if theta is None:
return None elif
x == y:
return theta
elif is_variable(x):
return unify_var(x, y, theta) elif
is_variable(y):
return unify_var(y, x, theta)
elif is_predicate(x) and is_predicate(y):
if x[0] != y[0] or len(x[1]) != len(y[1]): return
None
return unify(x[1], y[1], theta)
elif isinstance(x, list) and isinstance(y, list): if
len(x) != len(y):
return None
if not x: return theta
first = unify(x[0], y[0], theta) return
unify(x[1:], y[1:], first)
else:
return None

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:
theta2 = theta.copy()
theta2[var] = x return
theta2

def is_variable(x):
return isinstance(x, str) and x[0].islower()

def is_predicate(x):
return isinstance(x, tuple) and isinstance(x[1], list)

# ---------- APPLY SUBSTITUTION ----------
def substitute(clause, theta):
new_clause = []
for sign, pred in clause:
new_args = []
for a in pred[1]:
```

```
if isinstance(a, str) and a in theta:
new_args.append(theta[a]) else:
new_args.append(a) new_clause.append((sign,
(pred[0], new_args)))
return new_clause

# ---------- RESOLUTION ----------
def resolve(ci, cj):
resolvents = [] for
(si, pi) in ci:
for (sj, pj) in cj:
if si != sj: # opposite polarity theta =
unify(pi, pj, {})
if theta is not None:
# apply substitution
new_ci = substitute([x for x in ci if x != (si, pi)], theta) new_cj =
substitute([x for x in cj if x != (sj, pj)], theta) new_clause = new_ci
+ new_cj

# remove duplicates cleaned = []
for lit in new_clause: if lit not in
cleaned:
cleaned.append(lit)

resolvents.append(cleaned)

return resolvents

# ---------- RESOLUTION LOOP ----------
def fol_resolution(kb, query):
# Add negated query
neg_query = [("~", query)]
clauses = kb + [neg_query]

print("Initial clauses:") for
c in clauses:
print(c)
print()

new = [] while
True:
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: resolvents =
resolve(ci, cj)
```

```
for r in resolvents:
print(f"Resolve {ci} AND {cj} -> {r}") if r == []:
print("\n✔ Empty clause produced — Query is TRUE.")
return True if r not in
new:
new.append(r)

if not new:
print("\n❌ No new clauses — Query is FALSE.")
return False

for c in new:
clauses.append(c)
new = []

#  ------- EXAMPLE -------
if _name____== "_main_": #
KB:
# 1. Human(x) → Mortal(x)
# 2. Human(Socrates)

KB = [
[("~", ("Human", ["x"])), ("+", ("Mortal", ["x"]))], # ¬Human(x) ∨ Mortal(x)
[("+", ("Human", ["Socrates"]))]                    # Human(Socrates)
]

query = ("Mortal", ["Socrates"])

result = fol_resolution(KB, query)
print("\nOUTPUT:", result)
```

9. Implement Alpha-Beta Pruning.



* Algorithm –

function ALPHA-BETA-SEARCH (state) returns an action

V ← MAX-VALUE (state, α, β) returns ~~a utility~~ the action in ACTIONS (state) with a value v

---

function MAX-VALUE (state, α, β) returns a utility value

if TERMINAL-TEST (state) then return UTILITY (state)

V ← -∞

for each a in ACTIONS (state) do

V ← MAX (V, MIN-VALUE (RESULT (s, a), α, β))

if V ≥ β then return V

α ← MAX (α, v)

return V

function MIN-VALUE (state, α, β)
returns a utility value

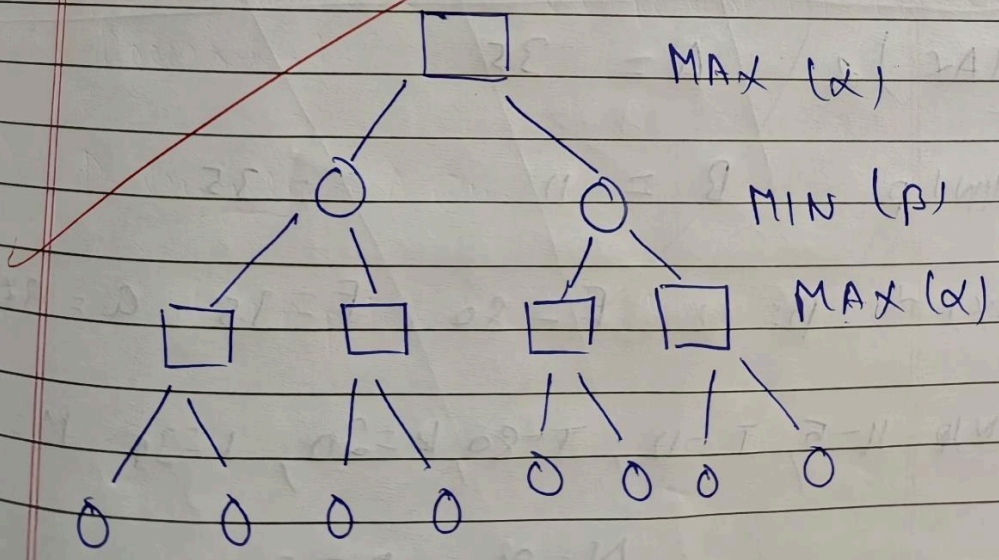
if TERMINAL-TEST (state) then
return UTILITY (state)

$v \leftarrow +\infty$

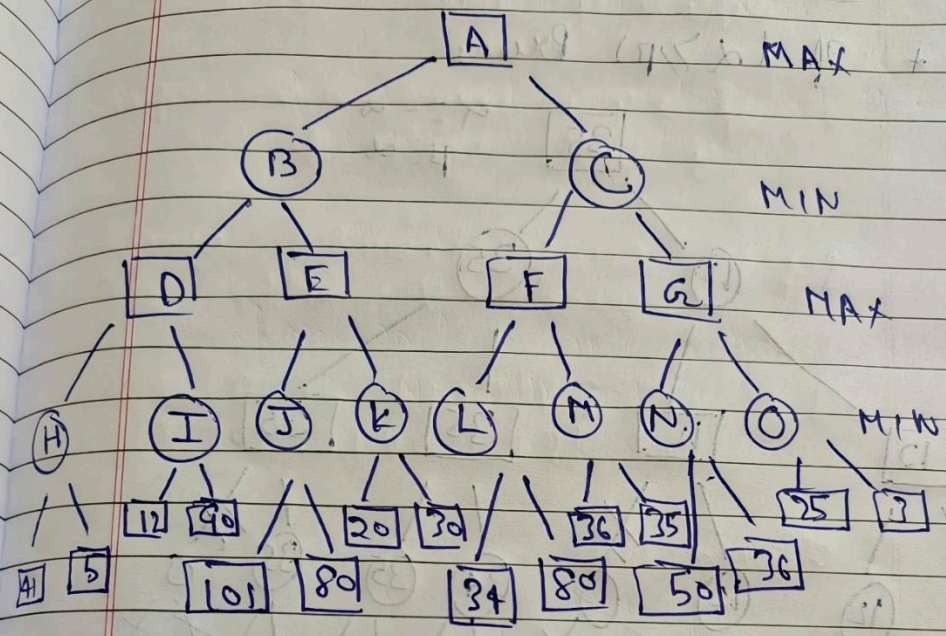
for each α in actions (state) do
$v \leftarrow$ MIN (v, MAX-VALUE (
RESULT (s, α), α, β))

if $v \leq \alpha$ then return v
$\beta \leftarrow$ MIN (β, v)
return v

* Solution



Q. Problem 2:



* H = MIN (41, 5) = 5
  I = MIN (12, 90) = 12
  J = MIN (101, 80) = 80
  K = MIN (20, 30) = 20
  L = MIN (34, 80) = 34
  M = MIN (36, 35) = 35
  N = MIN (50, 36) = 36
  O = MIN (25, 3) = 3

* D = MAX (H, I) = 12
  E = MAX (J, K) = 80
  F = MAX (L, M) = 35
  G = MAX (N, O) = 36

B = MIN (D, E) = 12
C = MIN (F, G) = 35

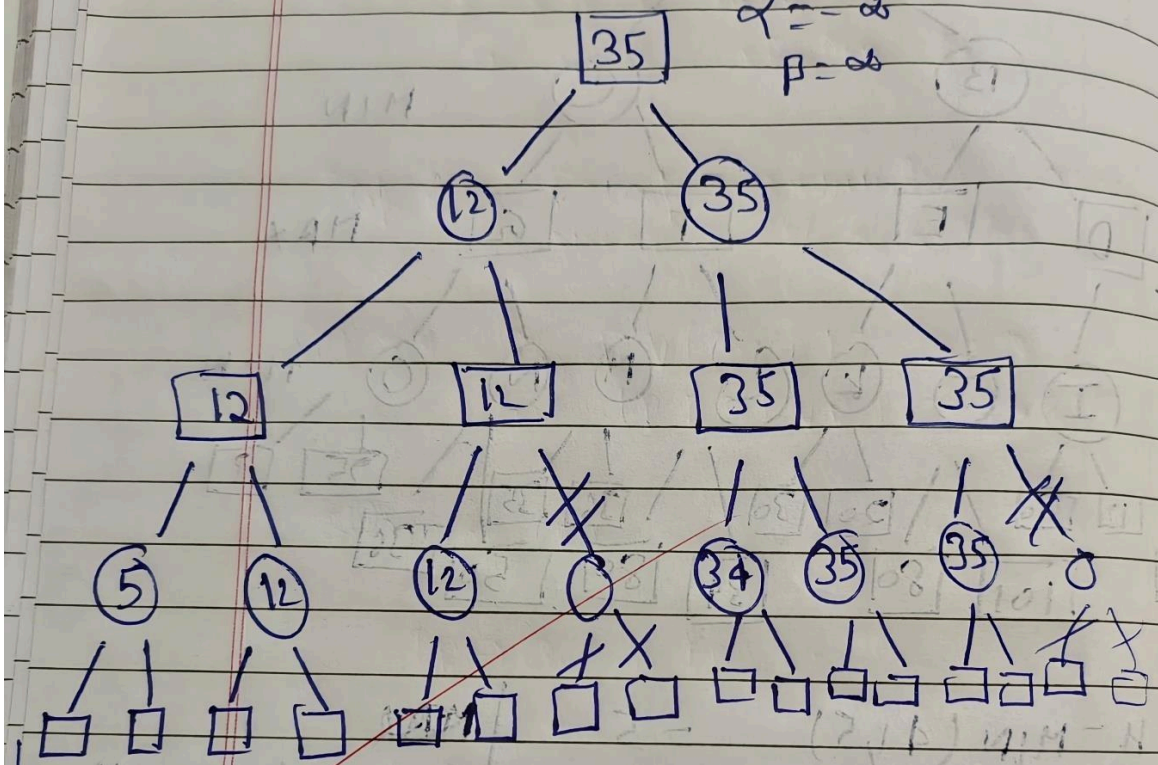
A = MAX (B, C)

⇓

A = 35

* Pruning-

* If ($\alpha \geq \beta$) Prune.



```
INF = float('inf')

# Leaf values (as given in the diagram)
leaves = {
'H': [41, 5],
'I': [12, 90],
'J': [101, 80],
'K': [20, 30],
'L': [34, 80],
'M': [36, 35],
'N': [50, 36],
'O': [25, 3]
}

# Tree structure from diagram
tree = {
'A': ['B', 'C'],
'B': ['D', 'E'],
'C': ['F', 'G'],
'D': ['H', 'I'],
'E': ['J', 'K'],
'F': ['L', 'M'],
'G': ['N', 'O']
}

# Node types
MAX = {'A', 'D', 'E', 'F', 'G'}
MIN = {'B', 'C', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O'}

pruned = []  # store pruned nodes


def alphabeta(node, alpha, beta):
# If leaf node bottom layer
if node in leaves:
return min(leaves[node])              # all leaves are MIN nodes

# MAX node
if node in MAX:
value = -INF
for child in tree[node]:
child_value = alphabeta(child, alpha, beta) value =
```

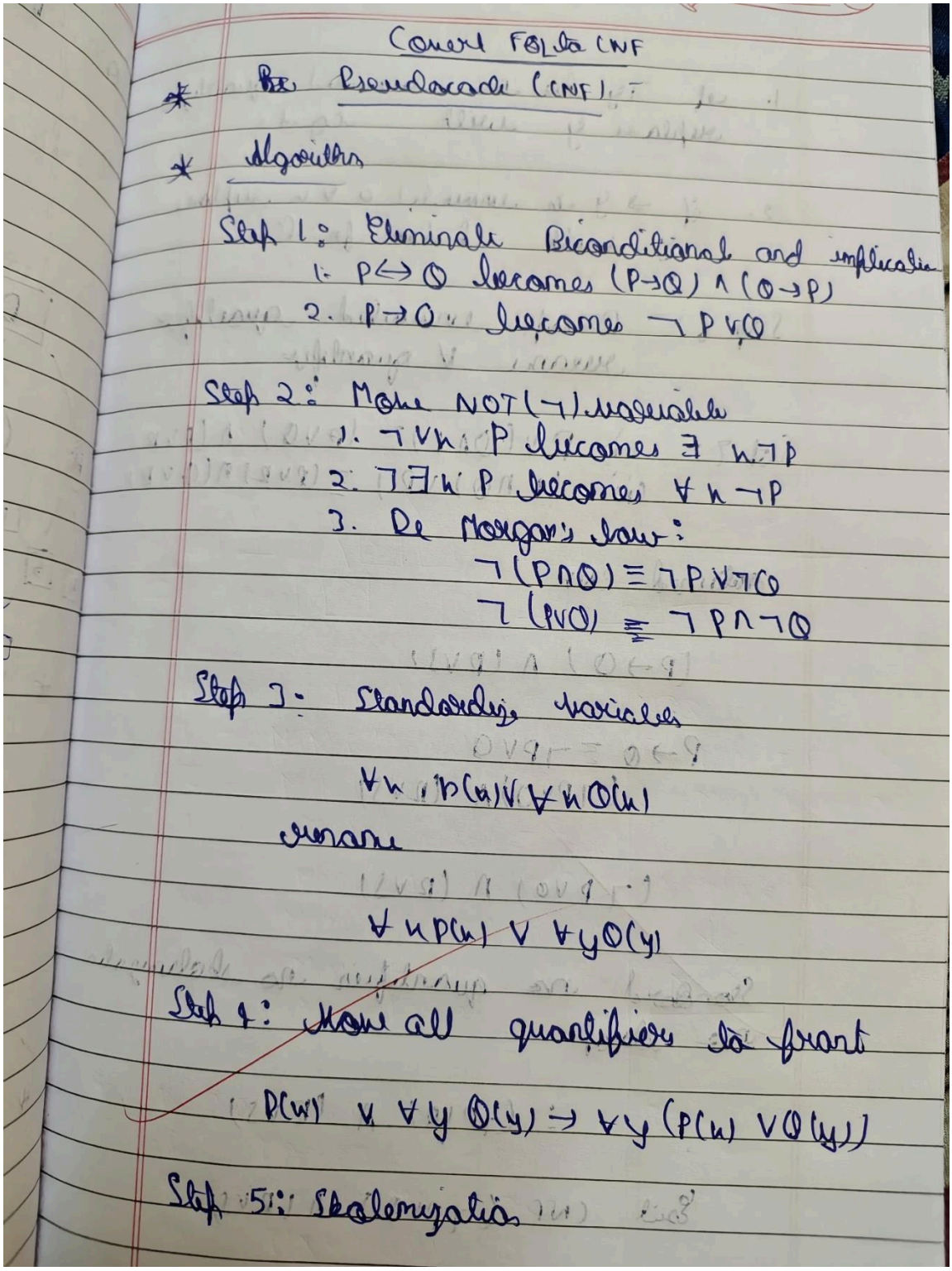
```
max(value, child_value)
alpha = max(alpha, value)
# PRUNE for MAX
if alpha >= beta:
pruned.extend(tree[node][tree[node].index(child)+1:]) break
return value

# MIN node
else:
value = INF
for child in tree[node]:
child_value = alphabeta(child, alpha, beta) value =
min(value, child_value)
beta = min(beta, value)

# PRUNE for MIN
if beta <= alpha:
pruned.extend(tree[node][tree[node].index(child)+1:]) break
return value


# Run alpha-beta
root_value = alphabeta('A', -INF, INF)

print("Root Value =", root_value)
print("Pruned Nodes =", pruned)
```

OUTPUT:

Root Value = 35

Pruned Nodes = ['K', 'O']

10. FOL to CNF



Convert FOL to CNF

* Pseudocode (CNF):

* Algorithm

Step 1: Eliminate Biconditional and implication
1. $P \leftrightarrow Q$ becomes $(P \rightarrow Q) \wedge (Q \rightarrow P)$
2. $P \rightarrow Q$ becomes $\neg P \vee Q$

Step 2: Move NOT ($\neg$) inwards
1. $\neg \forall x : P$ becomes $\exists x \neg P$
2. $\neg \exists x P$ becomes $\forall x \neg P$
3. De Morgan's law:
$\neg (P \wedge Q) \equiv \neg P \vee \neg Q$
$\neg (P \vee Q) \equiv \neg P \wedge \neg Q$

Step 3: Standardize variables

$\forall x P(x) \vee \forall x Q(x)$

rename

$\forall x P(x) \vee \forall y Q(y)$

Step 4: Move all quantifiers to front

$P(x) \vee \forall y Q(y) \rightarrow \forall y (P(x) \vee Q(y))$

Step 5: Skolemization

1. if $\exists y$ has no universal quantifier replace y with const c

2- if $\rightarrow y$ is universal a $\forall x$ replace y with skolem fn $= f(x)$

Step 6: Drop universal quantifier means $\forall$ quantifier

Step 7: (i) $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$
(ii) $(P \wedge Q) \vee R) \equiv (P \vee R) \wedge (Q \vee R)$

obtained CNF

$(P \rightarrow Q) \wedge (R \vee S)$

$P \rightarrow Q \equiv \neg P \vee Q$

$(\neg P \vee Q) \wedge (R \vee S)$

$(\neg P \vee Q) \wedge (R \vee S)$

Standard no quantifier no skolemization no drop

$\therefore (\neg P \vee Q) \wedge (R \vee S)$

Exit CNF $(\neg P \vee Q) \wedge (R \vee S)$

Outputs

Input : $(P \rightarrow Q) \ \& \ (R \mid S)$

CNF : $(C : P \ \& : Q) \ \& (R \mid S)$

CODE:

```
import itertools

# ----- Expression Constructors -----

def AND(*args): return ("and", list(args))
def OR(*args): return ("or", list(args)) def
NOT(a): return ("not", a)
def IMP(a, b): return ("imp", a, b)
def IFF(a, b): return ("iff", a, b)
def FORALL(v, a): return ("forall", v, a)
def EXISTS(v, a): return ("exists", v, a)
def PRED(name, *args): return ("pred", name, list(args))

# ----- Eliminate <=> and => -----
def remove_implications(expr):
t = expr[0] if t
== "iff":
A, B = expr[1], expr[2]
return AND(remove_implications(IMP(A, B)), remove_implications(IMP(B, A))) if t
== "imp":
A, B = expr[1], expr[2]
return OR(NOT(remove_implications(A)), remove_implications(B)) if t
in ["and", "or"]:
return (t, [remove_implications(e) for e in expr[1]]) if t
in ["not", "forall", "exists"]:
return (t, remove_implications(expr[1])) return
expr

# ----- Move NOT inward -----
def push_negation(expr):
t = expr[0]

if t == "not": e =
expr[1]
if e[0] == "not":
return push_negation(e[1]) if e[0]
== "and":
return OR(*[push_negation(NOT(x)) for x in e[1]]) if e[0]
== "or":
return AND(*[push_negation(NOT(x)) for x in e[1]]) if e[0]
== "forall":
```

```
return EXISTS(e[1], push_negation(NOT(e[2]))) if e[0]
== "exists":
return FORALL(e[1], push_negation(NOT(e[2]))) return
("not", push_negation(e))

if t in ["and", "or"]:
return (t, [push_negation(x) for x in expr[1]])

if t in ["forall", "exists"]:
return (t, expr[1], push_negation(expr[2])) return

expr


# ----- Standardize variables -----

counter = itertools.count()

def standardize(expr, mapping=None):
if mapping is None:
mapping = {} t =

expr[0]

if t in ["forall", "exists"]: v
= expr[1]
new_v = v + "_" + str(next(counter)) mapping[v]
= new_v
return (t, new_v, standardize(expr[2], mapping))

if t in ["and", "or"]:
return (t, [standardize(x, mapping) for x in expr[1]])

if t == "pred":
name, args = expr[1], expr[2]
new_args = [mapping.get(a, a) for a in args] return
("pred", name, new_args)

if t == "not":
return ("not", standardize(expr[1], mapping)) return

expr


# ----- Skolemization -----
```

```
sk_count = itertools.count()

def skolemize(expr, scope=None):
if scope is None:
scope = [] t =

expr[0]

if t == "forall":
return FORALL(expr[1], skolemize(expr[2], scope + [expr[1]]))

if t == "exists":
var = expr[1]
sk_name = "SK" + str(next(sk_count))

if scope:
return substitute(expr[2], var, sk_name + "(" + ",".join(scope) + ")") else:
return substitute(expr[2], var, sk_name)

if t in ["and", "or"]:
return (t, [skolemize(x, scope) for x in expr[1]])

if t == "not":
return ("not", skolemize(expr[1], scope)) return

expr


def substitute(expr, old, new):
t = expr[0]

if t == "pred":
return ("pred", expr[1], [new if a == old else a for a in expr[2]])

if t in ["and", "or"]:
return (t, [substitute(x, old, new) for x in expr[1]])

if t == "not":
return ("not", substitute(expr[1], old, new))

if t in ["forall", "exists"]:
return (t, expr[1], substitute(expr[2], old, new)) return

expr
```

```
# ----- Drop universal quantifiers -----
def drop_quantifiers(expr):
if expr[0] == "forall":
return drop_quantifiers(expr[2]) if
expr[0] in ["and", "or"]:
return (expr[0], [drop_quantifiers(x) for x in expr[1]]) if
expr[0] == "not":
return ("not", drop_quantifiers(expr[1])) return
expr

# ----- Distribute OR over AND -----
def distribute(expr):
if expr[0] != "or": return
expr

a, b = expr[1][0], expr[1][1]

a = distribute(a)
b = distribute(b)

if a[0] == "and":
return AND(*[distribute(OR(x, b)) for x in a[1]]) if
b[0] == "and":
return AND(*[distribute(OR(a, x)) for x in b[1]]) return

OR(a, b)

# ----- Full CNF conversion -----
def to_cnf(expr):
expr = remove_implications(expr)
expr = push_negation(expr)
expr = standardize(expr) expr
= skolemize(expr)
expr = drop_quantifiers(expr) expr
= distribute(expr)
return expr


# ----- Example -----

if __name__ == "__main__": #
Example FOL:
# ∀x ( P(x) → ∃y Q(x,y) )
```

```
formula = FORALL("x", IMP(PRED("P", "x"), EXISTS("y", PRED("Q", "x", "y"))))

print("Original FOL:")
print(formula)

cnf = to_cnf(formula)

print("\nCNF:")
print(cnf)
```