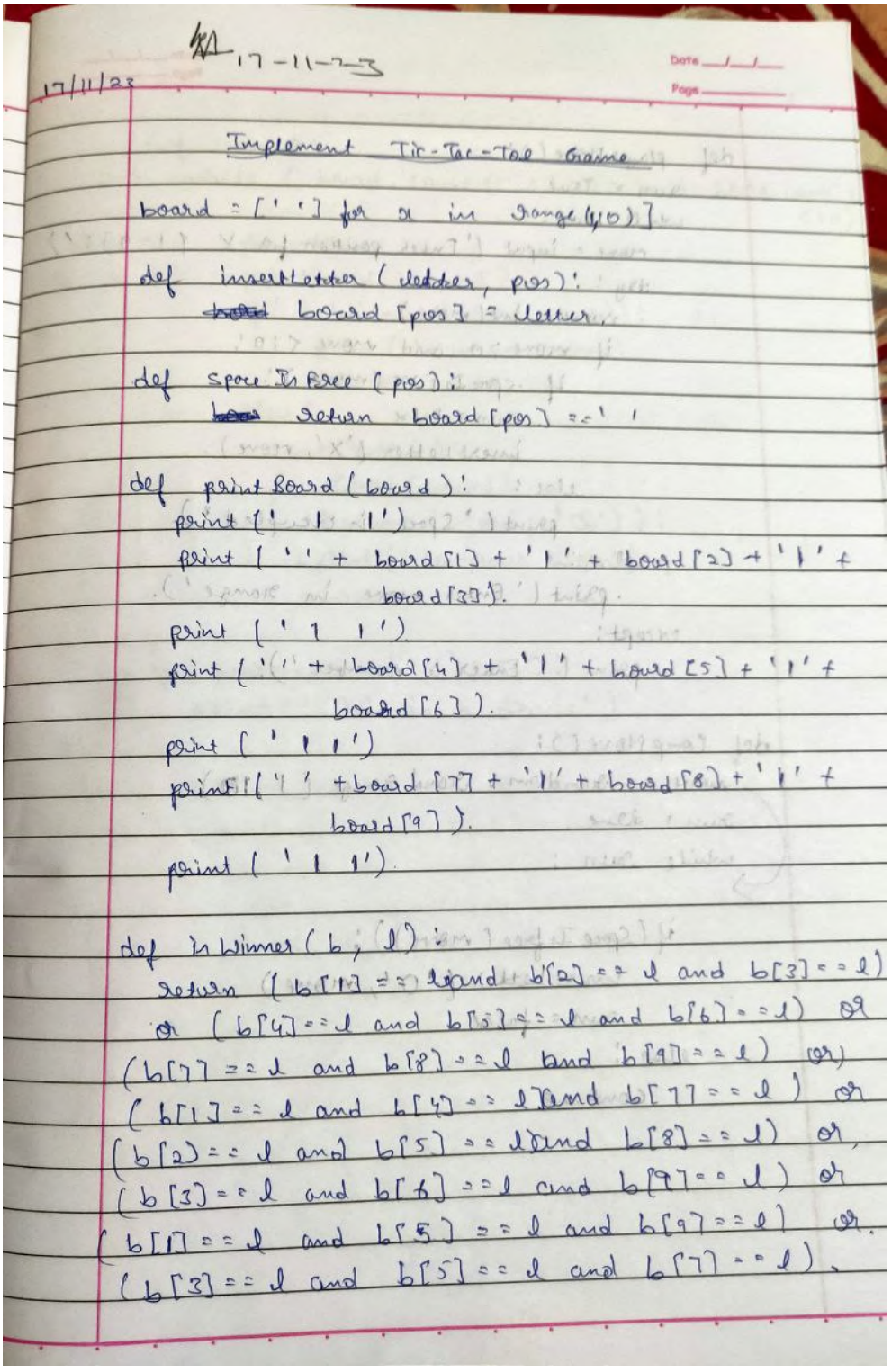
Program 1: Implement Tic Tac Toe

Observation:



17/11/23

17-11-23

Implement Tic-Tac-Toe Game

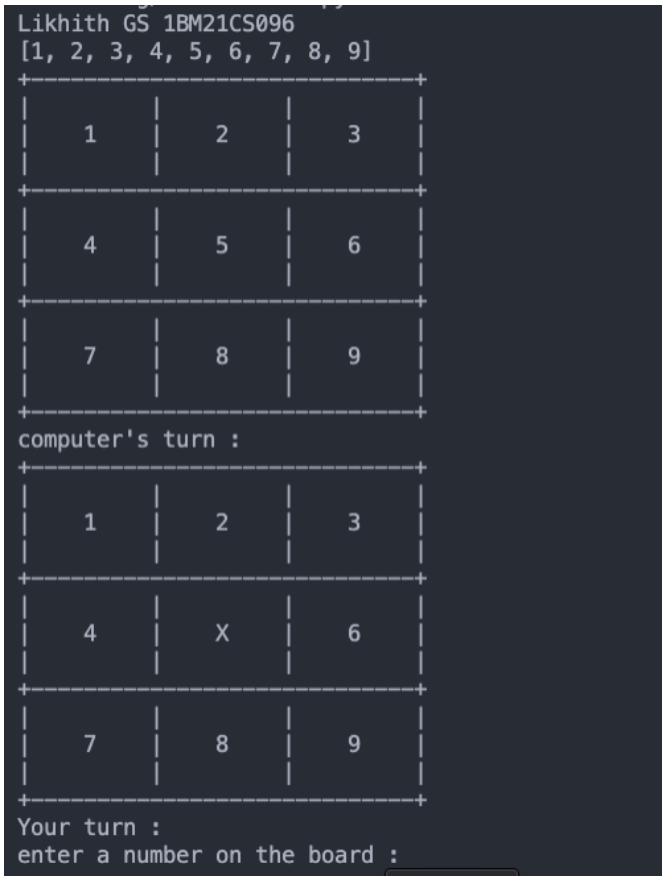
```
board = [' ' for x in range(10)]

def insertLetter(letter, pos):
    board[pos] = letter

def spaceIsFree(pos):
    return board[pos] == ' '

def printBoard(board):
    print(' | | ')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' +
          board[3])
    print(' | | ')
    print('-' + board[4] + '-' + board[5] + '-' +
          board[6])
    print(' | | ')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' +
          board[9])
    print(' | | ')

def isWinner(b, l):
    return (b[7] == l and b[8] == l and b[9] == l) or
    (b[4] == l and b[5] == l and b[6] == l) or
    (b[1] == l and b[2] == l and b[3] == l) or
    (b[1] == l and b[4] == l and b[7] == l) or
    (b[2] == l and b[5] == l and b[8] == l) or
    (b[3] == l and b[6] == l and b[9] == l) or
    (b[1] == l and b[5] == l and b[9] == l) or
    (b[3] == l and b[5] == l and b[7] == l)
```

```
def playerMove():
    run = True
    while run:
        move = input('Enter position for X (1-9):')
        try:
            move = int(move)
            if move > 0 and move < 10:
                if spaceIsFree(move):
                    run = False
                    insertLetter('X', move)
                else:
                    print('Space is occupied')
            else:
                print('Enter move in range')
        except:
            print('Enter a number')

def compMove():
    move = random.randrange(1,10)
    run = True
    while run:

        if(spaceIsfree(move)):
            insertLetter('O', move)
            run = false
        else:
            continue
```

```
def main():
    while (board.count(' ') > 0 and board.count('
                                                 < 10):
        playerMove():
        printBoard(board)
        if (isWinner(board, 'X')):
            print('Human won')
            break.
        else:
            compMove().
            printBoard(board).
            if (isWinner(board, 'O')):
                print('computer won').
                break.

    if (board.count(' ') == 0:
        print('This is a draw').
```

Output:

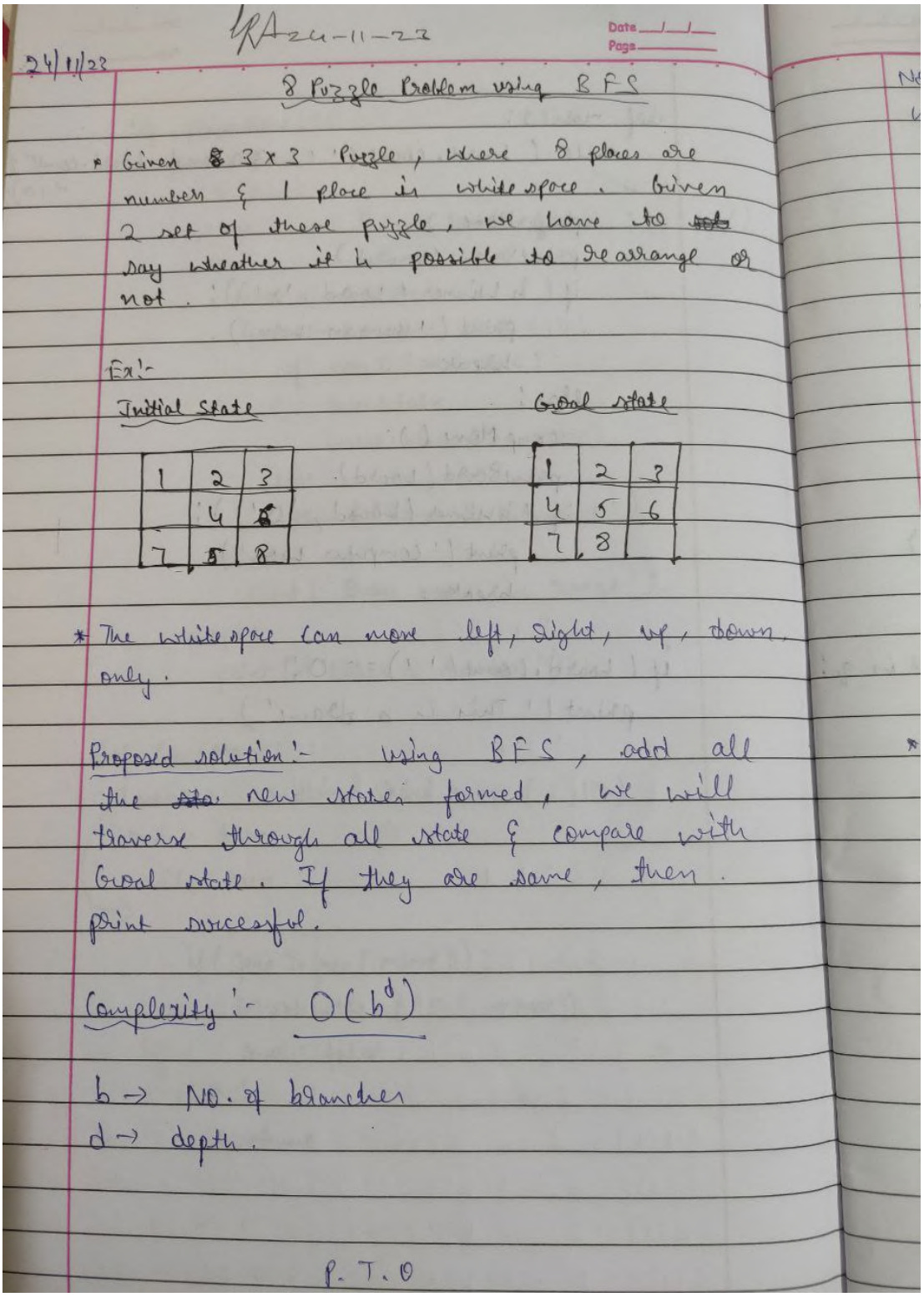
In the Beginning:



At the End:

Program 2 : 8 Puzzle Breadth First Search Algorithm

Observation:



24/11/23

RA2111123

Date / /
Page

8 Puzzle Problem using BFS

* Given a 3 x 3 Puzzle, where 8 places are numbers & 1 place is white space. Given 2 set of these puzzle, we have to say weather it is possible to rearrange or not.

Ex:-

Initial State

1	2	3
	4	6
7	5	8

Goal state

1	2	3
4	5	6
7	8	

* The white space can move left, right, up, down only.

Proposed solution:- using BFS, add all the new states formed, we will traverse through all state & compare with Goal state. If they are same, then print succesful.

Complexity :- $O(b^d)$

b → No. of branches
d → depth.

P.T.O

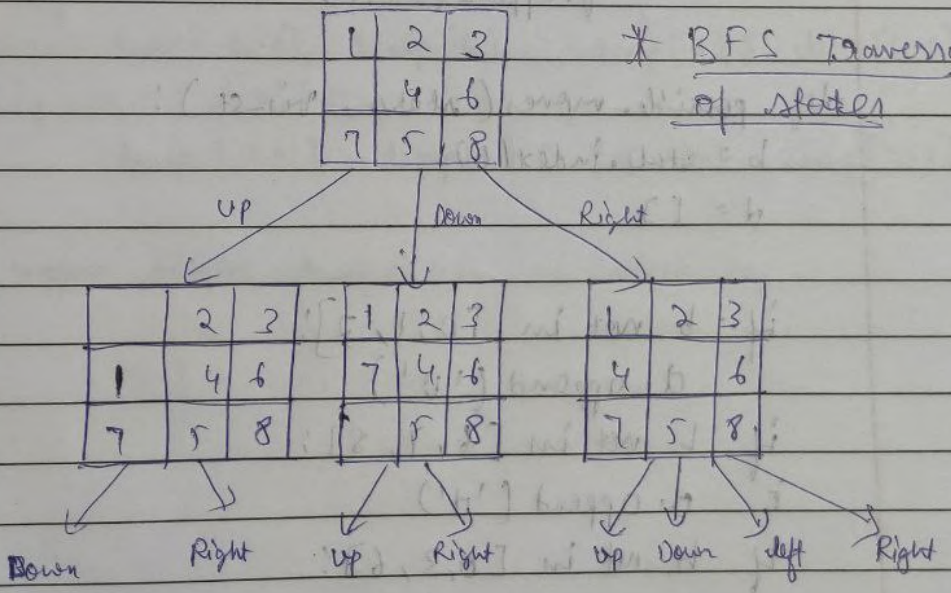
No. of branches depend on the position of white space.

•	x	•
x	*	x
•	x	•

'•' → ~~Only~~ 2 ways to branch.
x → ~~and~~ 3 ways to branch.
* → 4 ways to branch.

So, avg b value ⇒ $(4 \times 2 + 3 \times 4 + 1 \times 4)/9$
⇒ 2.6

At most, ~~Ato~~ Depth = 20
$\approx O(2.6^{20})$ → At worst case.



* BFS Traversal of states

```
def bfs(src, targ):
    q = []
    q.append(src)
    vis = []
    while len(q) > 0:
        src = q.pop(0)
        vis.append(src)
        print(src)
        if src == targ:
            print("success")
            return
        poss_moves = []
        poss_moves = possible_moves(src, vis)

        for move in poss_moves:
            if move not in vis and move not in q:
                q.append(move)

def possible_moves(state, vis_ct):
    b = state.index(0)
    d = []

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

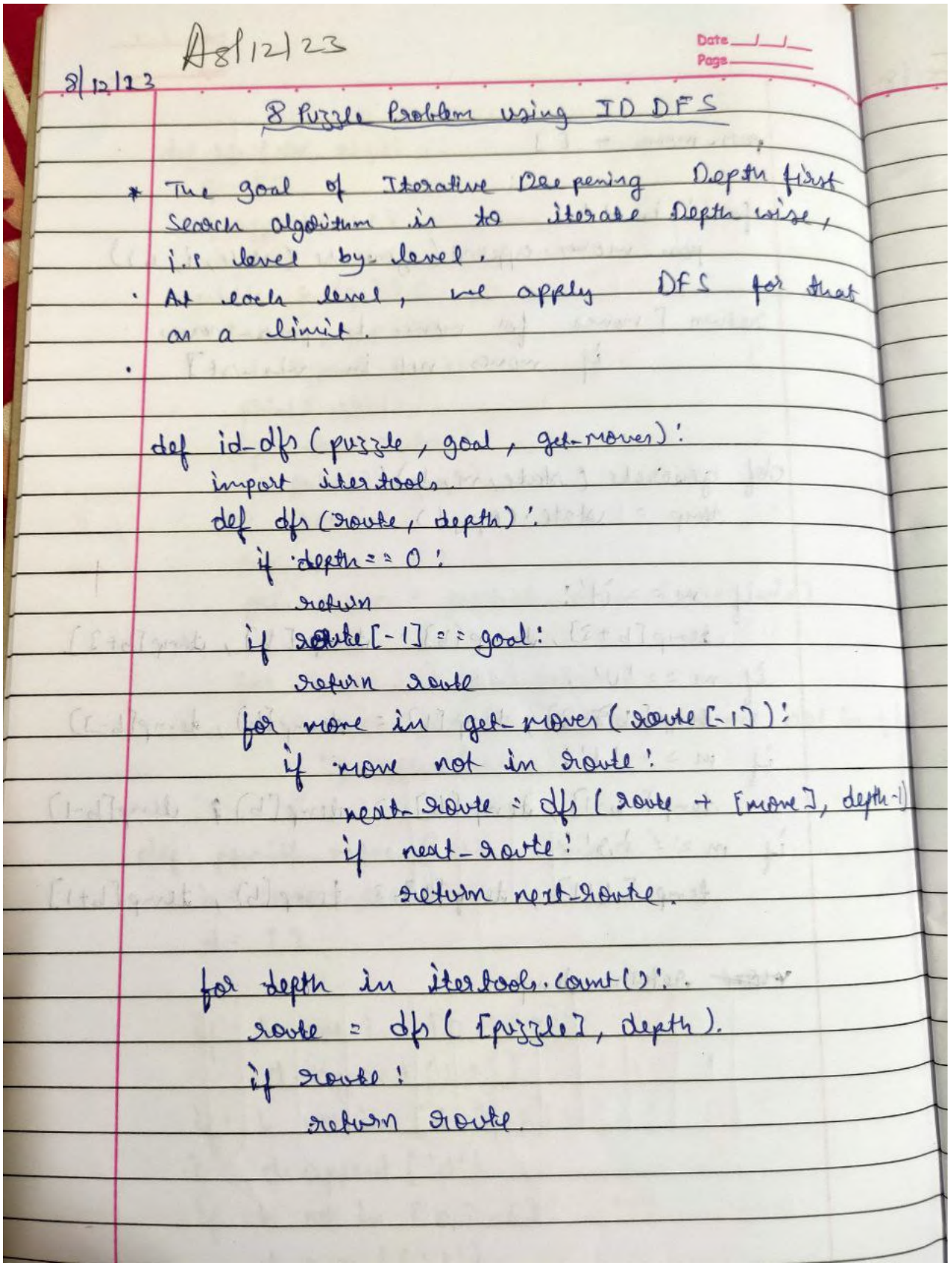
```
poss_moves = []
for i in d:
    poss_moves.append(generate(state, i, b))
return [moves for moves in poss_moves
        if moves not in vis_st]

def generate(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
```

Output:

```
Likhith GS-1BM21CS096
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 Puzzle Iterative Deepening Search Algorithm

Observation :



8/12/23

8/12/23

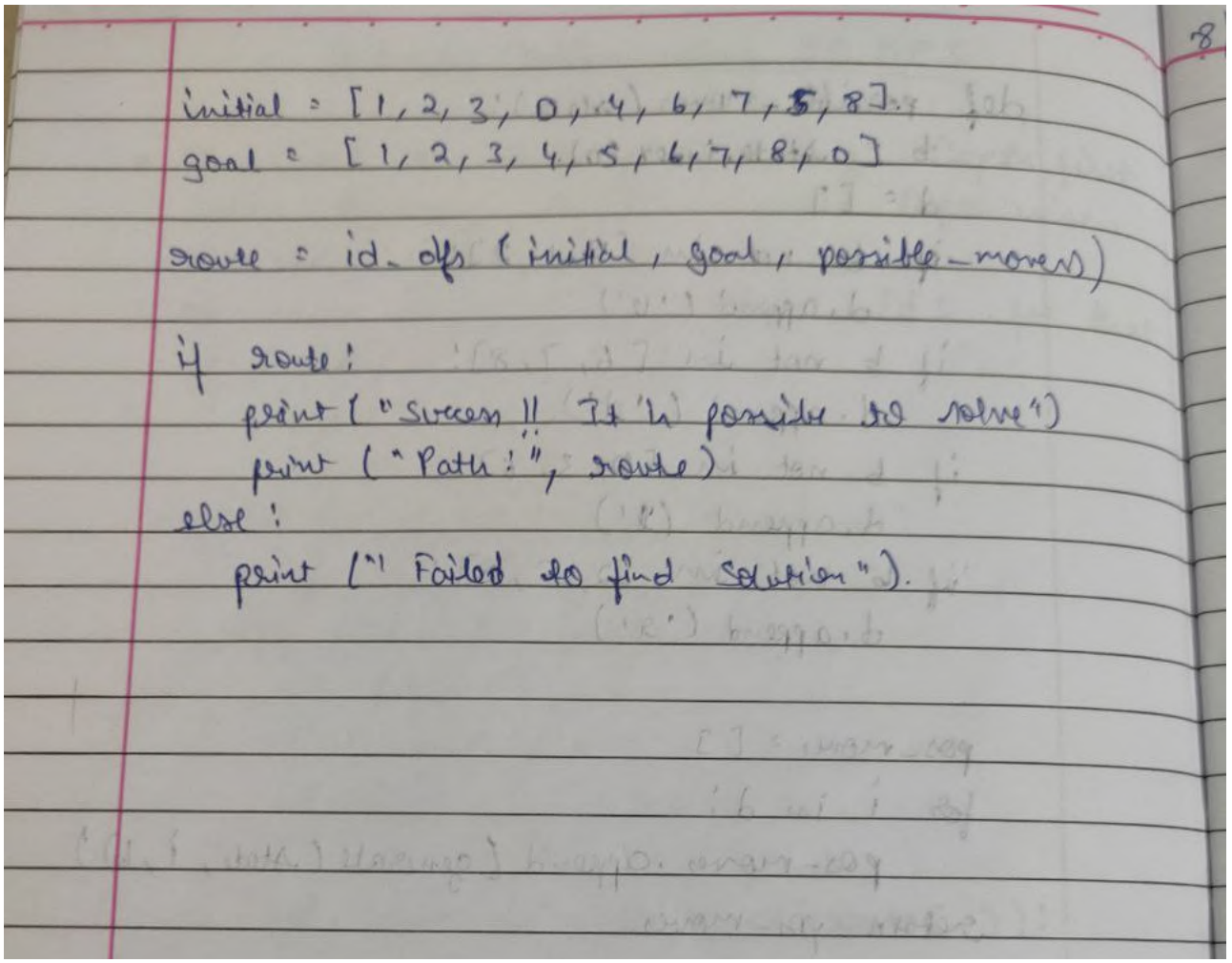
Date / /
Page

8 Puzzle Problem using ID DFS

* The goal of Iterative Deepening Depth first search algorithm is to iterate depth wise, i.e level by level.
* At each level, we apply DFS for that as a limit.

```
def id_dfs(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)
    d = []
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')

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

def generate(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
    return temp.
```

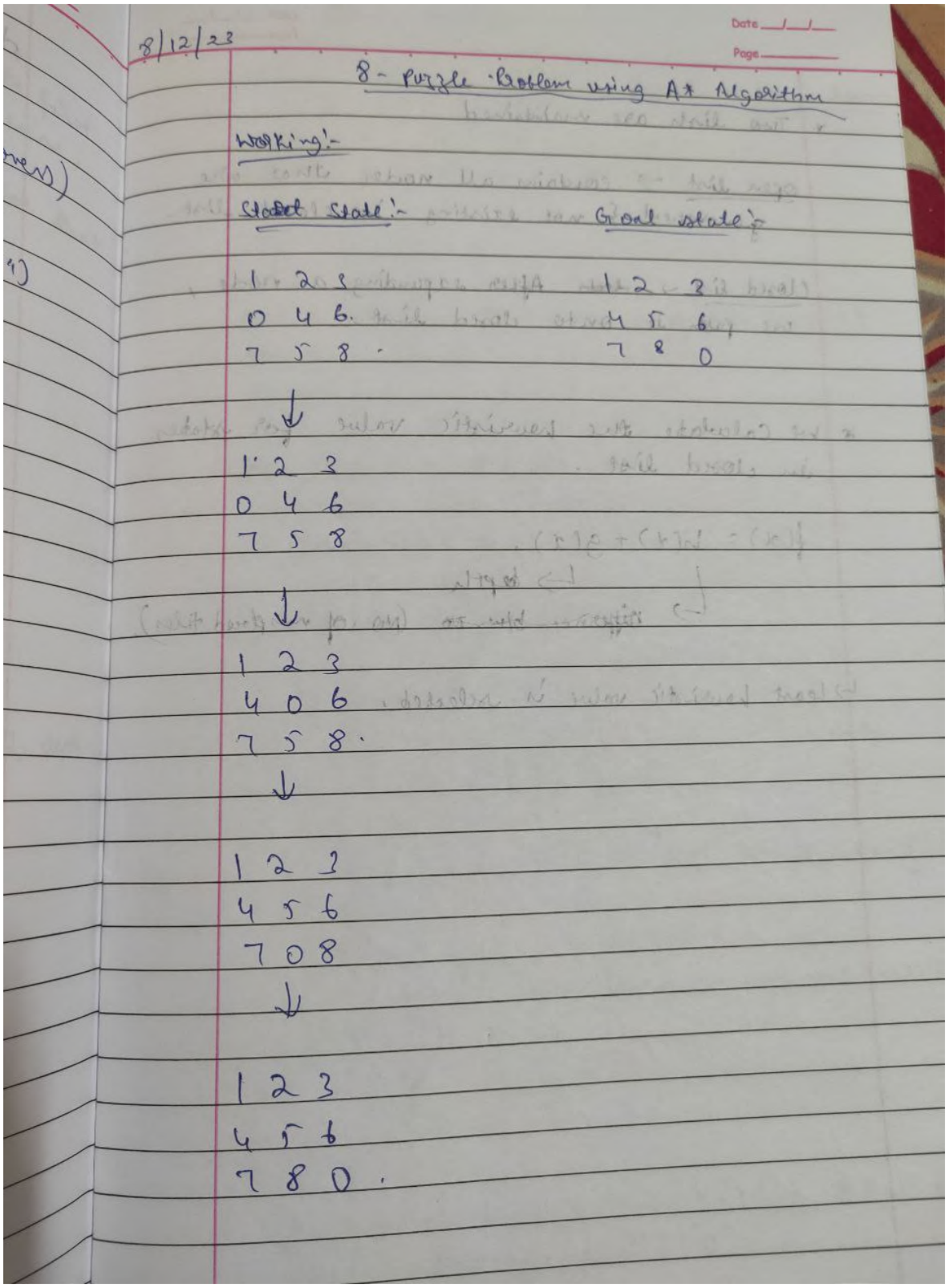


Output:

```
Likhith GS-1BM21CS096
Success!! It is possible to solve 8 Puzzle problem
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 4 : 8 Puzzle A* Search Algorithm

Observation :



* Two lists are maintained

Open list → contains all nodes that are generated & not existing in closed list.

Closed list → After expanding a node, we push it onto closed list.

* We calculate the heuristic value for states in closed list.

$f(x) = h(x) + g(x)$.

↳ depth

↳ ~~Difference b/w no~~ (no. of misplaced tiles).

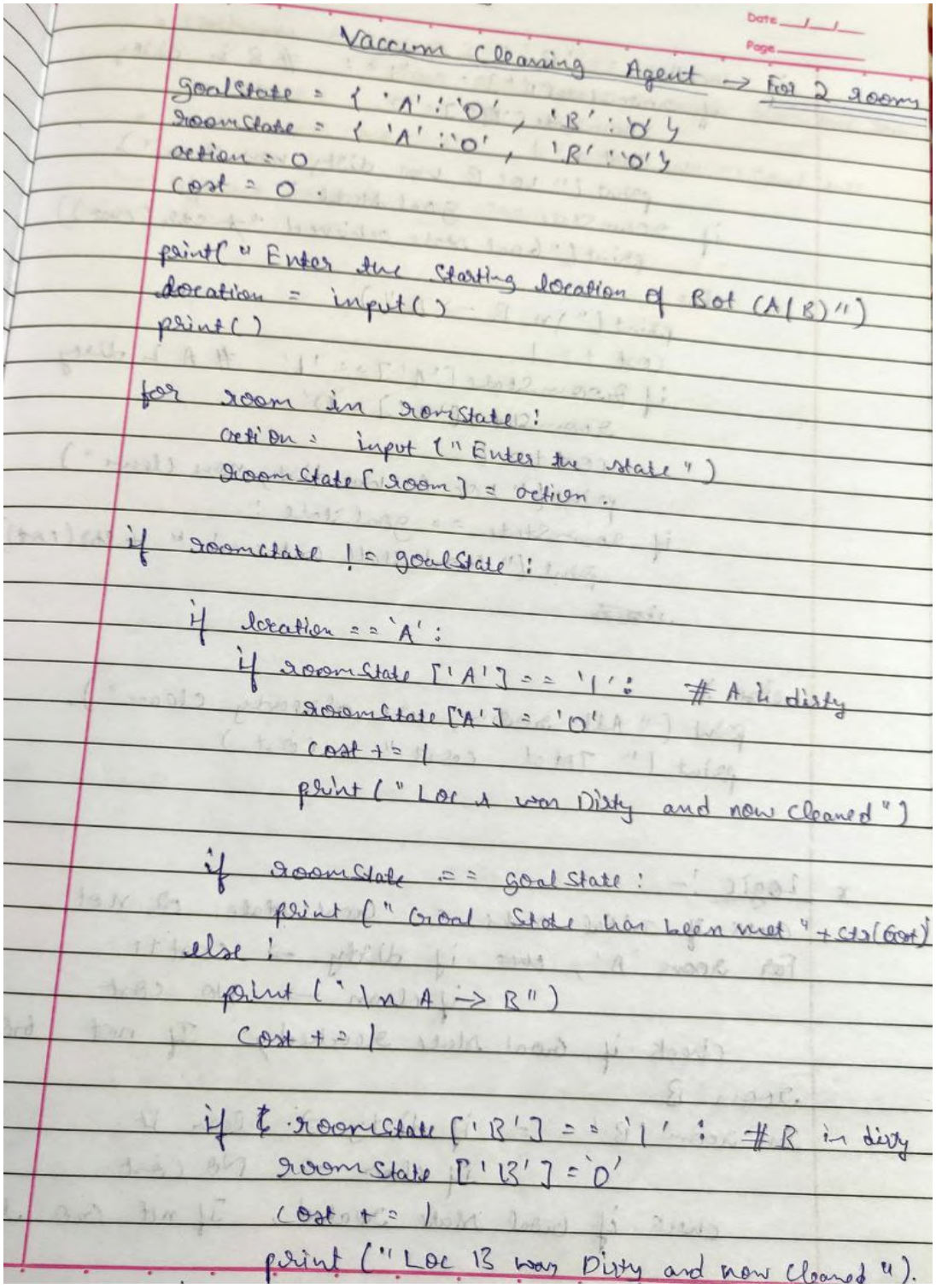
↳ Least heuristic value is selected.

Output :

```
Likhith GS 1BM21CS096
Enter the start state matrix

1 2 3
4 5 6
_ 7 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _




  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

1 2 3
4 5 6
7 8 _
```

Program 5 : Vacuum Cleaner

Observation:



Vaccum Cleaning Agent → For 2 rooms

```
goalState = {'A':'0', 'B':'0'}
roomState = {'A':'0', 'B':'0'}
action = 0
cost = 0.

print("Enter the starting location of Bot (A|B)")
location = input()
print()

for room in roomState:
    action = input("Enter the state")
    roomState[room] = action.

if roomstate != goalState:

    if location == 'A':
        if roomState['A'] == '1':  # A is dirty
            roomState['A'] = '0'
            cost += 1
            print("Loc A was Dirty and now cleaned")

        if roomState == goalState:
            print("Goal State has been met" + str(cost))
    else:
        print("\n A -> B")
        cost += 1

    if roomState['B'] == '1':  # B is dirty
        roomState['B'] = '0'
        cost += 1
        print("Loc B was Dirty and now cleaned").
```

```
if location == 'B':
    if roomState['B'] == '1':  # B is dirty
        roomState['B'] = '0'
        cost += 1
        print("Loc B was dirty now clean")
    if roomState == goalState:
        print("Goal state achieved " + str(cost))
else:
    print("\n B -> A")
    cost += 1
    if roomState['A'] == '1':  # A is dirty
        roomState['A'] = '0'
        cost += 1
        print("Loc A was dirty now clean")
    if roomState == goalState:
        print("Goal State achieved " + str(cost))

else:
    print("All rooms are already clean")
    print("Total cost " + (cost)
```

* Logic :-

Check if Initial state == Goal state or not.

For room 'A', if dirty → cost++

if clean → NO cost

Check if Goal State reached, If not GO to room B.

In room 'B' :- if dirty → clean it.

if clean → NO cost.

check if Goal State reached, If not GO to room A

Vaccum Cleaning Agents → 4 rooms

```
goalState = {'A': '0', 'B': '0', 'C': '0', 'D': '0'}
roomState = {'A': '0', 'B': '0', 'C': '0', 'D': '0'}
cost = 0.

print("Enter starting location of Bot ('A'/'B'/'C'/'D'):")
location = input()
print()

for room in roomState:
    roomState[room] = input("Enter State")

while roomState != goalState:
    if location in roomState:
        curr_room = location.
        if roomState[curr_room] == '1':
            cost += 1.

    next_room = input("Enter")

    if next_room in roomState and next_room != curr_room:
        cost += 1.
        location = next_room

print("Cost:", cost)

print("Goal state has been Achieved\n").
print("Performance Measure:", cost)
```

Output:

```
Likhith GS-1BM21CS096
Enter clean status for Room 1 (1 for dirty, 0 for clean): 1
Enter clean status for Room 2 (1 for dirty, 0 for clean): 0
Cleaning Room 1 (Room was dirty)
Room 1 is now clean.
Room 2 is already clean.
Returning to Room 1 to check if it has become dirty again:
Room 1 is already clean.
Room 1 is clean after checking.
```

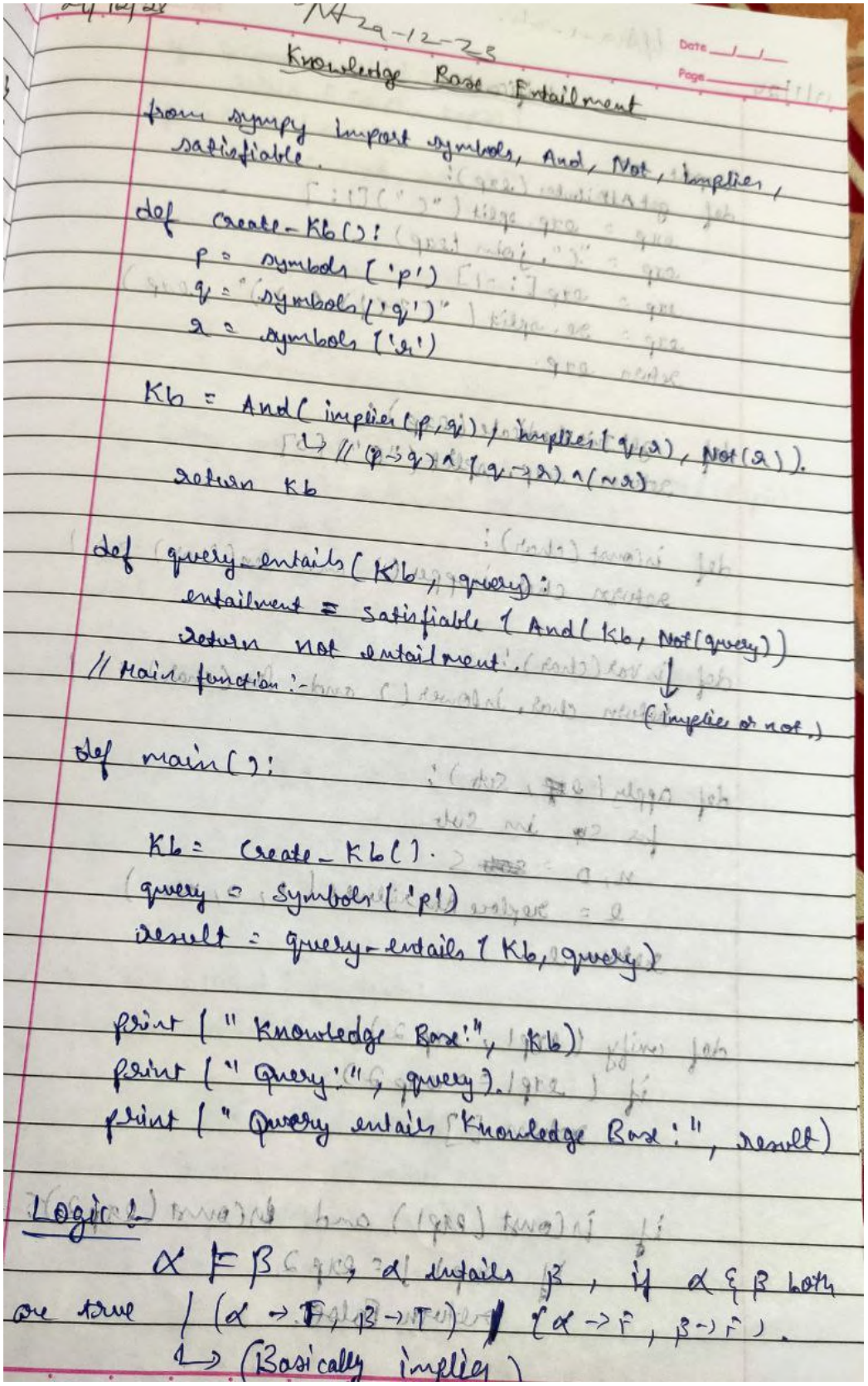
Vacuum cleaner 2 rooms

```
Likhith GS-1BM21CS096
Enter clean status for Room at (1, 1) (1 for dirty, 0 for clean): 1
Enter clean status for Room at (1, 2) (1 for dirty, 0 for clean): 0
Enter clean status for Room at (2, 1) (1 for dirty, 0 for clean): 1
Enter clean status for Room at (2, 2) (1 for dirty, 0 for clean): 1
Cleaning Room at (1, 1) (Room was dirty)
Room is now clean.
Room at (1, 2) is already clean.
Cleaning Room at (2, 1) (Room was dirty)
Room is now clean.
Cleaning Room at (2, 2) (Room was dirty)
Room is now clean.
Returning to Room at (1, 1) to check if it has become dirty again:
Room at (1, 1) is already clean.
```

Vacuum cleaner 4 rooms

Program 6 : Knowledge Base Entailment

Observation:



Knowledge Base Entailment

```
from sympy import symbols, And, Not, Implies,
satisfiable

def create_kb():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')

    kb = And(Implies(p,q), Implies(q,r), Not(r))
    return kb

def query_entails(kb, query):
    entailment = satisfiable(And(kb, Not(query)))
    return not entailment

# Main function:-

def main():

    kb = create_kb()
    query = symbols('p')
    result = query_entails(kb, query)

    print("Knowledge Base:", kb)
    print("Query:", query)
    print("Query entails Knowledge Base:", result)
```

Logic :-

$\alpha \models \beta$ entails β, if α & β both are true / $(\alpha \to T, \beta \to T)$ / $(\alpha \to F, \beta \to F)$.

↳ (Basically implies)

Output:

```
Likhith GS 1BM21CS096
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

Program 7 : Knowledge Base Resolution

Observation:

29/12/22

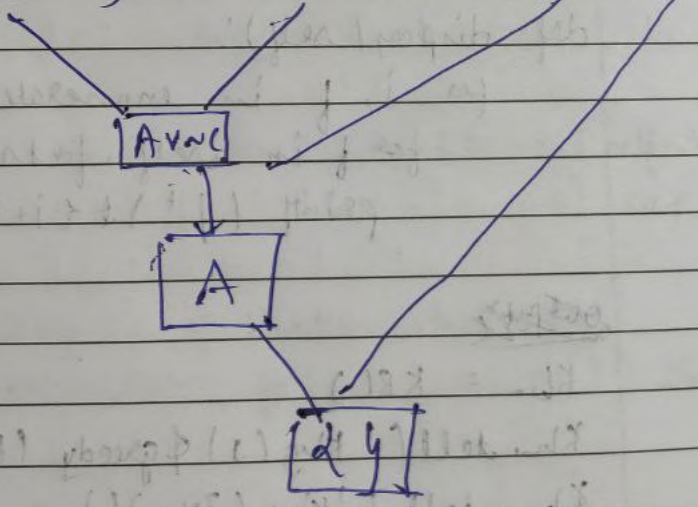
Date / /
Page

Knowledge Based Resolution

```
def negate_literal(literal):
    if literal[0] == 'N':
        return literal[1:]
    else:
        return '~' + literal

def resolve(l1, l2):
    resolve_clause = set(l1) | set(l2)
    for literal in l1:
        if negate_literal(literal) in l2:
            resolve_clause.remove(literal)

    return tuple(resolved_clause)
```

Ex:-

KB :- $(A \vee \neg B) \wedge (B \vee \neg C) \wedge C \wedge \neg A$



Output:

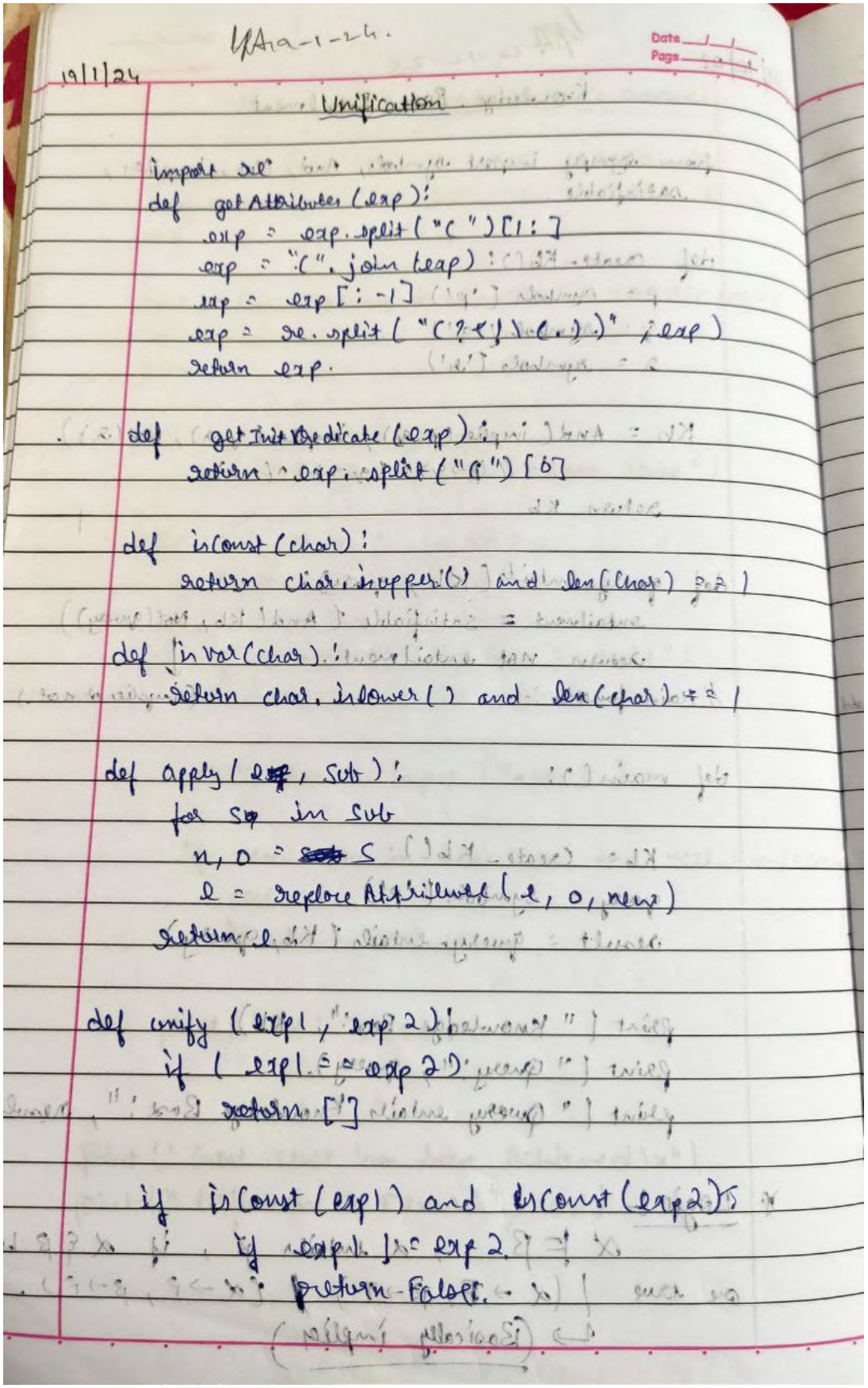
```
Likhith GS 1BM21CS096

Step    |Clause |Derivation
------------------------------
 1.     | Rv~P  | Given.
 2.     | Rv~Q  | Given.
 3.     | ~RvP  | Given.
 4.     | ~RvQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved Rv~P and ~RvP to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

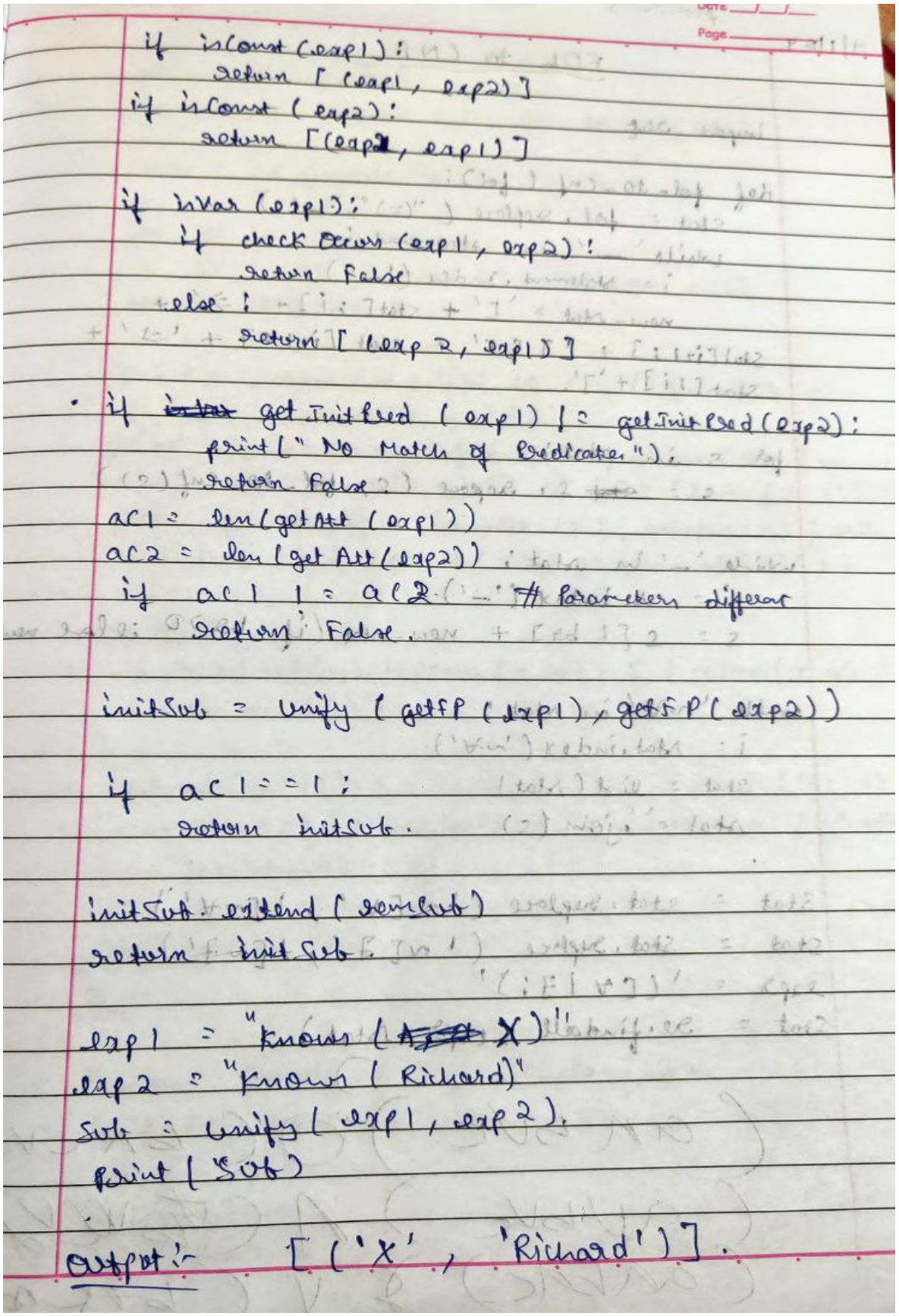
```
Likhith GS 1BM21CS096

Step    |Clause |Derivation
------------------------------
 1.     | PvQ   | Given.
 2.     | ~PvR  | Given.
 3.     | ~QvR  | Given.
 4.     | ~R    | Negated conclusion.
 5.     | QvR   | Resolved from PvQ and ~PvR.
 6.     | PvR   | Resolved from PvQ and ~QvR.
 7.     | ~P    | Resolved from ~PvR and ~R.
 8.     | ~Q    | Resolved from ~QvR and ~R.
 9.     | Q     | Resolved from ~R and QvR.
 10.    | P     | Resolved from ~R and PvR.
 11.    | R     | Resolved from QvR and ~Q.
 12.    |       | Resolved R and ~R to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

```
Likhith GS 1BM21CS096

Step    |Clause |Derivation
---------------------------
 1.     | PvQ   | Given.
 2.     | PvR   | Given.
 3.     | ~PvR  | Given.
 4.     | RvS   | Given.
 5.     | Rv~Q  | Given.
 6.     | ~Sv~Q | Given.
 7.     | ~R    | Negated conclusion.
 8.     | QvR   | Resolved from PvQ and ~PvR.
 9.     | Pv~S  | Resolved from PvQ and ~Sv~Q.
 10.    | P     | Resolved from PvR and ~R.
 11.    | ~P    | Resolved from ~PvR and ~R.
 12.    | Rv~S  | Resolved from ~PvR and Pv~S.
 13.    | R     | Resolved from ~PvR and P.
 14.    | S     | Resolved from RvS and ~R.
 15.    | ~Q    | Resolved from Rv~Q and ~R.
 16.    | Q     | Resolved from ~R and QvR.
 17.    | ~S    | Resolved from ~R and Rv~S.
 18.    |       | Resolved ~R and R to ~RvR, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

Program 8 : Unification

Observation :





```
if isConst(exp1):
    return [(exp1, exp2)]
if isConst(exp2):
    return [(exp2, exp1)]

if isVar(exp1):
    if checkOccurs(exp1, exp2):
        return False
    else:
        return [(exp2, exp1)]

• if getInitPred(exp1) != getInitPred(exp2):
        print("No Match of Predicates")
        return False
ac1 = len(getAtt(exp1))
ac2 = len(getAtt(exp2))
if ac1 != ac2: # Parameters differ
    return False

initSub = unify(getFP(exp1), getFP(exp2))

if ac1 == 1:
    return initSub.

initSub.extend(remSub)
return initSub

exp1 = "Knows(X)"
exp2 = "Knows(Richard)"
Sub = unify(exp1, exp2).
print(Sub)

Output :-  [('X', 'Richard')].
```

Output:

```
[5] exp1 = "knows(X)"
    exp2 = "knows(Richard)"
    substitutions = unify(exp1, exp2)
    print('Likhith GS 1BM21CS096')
    print("Substitutions:")
    print(substitutions)
```

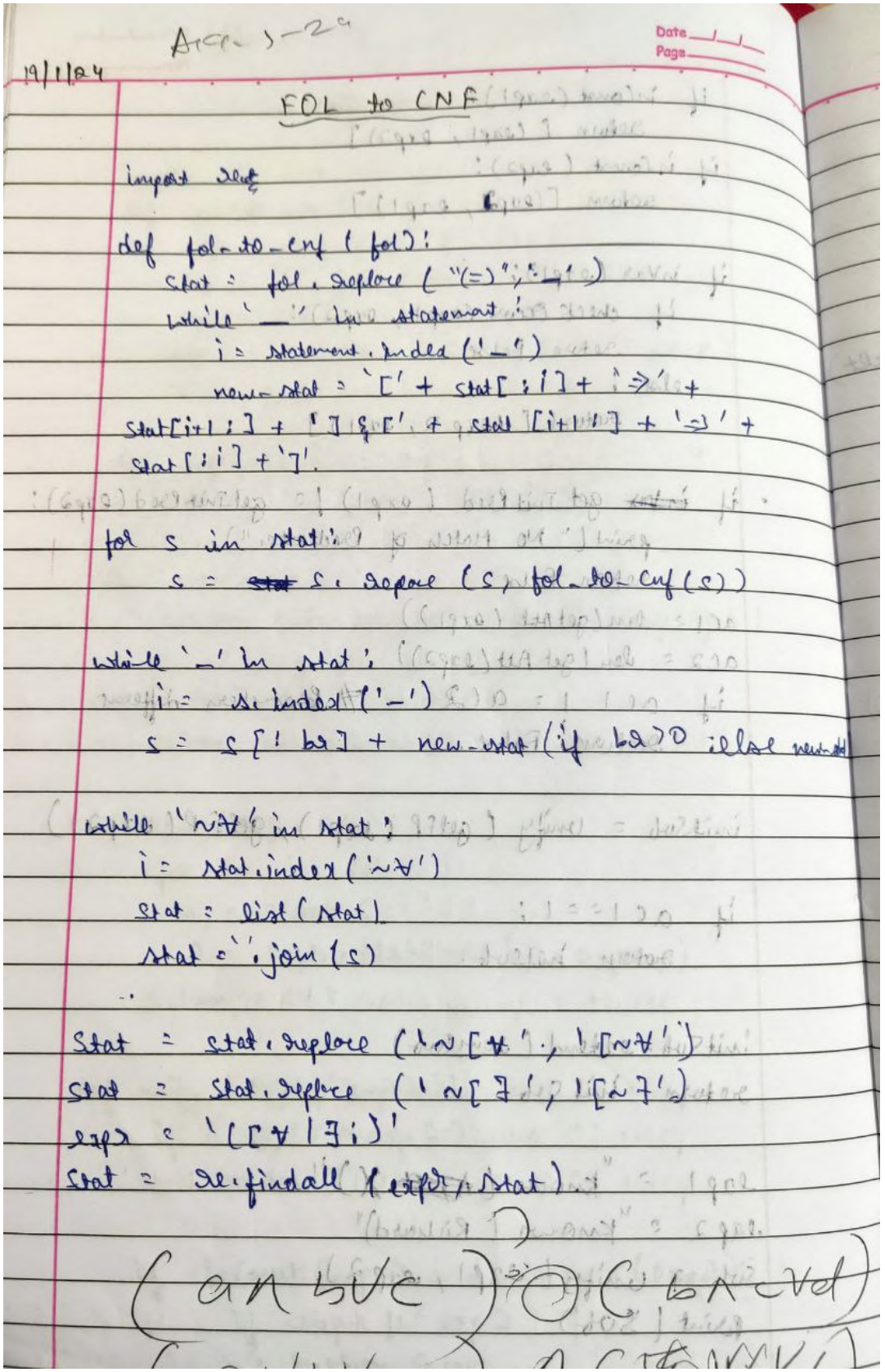
```
Likhith GS 1BM21CS096
Substitutions:
[('X', 'Richard')]
```

```
exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
substitutions = unify(exp1, exp2)
print('Likhith GS 1BM21CS096')
print("Substitutions:")
print(substitutions)
```

```
Likhith GS 1BM21CS096
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

Program 9 : FOL to CNF

Observation:



FOL to CNF

```
import re

def fol_to_cnf ( fol):
    stat = fol.replace ("<=>", "_")
    while '_' in statement:
        i = statement.index ('_')
        new_stat = '[' + stat[ :i] + '=>' +
    stat[i+1 :] + ']&[' + stat [i+1 :] + '=>' +
    stat [ :i] + ']'.

    for s in stat:
        s = stat s.replace (s, fol_to_cnf(s))

while '_' in stat:
    i = s.index('_')
    s = s[ : i] + new_stat (if i>0 :else new

while '~∀' in stat:
    i = stat.index('~∀')
    stat = list (stat)
    stat = ''.join (s)

stat = stat.replace ('~[∀', '[~∀')
stat = stat.replace ('~[∃', '[~∃')
expr = '([∀|∃];)'
stat = re.findall (expr, stat)
```

(a∧b∨c) ⊙ (b∧cvd)

```
for s in stat:
    sl = sl.replace(s, fol_to_cnf(s))
stats = re.findall(pepr, stat).
for s in statements:
    sl = sl.replace(s, DeMorgan(s))
return sl.
```

```
print(skolemization(fol_to_cnf("animal(y)<=>
        loves(x,y)")))
print(skolemization(fol_to_cnf("∀x[∀y[animal
(y) => loves(x,y)]] => [∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x) & weapon(y) & sells
(x,y,z) & hostile(z)] => criminal(x)"))
```

Output :-

- [~animal(y) | loves(x,y)] & [~loves(x,y) | animal(y)]
- [animal(G(x)) & ~loves(x, G(x))] | [loves(F(x), x)]
- [~american(x) | ~weapon(y) | ~sells(x,y,z) | ~hostile(z)] | criminal(x)

Logic :-

Steps :-

- Remove Quantifiers by Scholamization
- Split the expression when => is found & solve it.

Apply Demorgan.

Solve until only

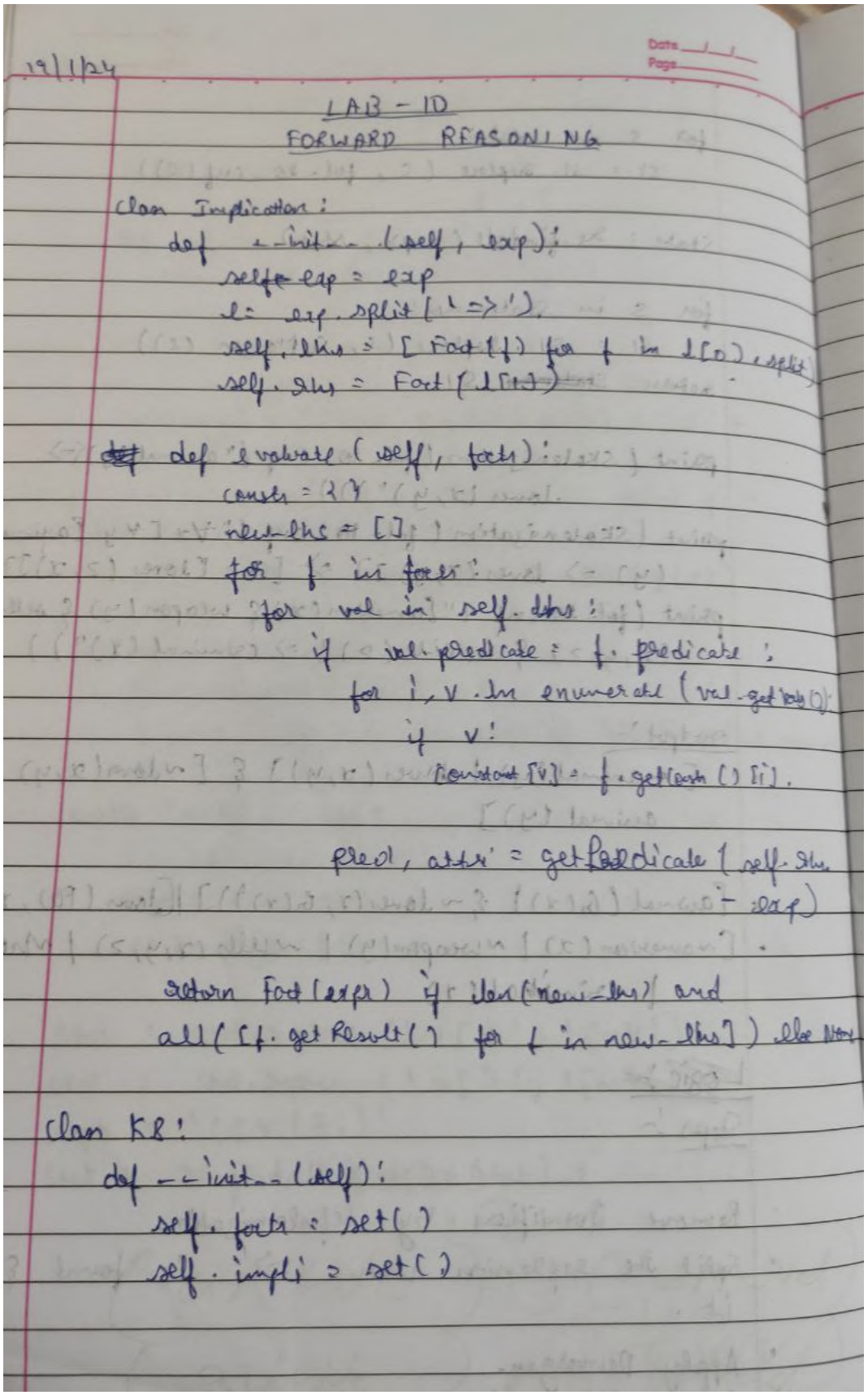
Output:

```
print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

```
Likhith GS 1BM21CS096
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

Program 10 : Forward Reasoning

Observation:



19/1/24

LAB - 10

FORWARD REASONING

```
class Implication:
    def __init__(self, exp):
        self.exp = exp
        l = exp.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for f in facts:
            for val in self.lhs:
                if val.predicate == f.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = f.getConstants()[i]

        pred, attr = getPredicate(self.rhs.exp)

        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.impli = set()
```

```
def tell(self, e):
    if '=>' in e:
        self.implication.add(Implication(e))
    else:
        self.facts.add(Fact(e))
    for i in self.implication:
        res = i.evaluate(self.facts)
        if res:
            self.facts.add(res)

def query(self, e):
    f = set([f.e for f in self.facts])
    i = 1
    for f in facts:
        if Fact(f).pred == Fact(e).predicate:
            i += 1

def display(self):
    for i, f in enumerate(set([f.e
        for f in self.facts])):
        print(f'{i+1}. {f}')
```

```
kb = KB()
kb.tell('King(x) & greedy(x) => evil(x)')
kb.tell('King(John)')
kb.tell('greedy(John)')
kb.tell('King(Richard)')
kb.query('evil(x)')
```

Output:

```
Querying evil(x):
    1. evil(John)
```

Output:

```
Likhith GS 1BM21CS096
Querying criminal(x):
        1. criminal(West)
All facts:
        1. criminal(West)
        2. enemy(Nono,America)
        3. owns(Nono,M1)
        4. missile(M1)
        5. weapon(M1)
        6. hostile(Nono)
        7. sells(West,M1,Nono)
        8. american(West)
```

```
[4] kb_ = KB()
    kb_.tell('king(x)&greedy(x)=>evil(x)')
    kb_.tell('king(John)')
    kb_.tell('greedy(John)')
    kb_.tell('king(Richard)')
    kb_.query('evil(x)')
```

Focus the last run cell

02:05 (0 minutes ago)
executed in 0.008 s

```
Querying evil(x):
        1. evil(Richard)
        2. evil(John)
```