VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

ARTIFICIAL INTELLIGENCE

Submitted by

PALLAVI MANUBALLA (1BM21CS124)

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
Oct 2023-Feb 2024

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

CERTIFICATE

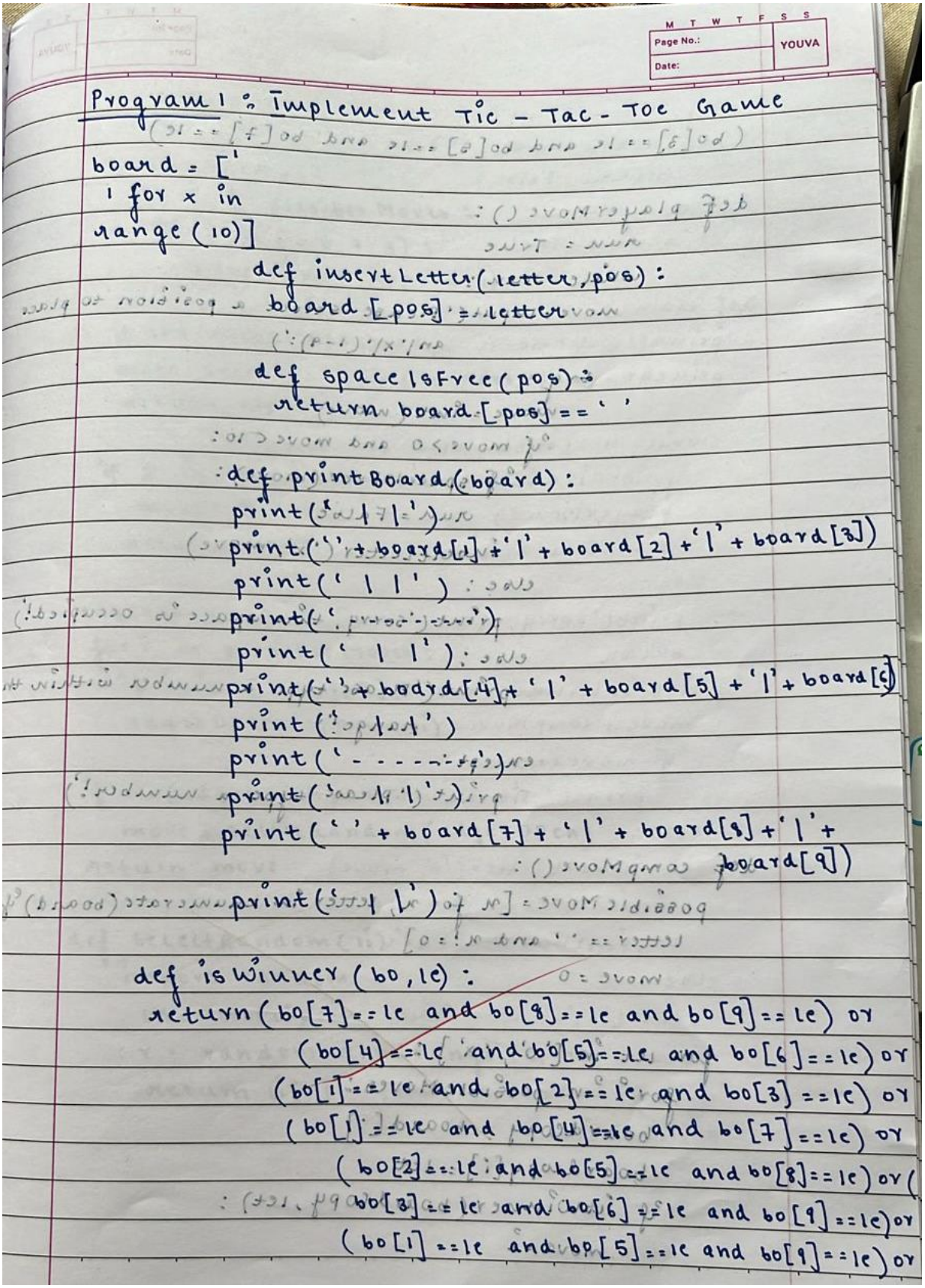
This is to certify that the Lab work entitled **"ARTIFICIAL INTELLIGENCE"** carried out by **PALLAVI MANUBALLA(1BM21CS124)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence Lab - **(22CS5PCAIN)**work prescribed for the said degree.

Saritha A N
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table of Contents

1.Implement Tic –Tac –Toe Game.



Program 1 : 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('-----------')
             print('   |   |')
            print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
             print('   |   |')
             print('-----------')
             print('   |   |')
             print(' ' + board[7] + ' | ' + board[8] + ' | ' +
                                                board[9])

             print('   |   |')

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

```
(bo[3] == le and bo[5] == le and bo[7] == le)

def playerMove():
    run = True
    while run:
        move = input('please select a position to place
                an 'x' (1-9):')
        try:
            move = int(move)
            if move > 0 and move < 10:
                if spaceIsFree(move):
                    run = False
                    insertLetter('x', move)
                else:
                    print('sorry, this space is occupied!')
            else:
                print('please type a number within the
                    range!')
        except:
            print('please type a number!')

def compMove():
    possibleMove = [x for x, letter in enumerate(board) if
        letter == ' ' and x != 0]
    move = 0

    for let in ['o', 'x']:
        for i in possibleMoves:
            boardCopy = board[:]
            boardCopy[i] = let
            if winner(boardCopy, let):
                move = i
```

```
    return move

cornersOpen = []
for i in possibleMoves:
    if i in [1,3,7,9]:
        cornersOpen.append(i)

if len(cornersOpen) > 0:
    move = selectRandom(cornersOpen)
    return move

if 5 in possibleMoves:
    move = 5
    return move

edgesOpen = []
for i in possibleMoves:
    if i in [2,4,6,8]:
        edgesOpen.append(i)

if len(edgesOpen) > 0:
    move = selectRandom(edgesOpen)
return move

def selectRandom(li):
    import random
    r = random.randrange(0, ln)
    return li[r]
```

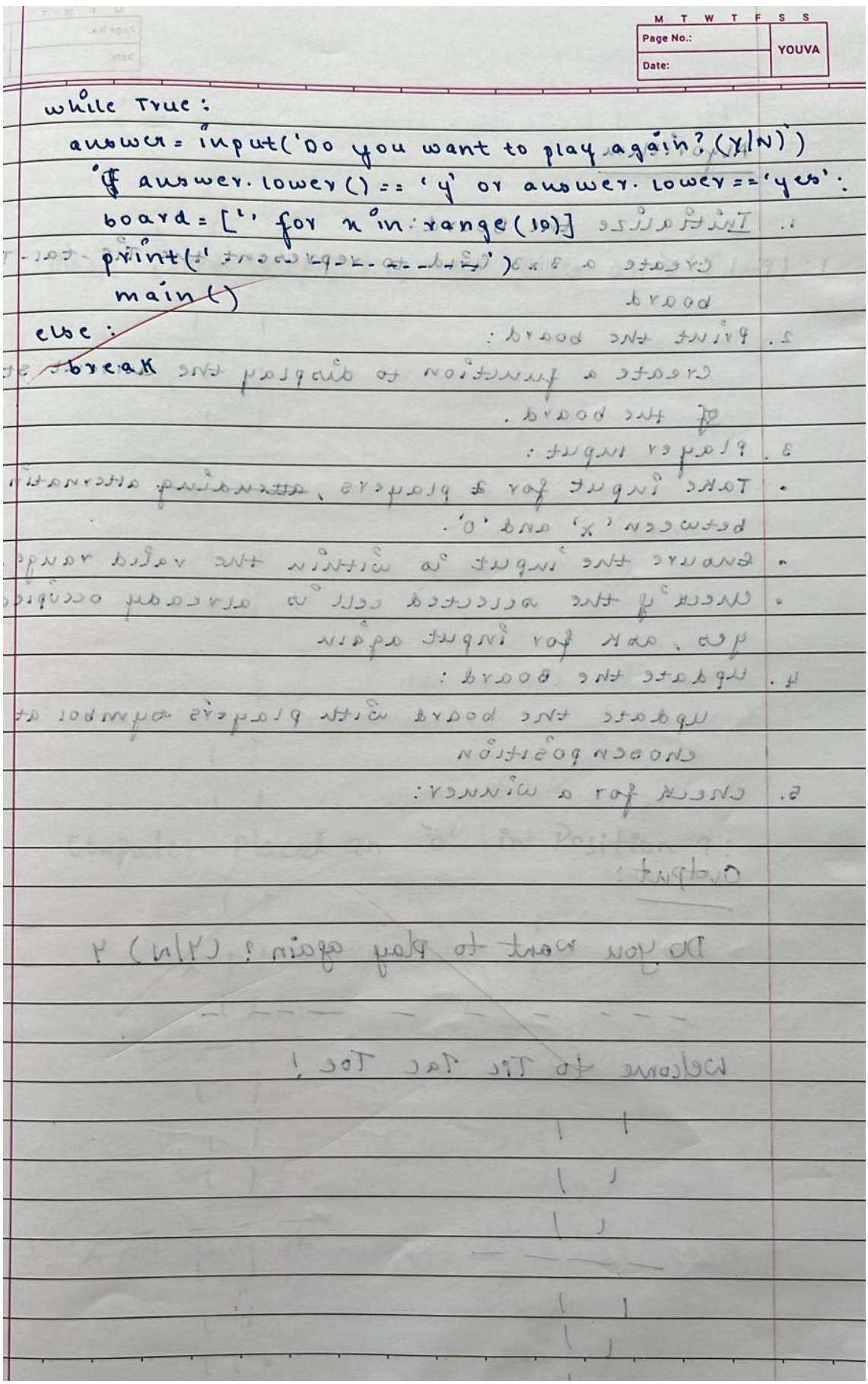
```
def isBoardFull(board):
    if board.count(' ') > 1:
        return False
    else:
        return True

def main():
    print('Welcome to Tic Tac Toe!')
    printBoard(board)

    while not(isBoardFull(board)):
        if not(isWinner(board, 'o')):
            playerMove()
            printBoard(board)
        else:
            print('Sorry, O's won this time!')
            break
    if not(isWinner(board, 'x')):
        move = compMove()
        if move == 0:
            print('Tie Game!')
        else:
            insertLetter('O', move)
            print('Computer placed an 'O' in position, move,:')
            printBoard(board)
    else:
        print('X's won this time! Good Job!')
        break
    if isBoardFull(board):
        print('Tie Game!')
```

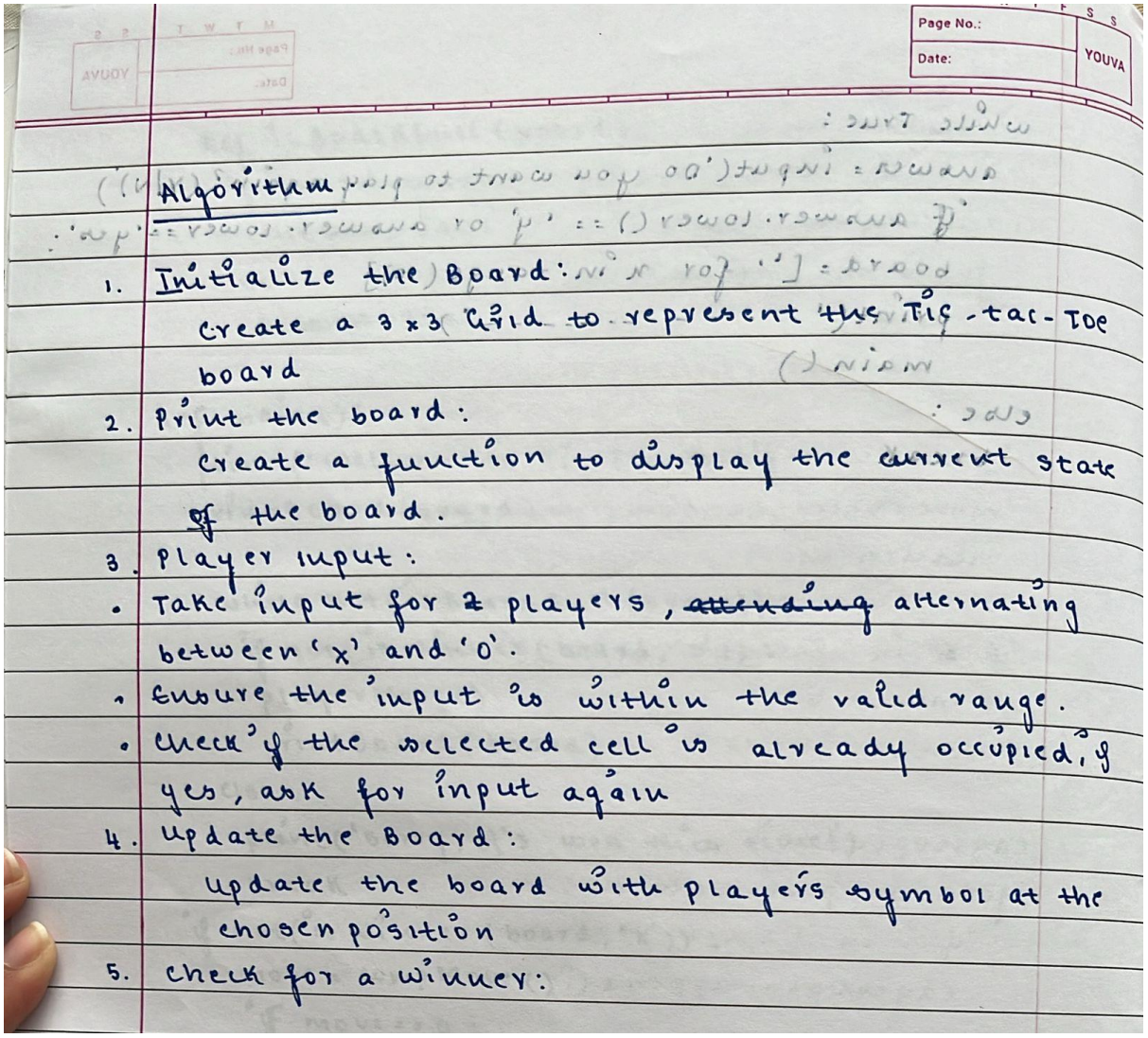
```
while True:
    answer = input('Do you want to play again? (Y/N)')
    if answer.lower() == 'y' or answer.lower == 'yes':
        board = [' ' for x in range(10)]
        print('Welcome to Tic Tac Toe!')
        main()
    else:
        break
```

Output:

Do you want to play again? (Y/N) ?

Welcome to Tic Tac Toe!





Algorithm

1. Initialize the Board:
   Create a 3 x 3 Grid to represent the Tic-tac-Toe board
2. Print the board:
   Create a function to display the current state of the board.
3. Player input:
   - Take input for 2 players, ~~attending~~ alternating between 'x' and 'O'.
   - Ensure the input is within the valid range.
   - Check if the selected cell is already occupied, if yes, ask for input again
4. Update the Board:
   Update the board with players symbol at the chosen position
5. Check for a winner:

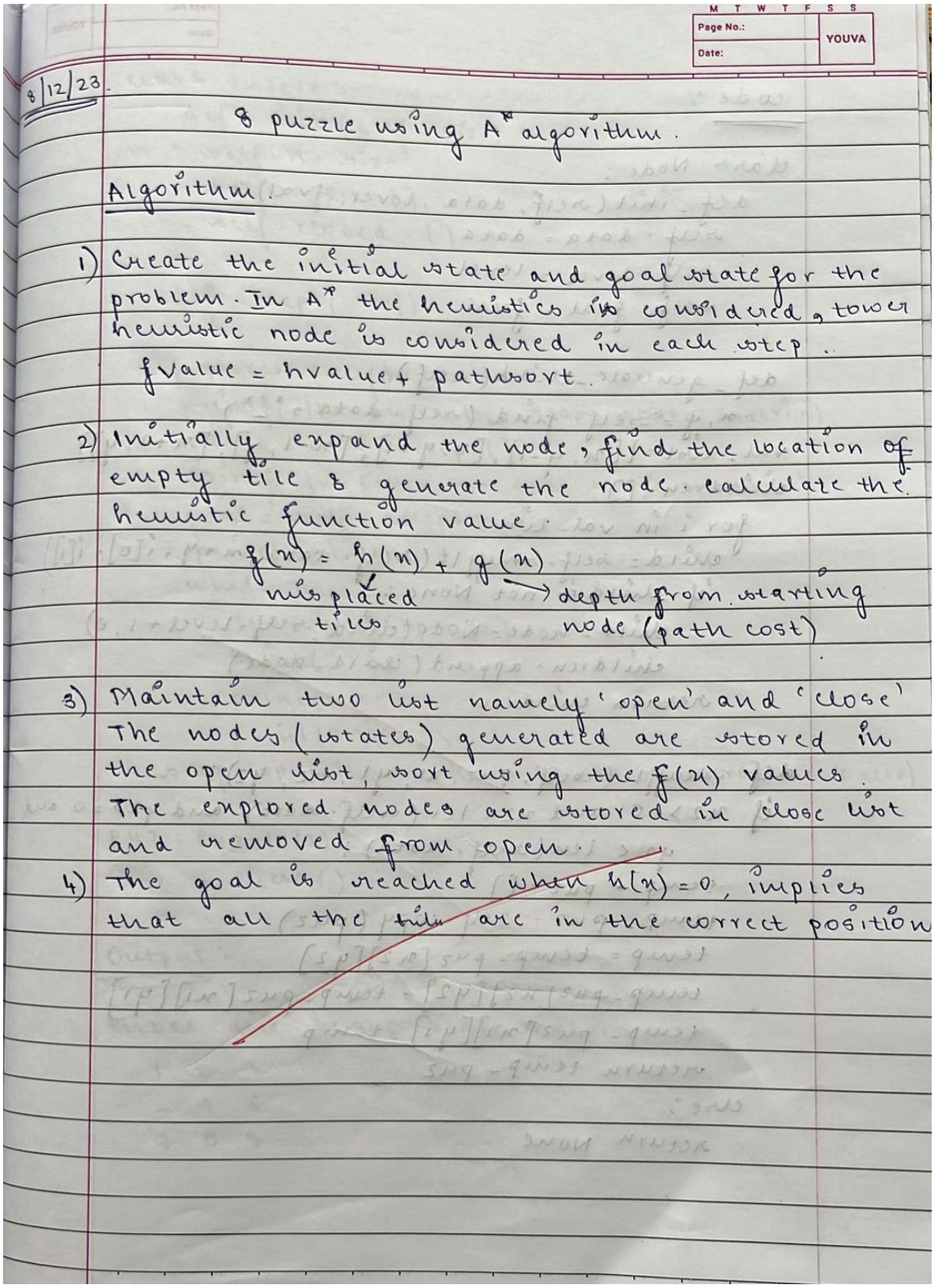
OUTPUT

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
+-------------------------------+
|         |         |         |
|    1    |    2    |    3    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    4    |    5    |    6    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    7    |    8    |    9    |
|         |         |         |
+-------------------------------+
computer's turn :
+-------------------------------+
|         |         |         |
|    1    |    X    |    3    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    4    |    5    |    6    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    7    |    8    |    9    |
|         |         |         |
+-------------------------------+
Your turn :
enter a number on the board :4
```

```
+-------------------------------+
Your turn :
enter a number on the board :4
+-------------------------------+
|         |         |         |
|    1    |    X    |    3    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    0    |    5    |    6    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    7    |    8    |    9    |
|         |         |         |
+-------------------------------+
computer's turn :
+-------------------------------+
|         |         |         |
|    X    |    X    |    3    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    0    |    5    |    6    |
|         |         |         |
+-------------------------------+
|         |         |         |
|    7    |    8    |    9    |
|         |         |         |
+-------------------------------+
Your turn :
enter a number on the board :5
+-------------------------------+
```

```
+-----------------------------+
Your turn :
enter a number on the board :5
+-----------------------------+
|             |             |             |
|      X      |      X      |      3      |
|             |             |             |
+-----------------------------+
|             |             |             |
|      O      |      O      |      6      |
|             |             |             |
+-----------------------------+
|             |             |             |
|      7      |      8      |      9      |
|             |             |             |
+-----------------------------+
computer's turn :
+-----------------------------+
|             |             |             |
|      X      |      X      |      X      |
|             |             |             |
+-----------------------------+
|             |             |             |
|      O      |      O      |      6      |
|             |             |             |
+-----------------------------+
|             |             |             |
|      7      |      8      |      9      |
|             |             |             |
+-----------------------------+
winner is  X
```

2 . Solve 8 puzzle problems.



8/12/23

8 puzzle using A* algorithm.

Algorithm:

1) Create the initial state and goal state for the problem. In A* the heuristics is considered, lower heuristic node is considered in each step.
   fvalue = hvalue + pathcost.

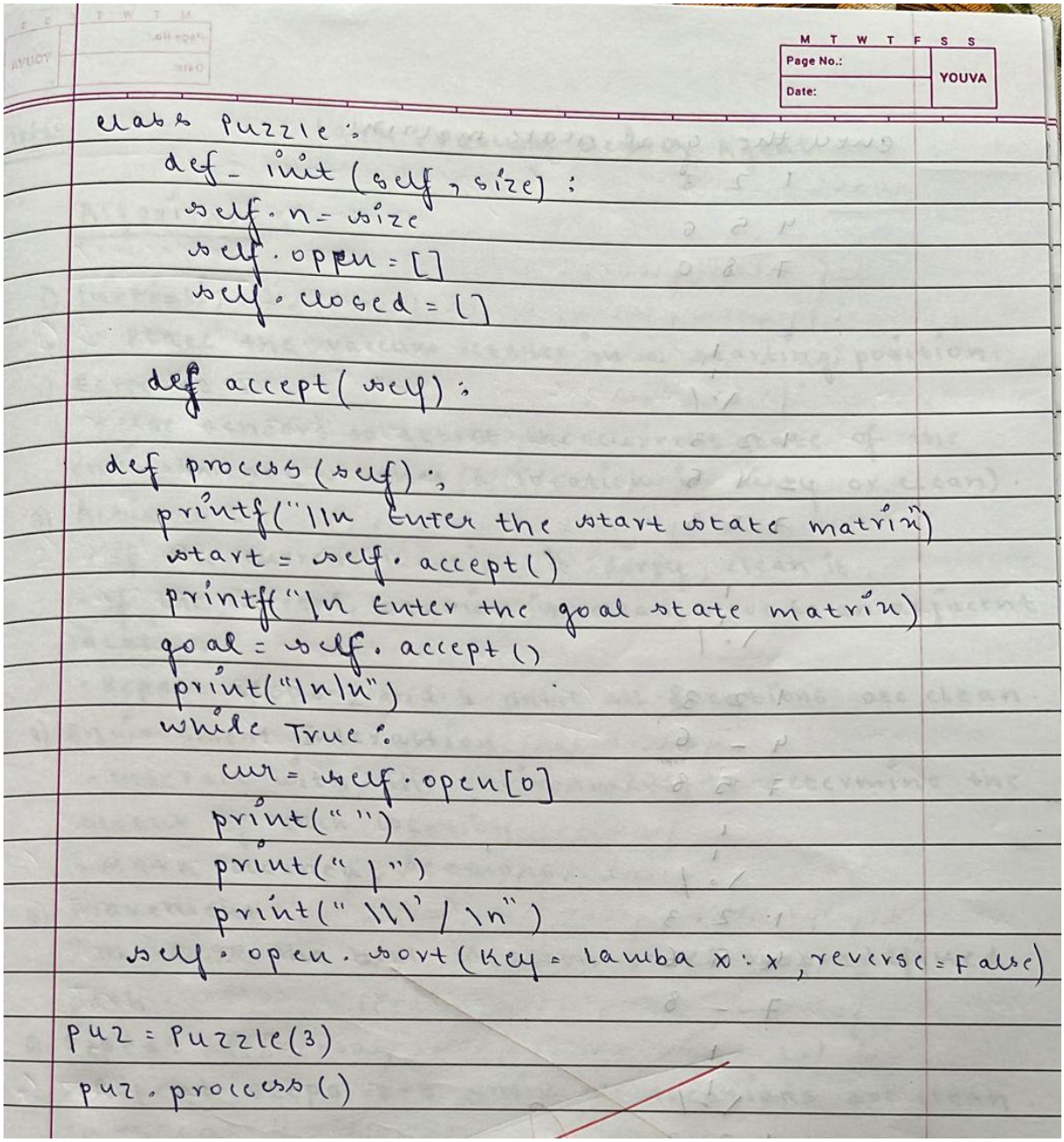
2) Initially expand the node, find the location of empty tile & generate the node. calculate the heuristic function value.
   $f(n) = h(n) + g(n)$
   $h(n)$ → misplaced tiles
   $g(n)$ → depth from starting node (path cost)

3) Maintain two list namely 'open' and 'close' The nodes (states) generated are stored in the open list, sort using the $f(n)$ values. The explored nodes are stored in close list and removed from open.

4) The goal is reached when $h(n) = 0$, implies that all the tile are in the correct position

code :-

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find(self.data, '_')
val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
children = []
    for i in val_list:
        child = self.shuffle(self.data, x, y, i[0], i[1])
        if child is not None:
            child_node = Node(child, self.level+1, 0)
            children.append(child_node)
    return children

def shuffle(self, puz, x1, y1, x2, y2):
if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and
        y2 < len(self.data):
    temp_puz = []
    temp_puz = self.copy(puz)
    temp = temp_puz[x2][y2]
    temp_puz[x2][y2] = temp_puz[x1][y1]
    temp_puz[x1][y1] = temp
    return temp_puz
else:
    return None
```

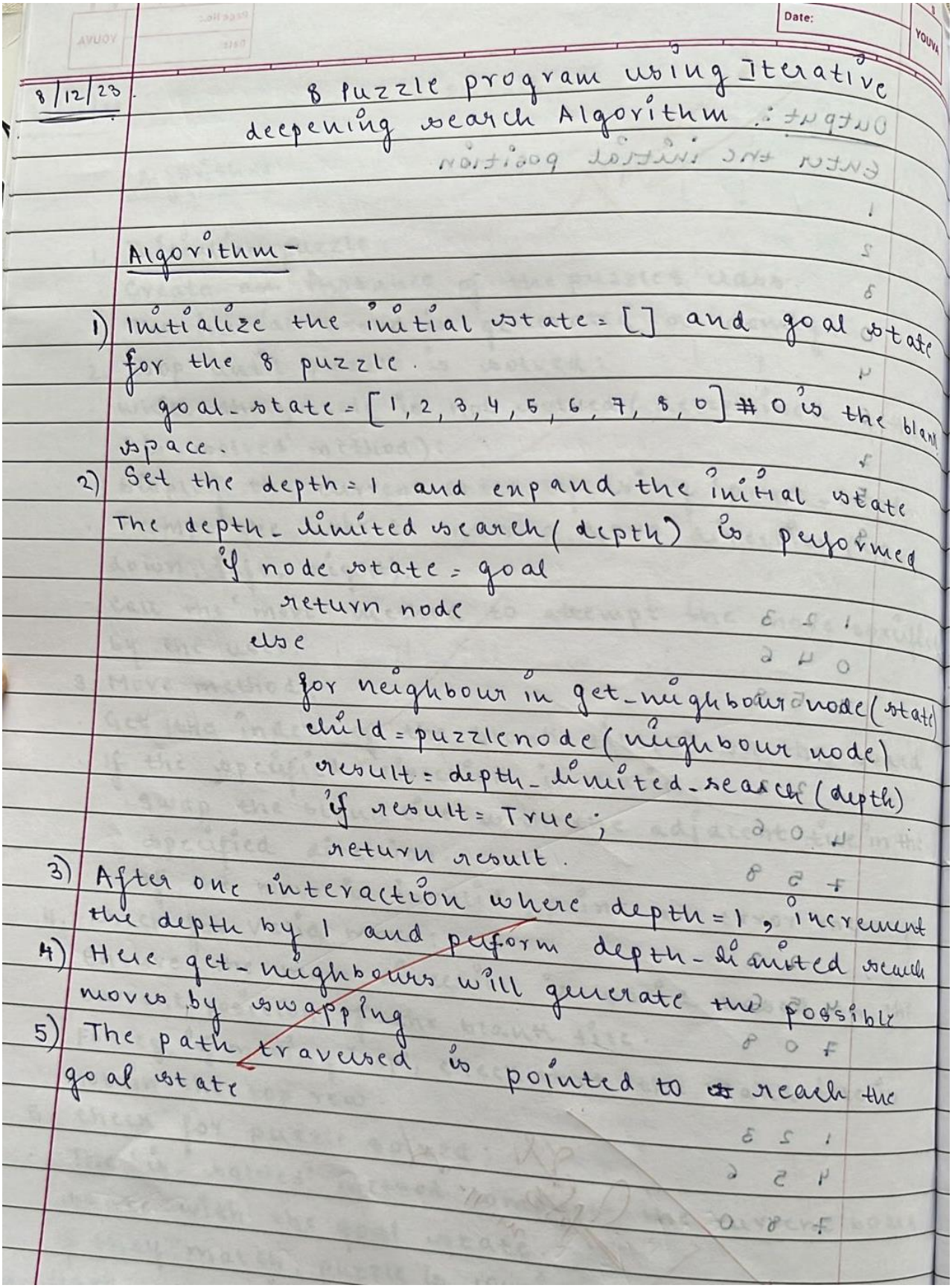


```
class Puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):

def process(self):
    printf("\n Enter the start state matrix")
    start = self.accept()
    printf("\n Enter the goal state matrix")
    goal = self.accept()
    print("\n\n")
    while True:
        cur = self.open[0]
        print(" ")
        print(" |")
        print(" \\\|/ \n")
    self.open.sort(key = lamba x: x, reverse = False)

puz = Puzzle(3)
puz.process()
```

OUTPUT

```
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
```

3. Implement Iterative deepening search algorithm.



8/12/23

8 puzzle program using Iterative deepening search Algorithm

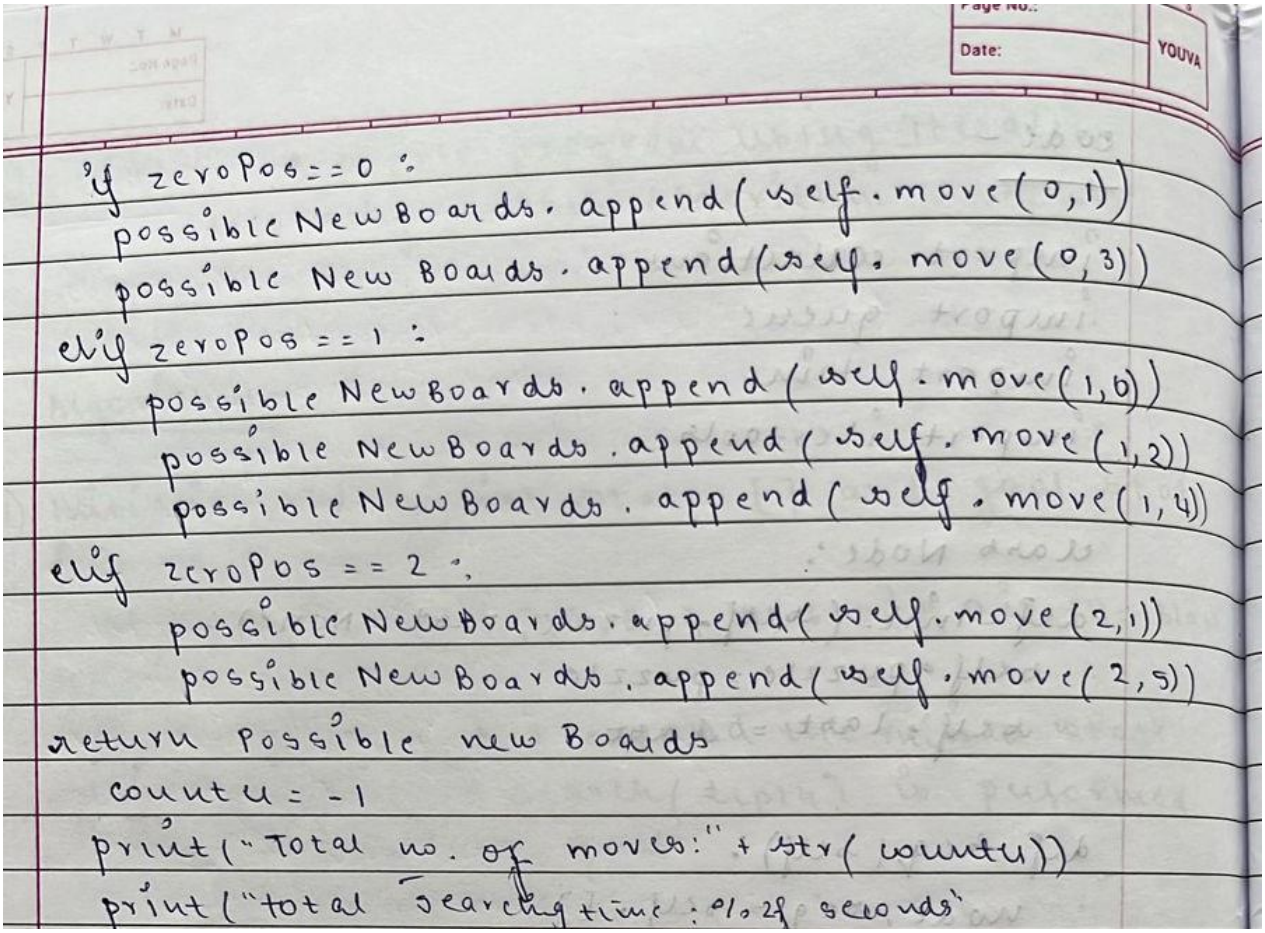
Algorithm -

1) Initialize the initial state = [] and goal state for the 8 puzzle.
   goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0] # 0 is the blank space.
2) Set the depth = 1 and expand the initial state The depth_limited search(depth) is performed
   if node state = goal
   return node
   else
   for neighbour in get_neighbour node(state)
   child = puzzlenode(neighbour node)
   result = depth_limited_search(depth)
   if result = True;
   return result.
3) After one interaction where depth = 1, increment the depth by 1 and perform depth-limited search
4) Here get-neighbours will generate the possible moves by swapping
5) The path traversed is pointed to reach the goal state

Date: YOUVA

code -

```
import collections
import queue
import time
import itertools

class Node:
def __init__(self, puzzle, last = None):
  self.puzzle = puzzle
  self.last = last

def seq(self):
  node, seq = self, []
  while node:
    seq.append(node)
    node = node.last
  yield from reversed(seq)

def state(self):
  return str(self.puzzle.board)

def issolved(self):
  return self.puzzle.issolved

def getmoves(self):
class Puzzle:
  def __init__(self, startBoard):

  def getmoves(self):
```

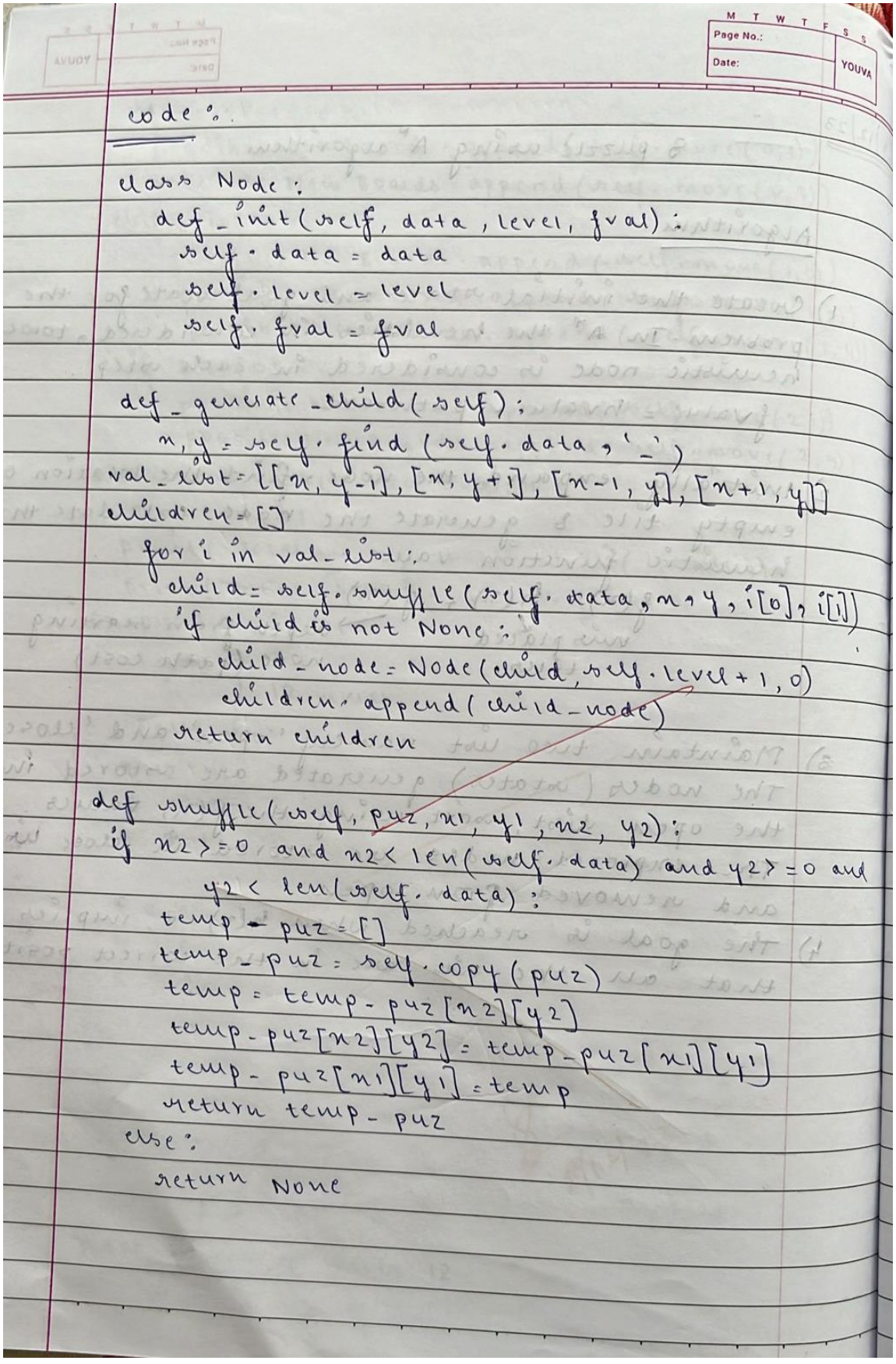


```
if zeroPos == 0 :
    possibleNewBoards.append(self.move(0,1))
    possibleNewBoards.append(self.move(0,3))
elif zeroPos == 1 :
    possibleNewBoards.append(self.move(1,0))
    possibleNewBoards.append(self.move(1,2))
    possibleNewBoards.append(self.move(1,4))
elif zeroPos == 2 :
    possibleNewBoards.append(self.move(2,1))
    possibleNewBoards.append(self.move(2,5))
return possible new Boards
    counter = -1
print("Total no. of moves:" + str(counter))
print("total searching time : %.2f seconds"
```

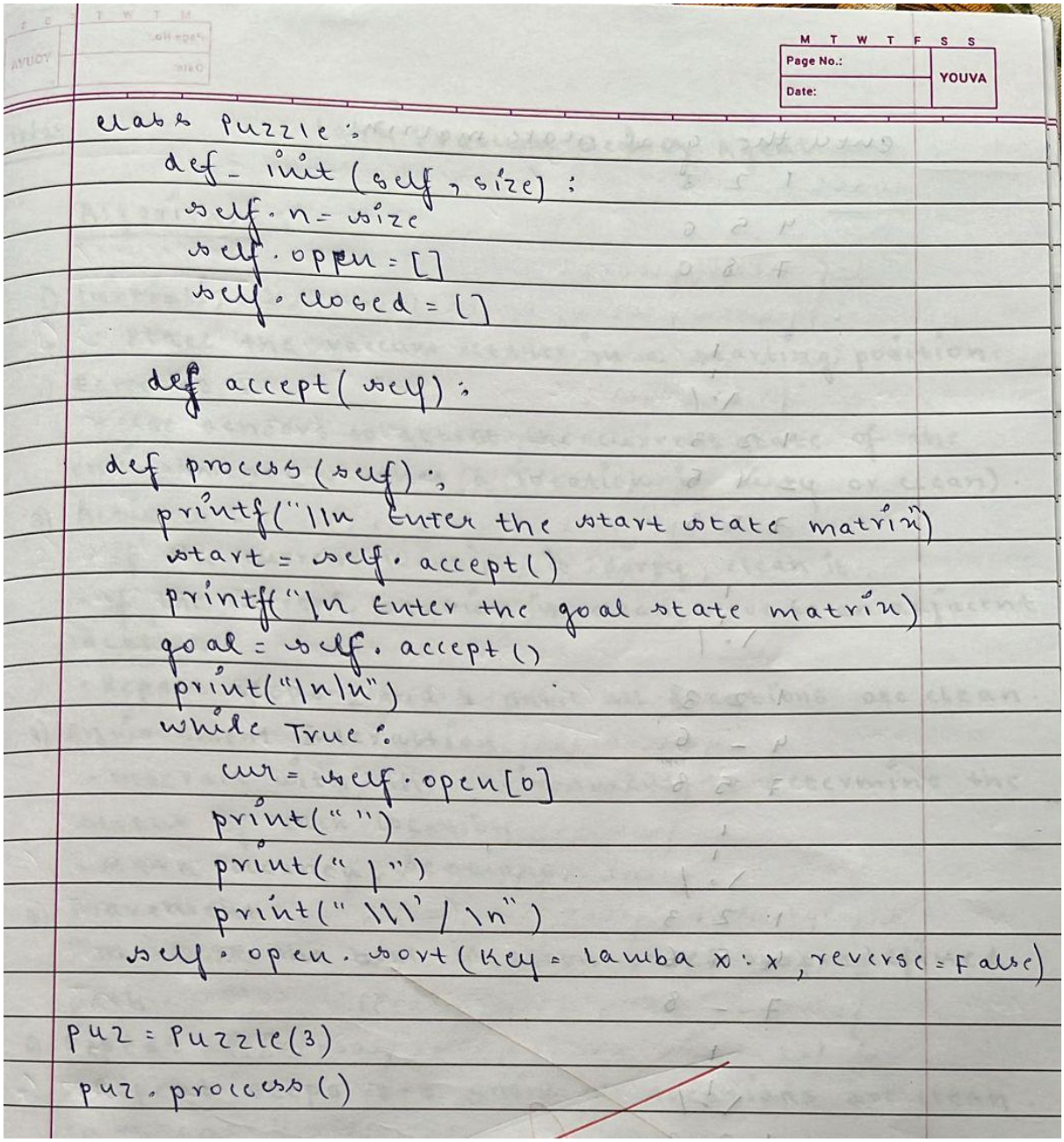
OUTPUT

```
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]]
```

4. Implement A* search algorithm.



```
code :

class Node :
    def __init(self, data , level, fval) :
        self . data = data
        self . level = level
        self . fval = fval

    def _generate_child( self) :
        x, y = self . find (self . data , '_')
val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
children = []
    for i in val_list :
        child = self . shuffle ( self . data , x , y , i[0], i[1])
        if child is not None :
            child_node = Node (child, self . level + 1 , 0)
            children . append ( child_node)
    return children

def shuffle ( self, puz , x1 , y1 , x2 , y2) :
    if x2 >= 0 and x2 < len ( self . data) and y2 >= 0 and
        y2 < len ( self . data) :
        temp _ puz = []
        temp _ puz = self . copy ( puz)
        temp = temp - puz [x2][y2]
        temp - puz [x2][y2] = temp - puz [x1][y1]
        temp - puz [x1][y1] = temp
        return temp - puz
    else :
        return None
```

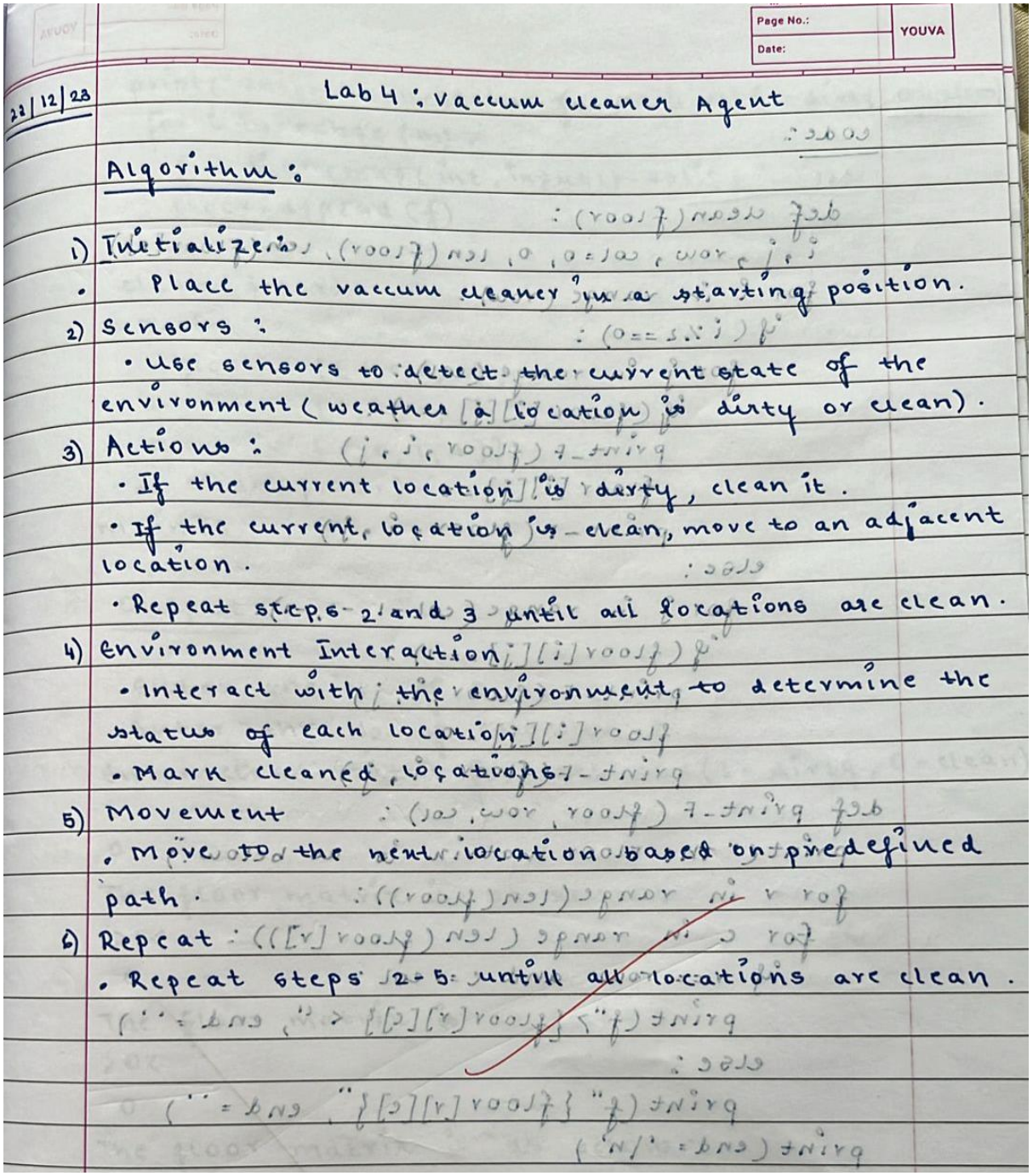


```
class Puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):

def process(self):
    printf("\n Enter the start state matrix")
    start = self.accept()
    printf("\n Enter the goal state matrix")
    goal = self.accept()
    print("\n\n")
    while True:
        cur = self.open[0]
        print(" ")
        print(" |")
        print(" \\\|/ \n")
    self.open.sort(key = lamba x: x, reverse = False)

puz = Puzzle(3)
puz.process()
```

OUTPUT

```
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 _
```

5. Implement vaccum cleaner agent.



22/12/23

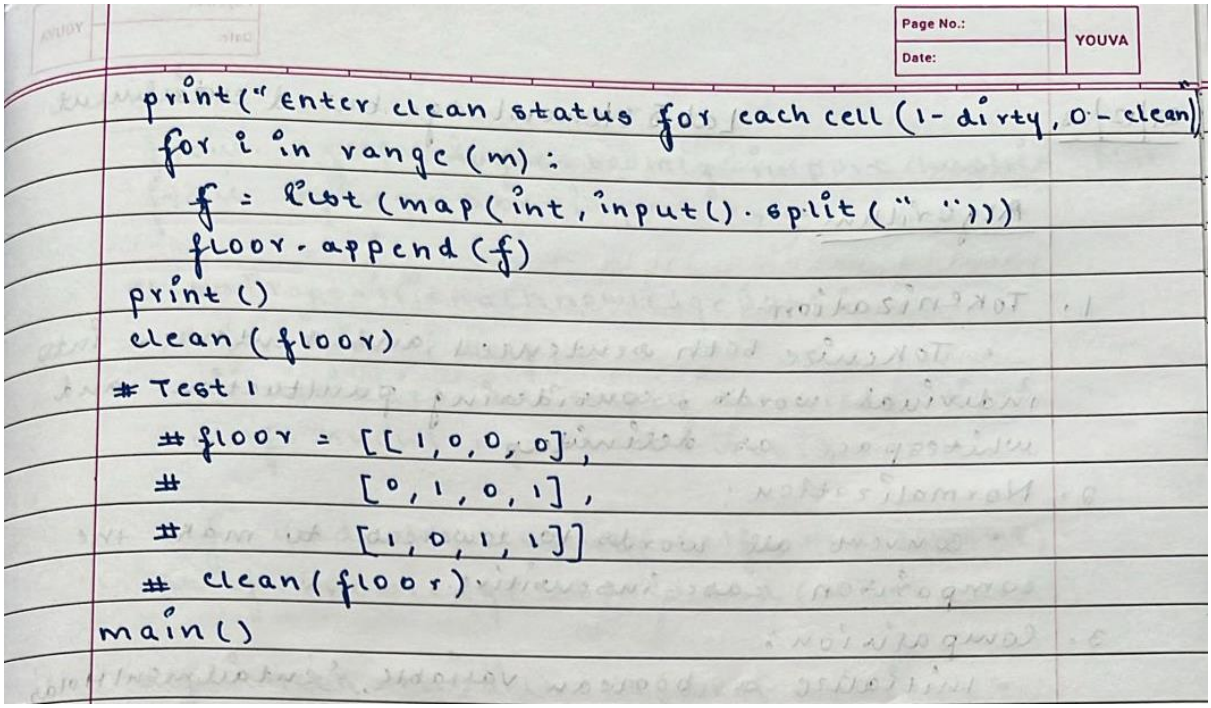
Lab 4 : Vaccum cleaner Agent

Algorithm :

1) Initialize :
   - Place the vaccum cleaner in a starting position.
2) Sensors :
   - Use sensors to detect the current state of the environment (weather a location is dirty or clean).
3) Actions :
   - If the current location is dirty, clean it.
   - If the current location is clean, move to an adjacent location.
   - Repeat steps 2 and 3 until all locations are clean.
4) Environment Interaction
   - Interact with the environment to determine the status of each location
   - Mark cleaned locations
5) Movement
   - Move to the next location based on predefined path.
6) Repeat :
   - Repeat steps 2-5 until all locations are clean.

code:

```
def clean(floor):
    i, j, row, col = 0, 0, len(floor), len(floor[0])
    for i in range(row):
        if (i%2==0):
            for j in range(col):
                if (floor[i][j] == 1):
                    print_F(floor, i, j)
                    floor[i][j] = 0
                    print_F(floor, i, j)
        else:
            for j in range(col-1, -1, -1):
                if (floor[i][j] == 1):
                    print_F(floor, i, j)
                    floor[i][j] = 0
                    print_F(floor, i, j)
def print_F(floor, row, col):
    print("The Floor matrix is as below:")
    for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r==row and c==col:
                print(f"> {floor[r][c]} <", end = '')
            else:
                print(f" {floor[r][c]}", end = '')
        print(end = '\n')
    print(end = '\n')
def main():
    floor = []
    m = int(input("Enter the No. of Rows:"))
    n = int(input("Enter the No of columns:"))
```

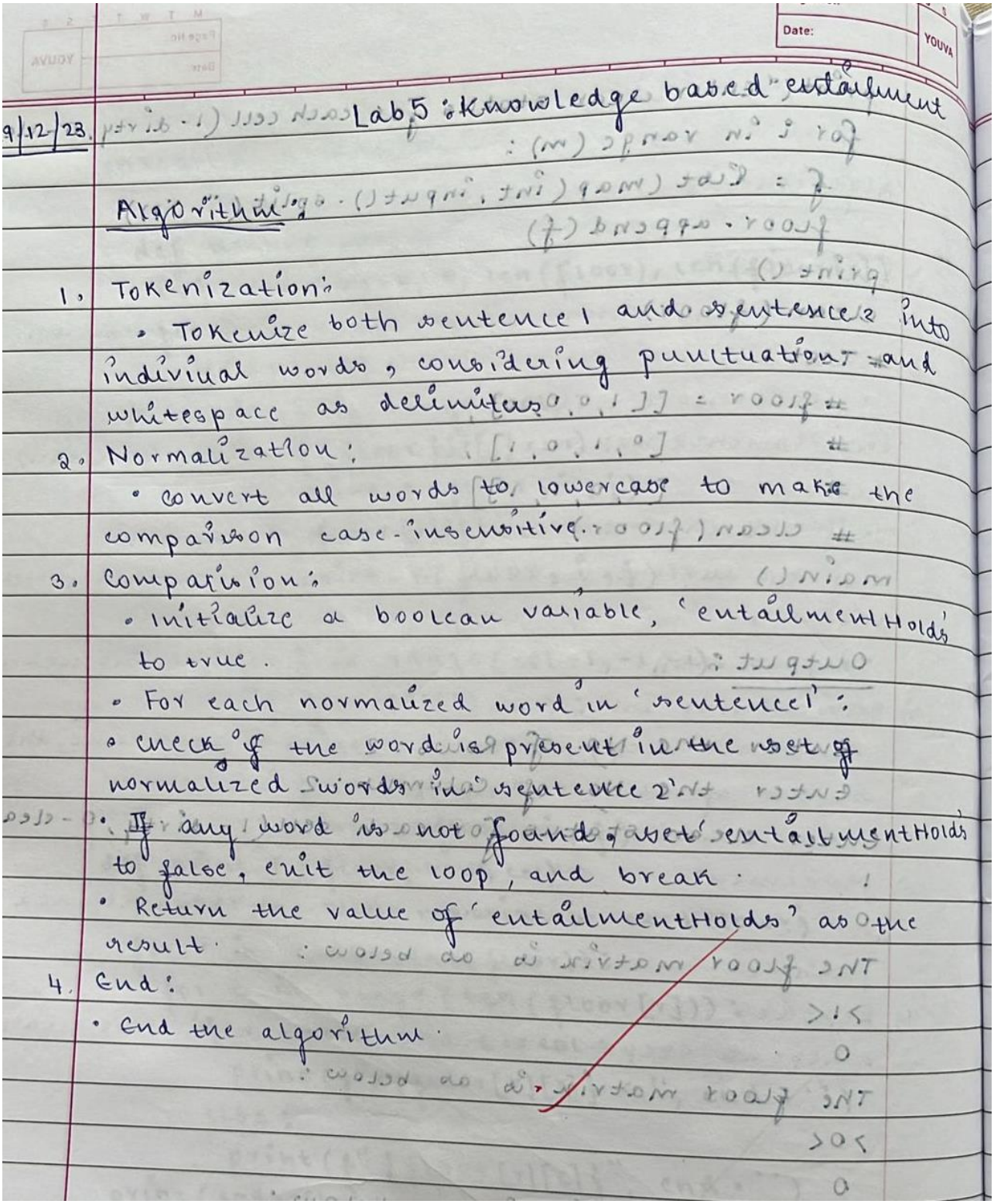


```
print("enter clean status for each cell (1- dirty, 0- clean)")
for i in range(m):
    f = list(map(int,input().split(" ")))
    floor.append(f)
print()
clean(floor)
#Test 1
#floor = [[1,0,0,0],
#         [0,1,0,1],
#         [1,0,1,1]]
#clean(floor)
main()
```

OUTPUT:

```
0 indicates clean and 1 indicates dirty
Enter Location of Vacuumb
Enter status of b1
Enter status of other room1
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
```

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



9/12/23

Lab 5 : Knowledge based entailment

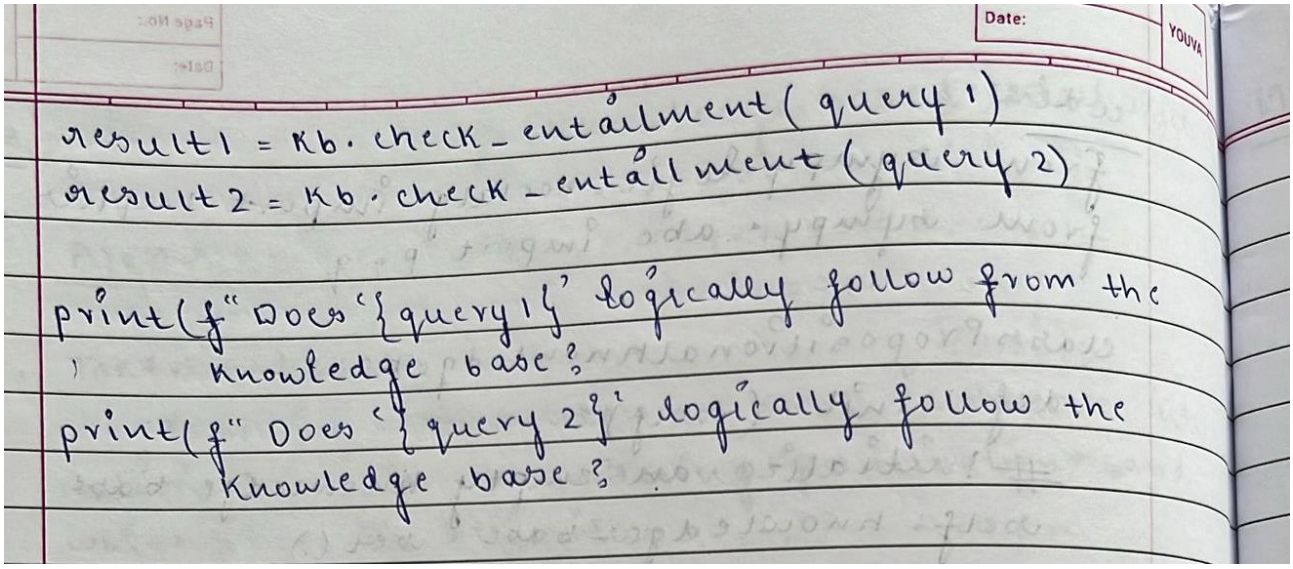
Algorithm :

1. Tokenization :
   - Tokenize both sentence1 and sentence2 into individual words, considering punctuation and whitespace as delimiters.
2. Normalization :
   - Convert all words to lowercase to make the comparison case-insensitive.
3. Comparision :
   - Initialize a boolean variable, 'entailmentHolds' to true.
   - For each normalized word in 'sentence1' :
   - Check if the word is present in the set of normalized words in 'sentence2'.
   - If any word is not found, set 'entailmentHolds' to false, exit the loop, and break.
   - Return the value of 'entailmentHolds' as the result.
4. End :
   - End the algorithm.

Page No.: YOUVA
Date:

code:

```
from sympy.logic.boolalg import Implies, Not
from sympy.abc import p, q

class PropositionalKnowledgeBase:
    def __init__(self):
        # Initialize an empty knowledge base
        self.knowledge_base = set()

    def add_statement(self, statement):
        self.knowledge_base.add(statement)

    def check_entailment(self, query):
        return query.simplify() in self.knowledge_base

Kb = PropositionalKnowledgeBase()
#
Kb.add_statement(Implies(p, q))
Kb.add_statement(Not(q))

query1 = p
query2 = Not(p)
return query.simplify() in self.knowledge_base

Kb = PropositionalKnowledgeBase()

Kb.add_statement(Implies(p, q))
Kb.add_statement(Not(q))

query1 = p
query2 = Not(p)
```



OUTPUT:

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

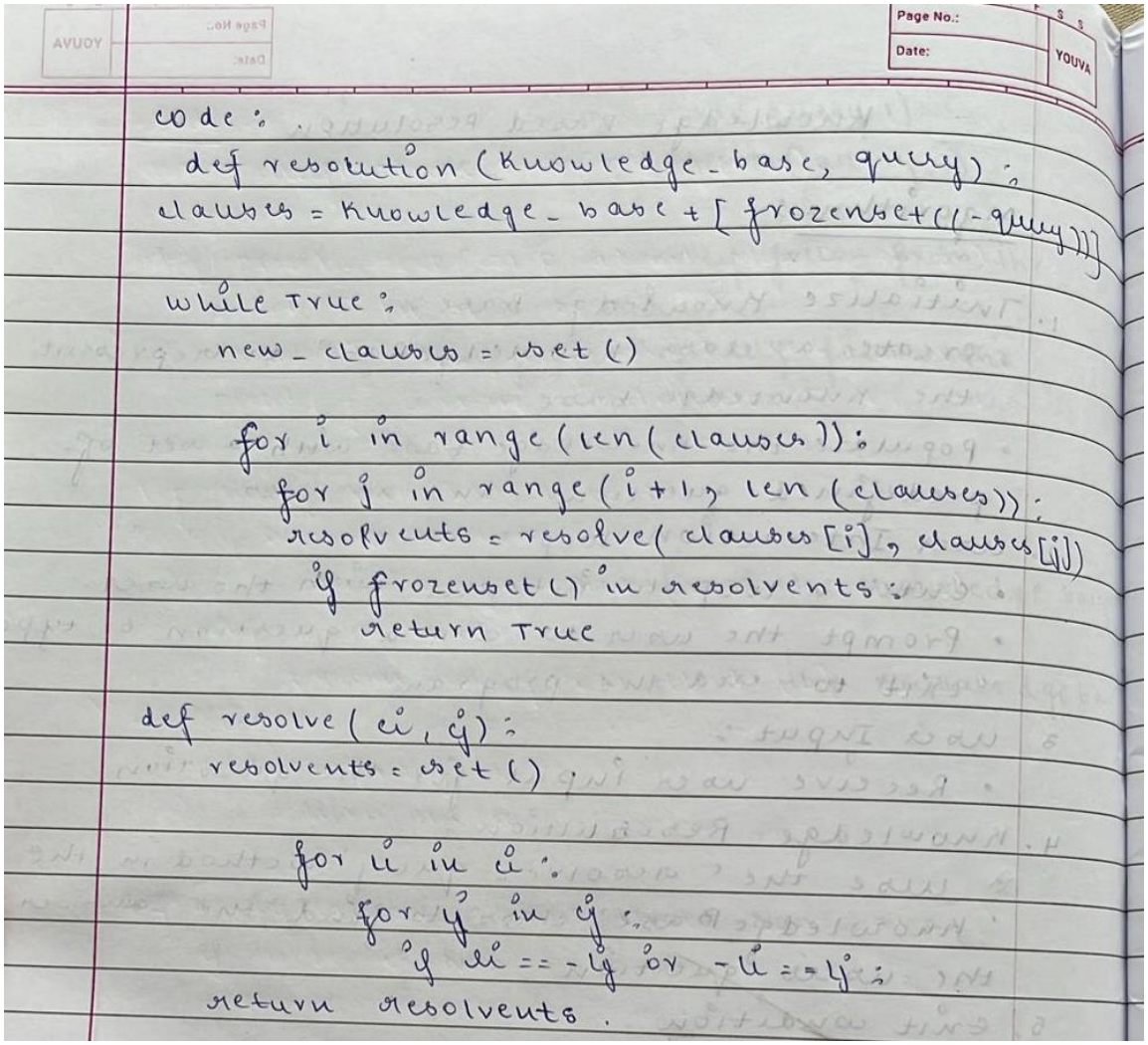
7. Create a knowledge base using prepositional logic and prove the given query using resolution

Date: YOUVA

Knowledge Based Resolution

Algorithm:

1. Initialize Knowledge Base:
   - Create a class 'KnowledgeBase' to represent the knowledge base.
   - Populate the knowledge base with a set of predefined questions and answers.
2. User Interaction loop:
   - Enter a loop to interact with the user.
   - Prompt the user to ask a question or type 'exit' to end the program.
3. User Input:
   - Receive user input for the question.
4. Knowledge Resolution:
   - Use the 'resolve_query' method in the 'KnowledgeBase' class to find the answer to the user's question.
5. Exit condition:



code :

```
def resolution(knowledge_base, query):
clauses = knowledge_base + [frozenset((-query))]

while True:
    new_clauses = set()

    for i in range(len(clauses)):
        for j in range(i+1, len(clauses)):
        resolvents = resolve(clauses[i], clauses[j])
        if frozenset() in resolvents:
            return True

def resolve(ci, cj):
    resolvents = set()

        for ci in ci:
            for cj in cj:
                if ci == -cj or -ci == cj:
    return resolvents.
```

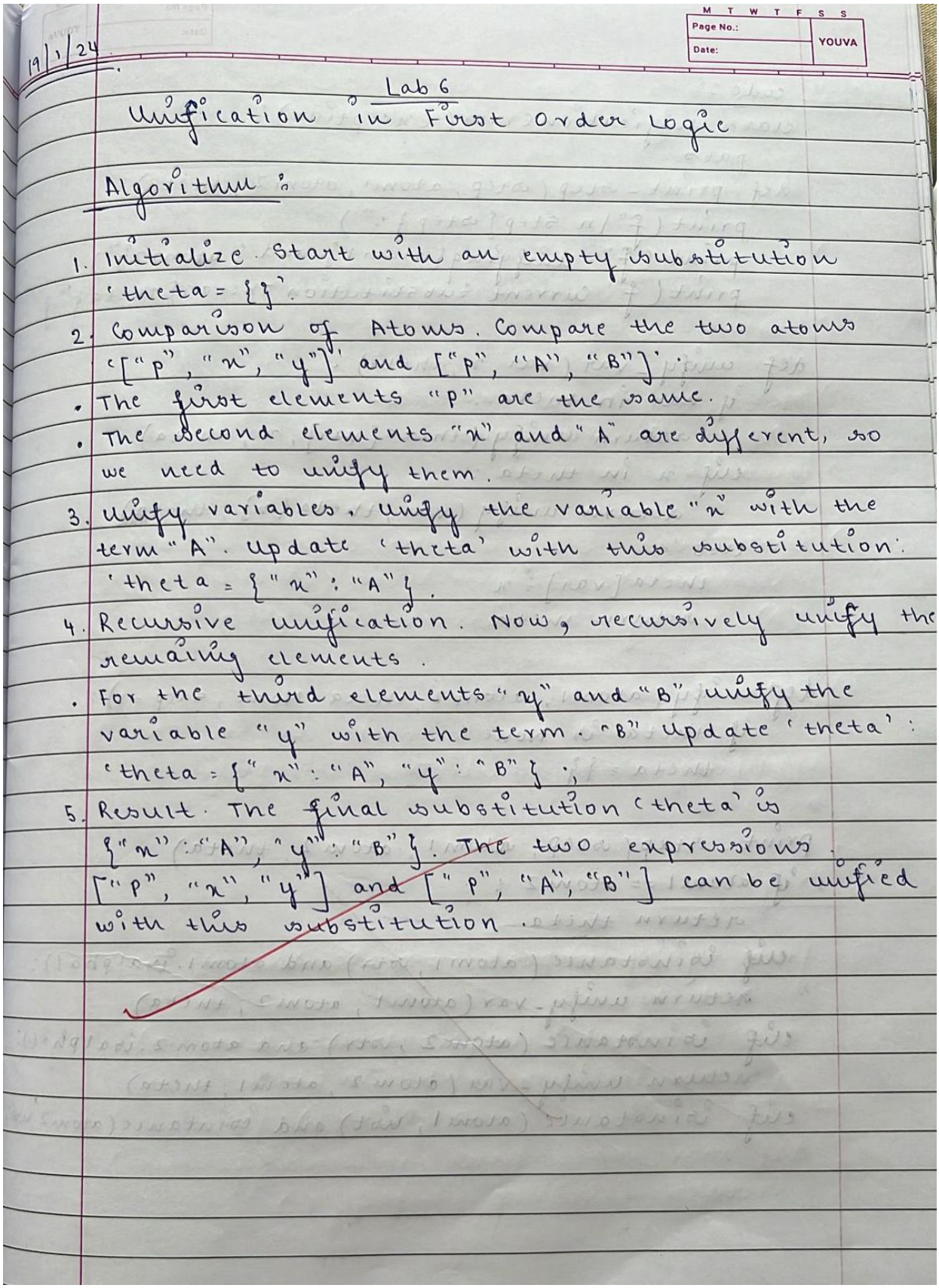
OUTPUT:

```
['~P', 'R']
```

```
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.
```

```
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.
```

8. Implement unification in first order logic



19/1/24

Lab 6

Unification in First Order Logic

Algorithm :

1. Initialize. Start with an empty substitution 'theta = {}'.
2. Comparison of Atoms. Compare the two atoms '["p", "x", "y"]' and '["p", "A", "B"]'.
   - The first elements "p" are the same.
   - The second elements "x" and "A" are different, so we need to unify them.
3. Unify variables. Unify the variable "x" with the term "A". Update 'theta' with this substitution: 'theta = {"x": "A"}'.
4. Recursive unification. Now, recursively unify the remaining elements.
   - For the third elements "y" and "B" unify the variable "y" with the term "B" Update 'theta': 'theta = {"x": "A", "y": "B"}'.
5. Result. The final substitution 'theta' is {"x": "A", "y": "B"}. The two expressions ["p", "x", "y"] and ["p", "A", "B"] can be unified with this substitution.

code:

```
class UnificationError(Exception):
    pass
def print_step(step, atom1, atom2, theta):
    print(f"\n step{step}:")
    print(f" unifying{atom1} and {atom2}")
    print(f" Current substitution Theta:{theta}")

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

def unify(atom1, atom2, theta=None, step=1):
    if theta is None:
        theta = {}

print_step(step, atom1, atom2, theta)
if atom1 == atom2:
    return theta
elif isinstance(atom1, str) and atom1.isalpha():
    return unify_var(atom1, atom2, theta)
elif isinstance(atom2, str) and atom2.isalpha():
    return unify_var(atom2, atom1, theta)
elif isinstance(atom1, list) and isintance(atom2, list)
```

OUTPUT

```
Substitutions:
[('X', 'Richard')]
```

exp1 = "knows(A,x)"

exp2 = "knows(y,mother(y))"

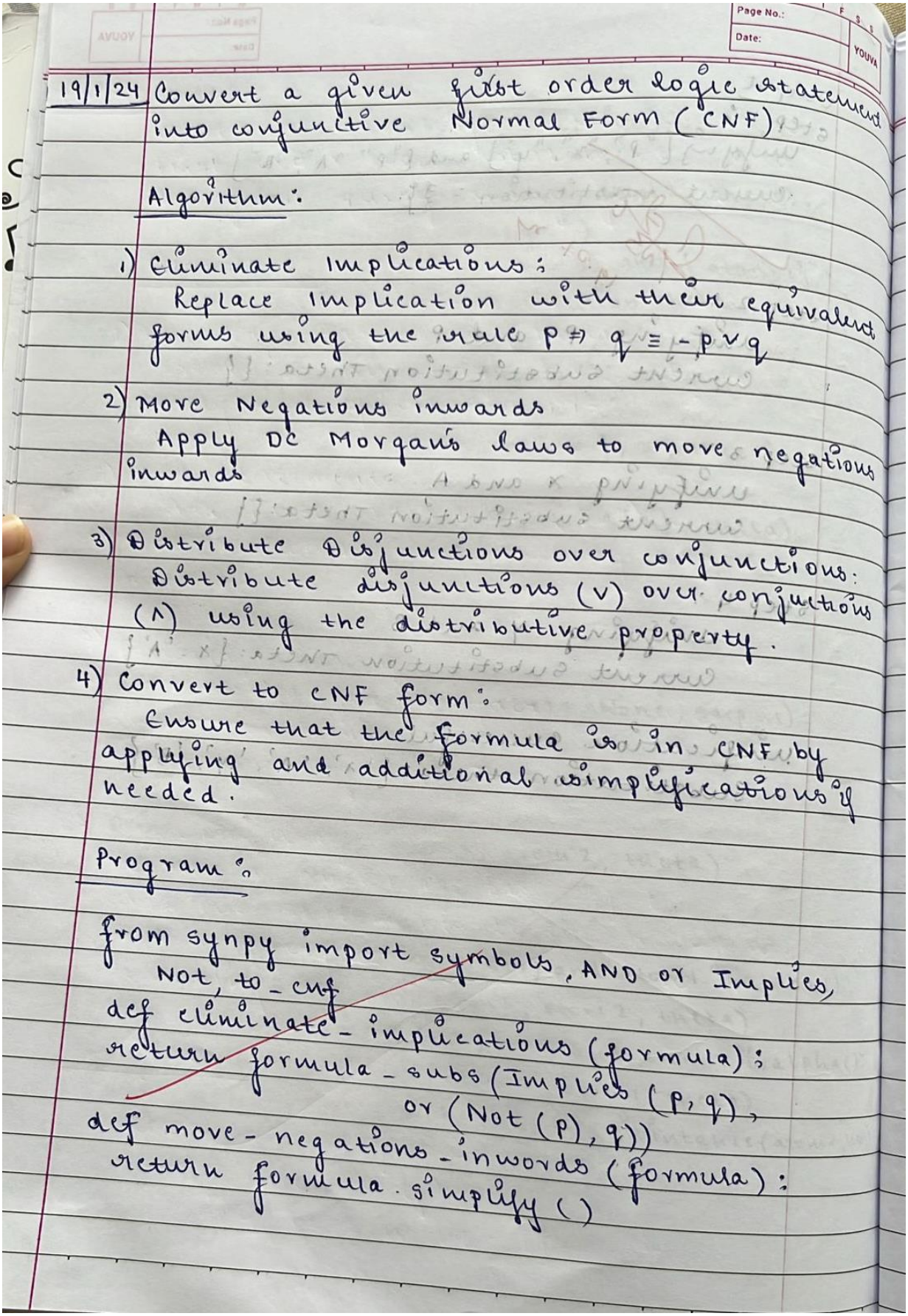
substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)

```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

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



19/1/24 Convert a given first order logic statement into conjunctive Normal Form (CNF).

Algorithm:

1) Eliminate Implications:
   Replace Implication with their equivalent forms using the rule $P \Rightarrow q \equiv \neg P \vee q$

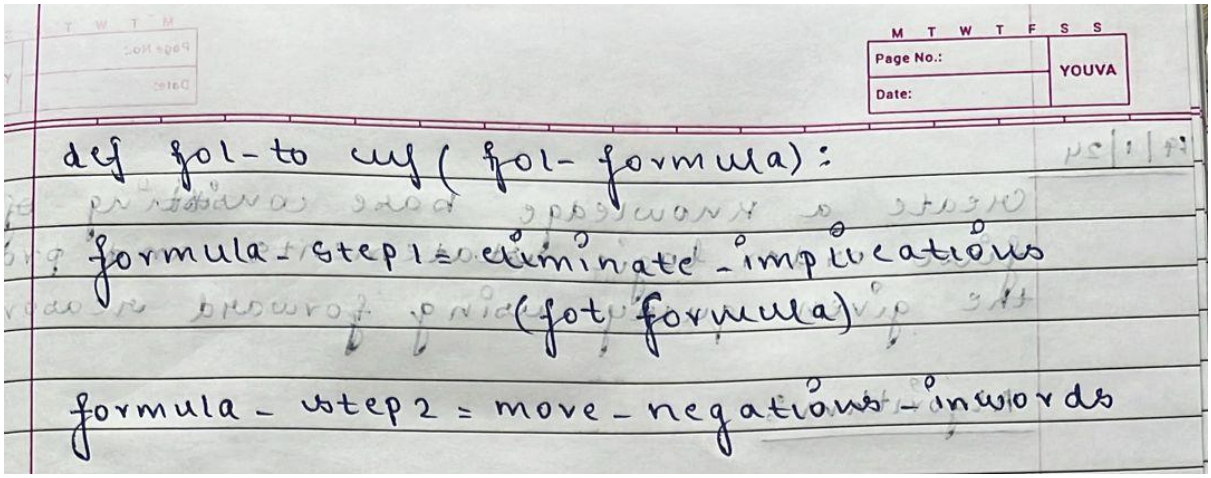
2) Move Negations inwards
   Apply De Morgan's laws to move negations inwards

3) Distribute Disjunctions over conjunctions:
   Distribute disjunctions ($\vee$) over conjuctions ($\wedge$) using the distributive property.

4) Convert to CNF form:
   Ensure that the formula is in CNF by applying and additional simplifications if needed.

Program:

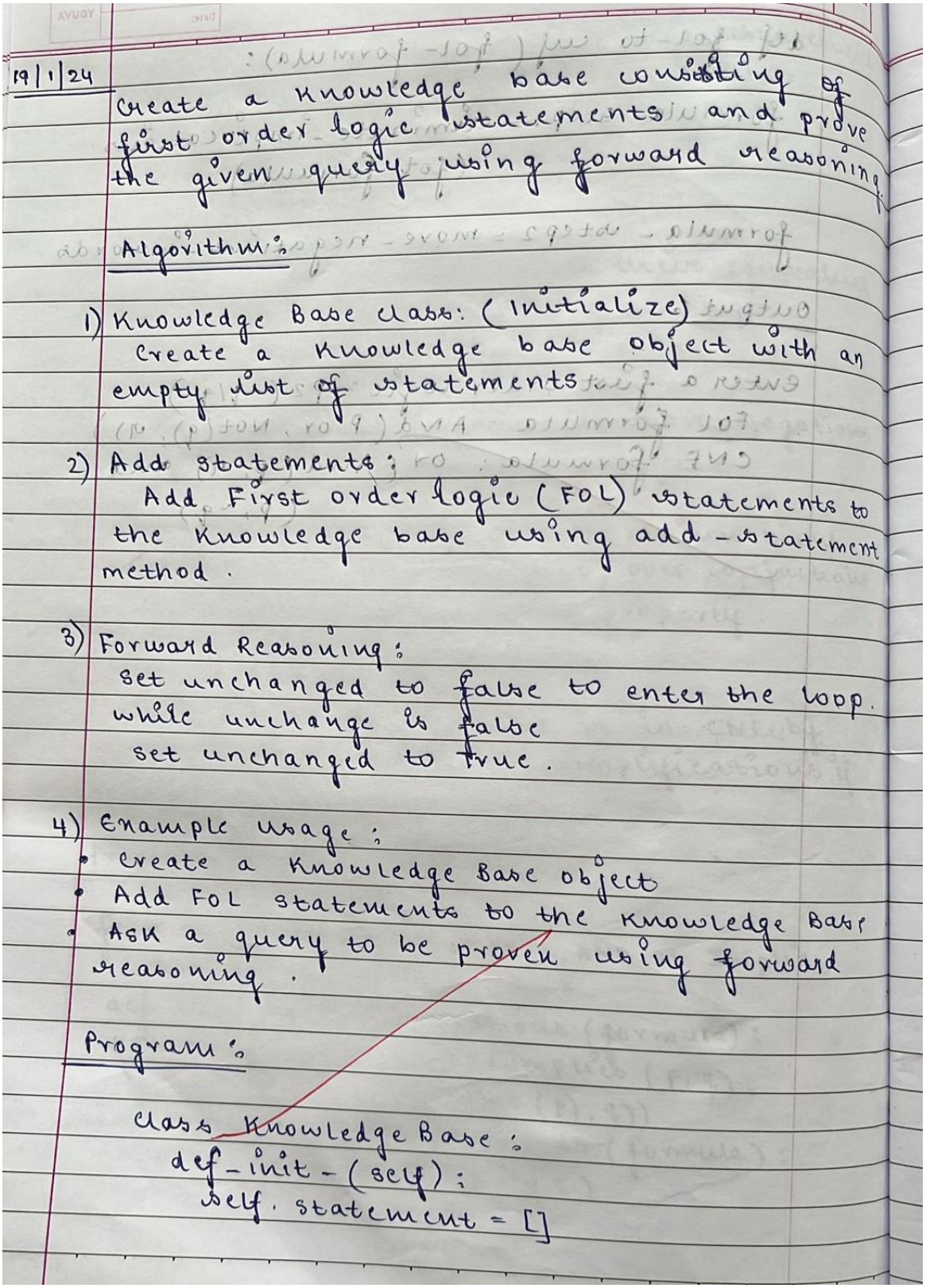
```
from sympy import symbols, AND or Implies,
    Not, to_cnf
def eliminate_implications(formula):
return formula.subs(Implies(P, q),
                    or(Not(P), q))
def move_negations_inwords(formula):
return formula.simplify()
```



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)
```

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



19/1/24

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

Algorithm:

1) Knowledge Base class: (Initialize)
   Create a knowledge base object with an empty list of statements

2) Add statements:
   Add First order logic (FOL) statements to the knowledge base using add-statement method.

3) Forward Reasoning:
   set unchanged to false to enter the loop.
   while unchange is false
   set unchanged to true.

4) Example usage:
   - Create a Knowledge Base object
   - Add FOL statements to the Knowledge Base
   - Ask a query to be proven using forward reasoning.

Program:

```
class KnowledgeBase:
    def-init-(self):
        self.statement = []
```

```
def add_statement(self, statement):
    self.statements.append(statement)

def forward_reasoning(self, query):
    working_memory = set()
    unchanged = false

while not unchanged:
    unchanged = True
    for rule in self.statements:
        if rule conditions.issubset(working memory)
            unchanged = False

return query in working_memory

#Example usage:
    Kb = KnowledgeBase()
    Kb.add_statement({"p", "Q"}).Kb.add_
                                statement({"Q", "R"})
    query = {"O"}
    result = Kb forward_reasoning(query)

if result:
    print("Query is true")
else:
    print("Query is false")
```

OUTPUT

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