1

**Artificial Intelligence Lab Report**

****

***Submitted by***

**Ayush Aditya(1BM23CS057)**

**Batch: A3**

**Course: Artificial Intelligence**

**Course Code: 23CS5PCAIN**

**Sem & Section: 5A**

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

****

**B. M. S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**2025-2026**

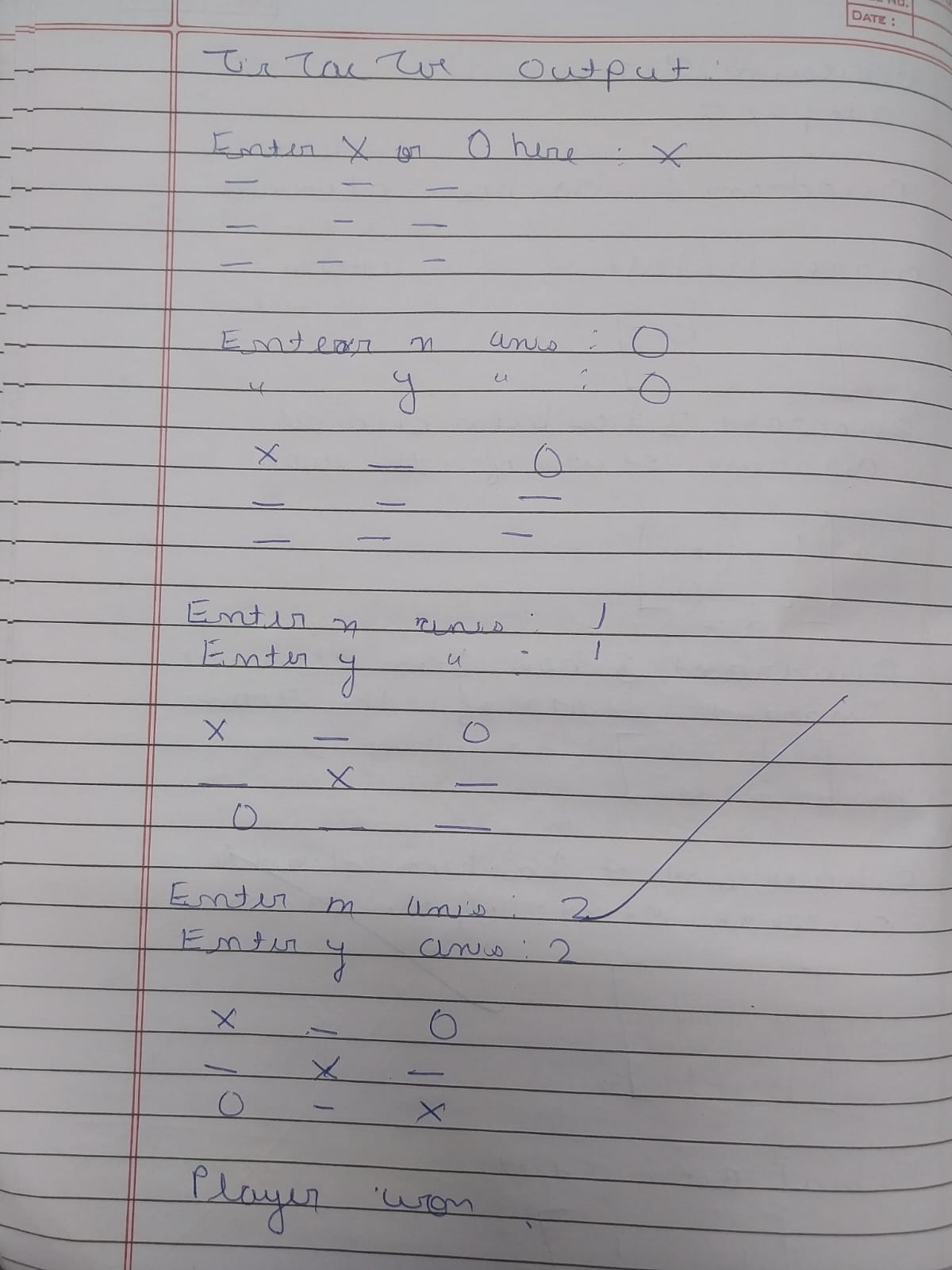
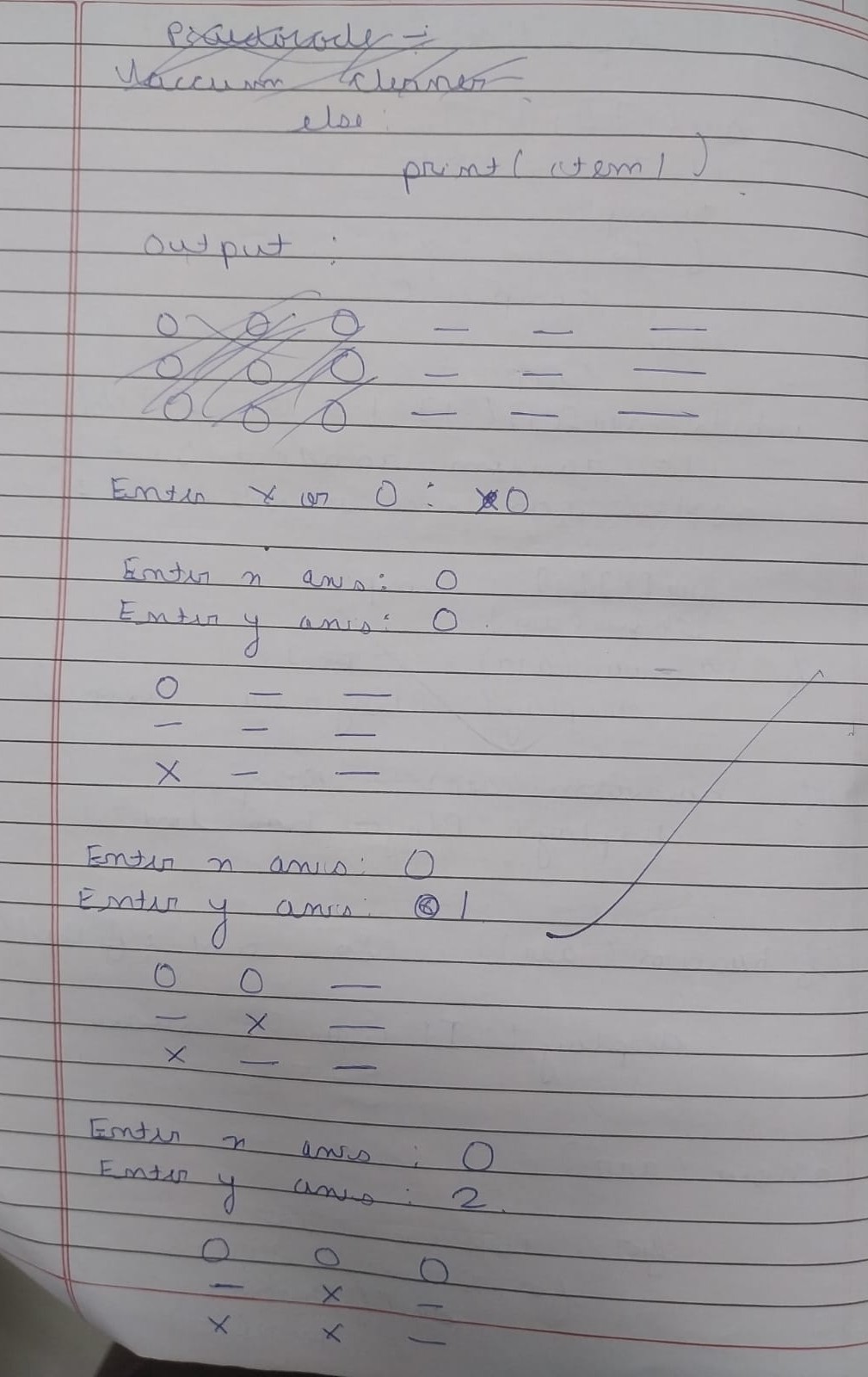
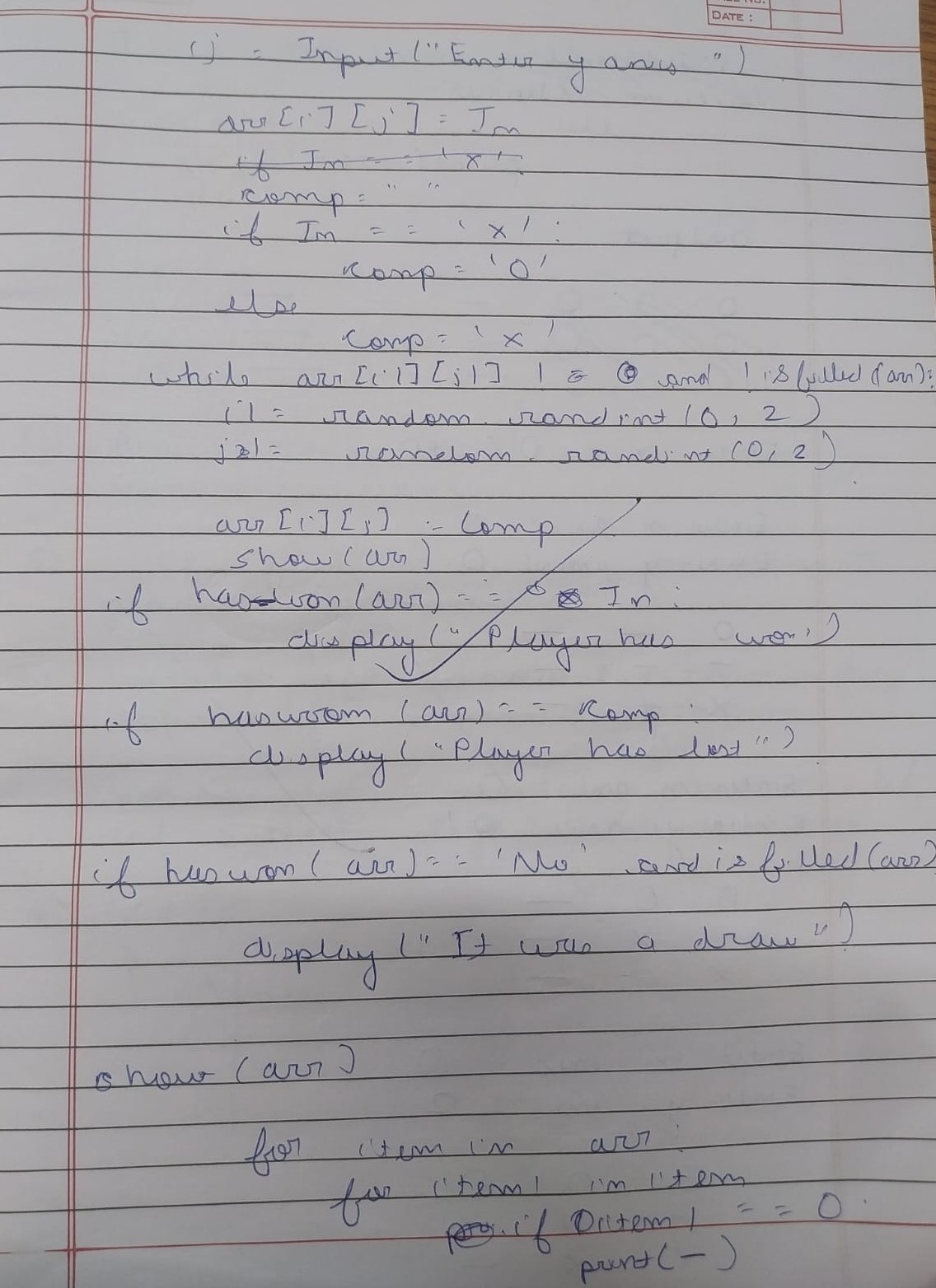
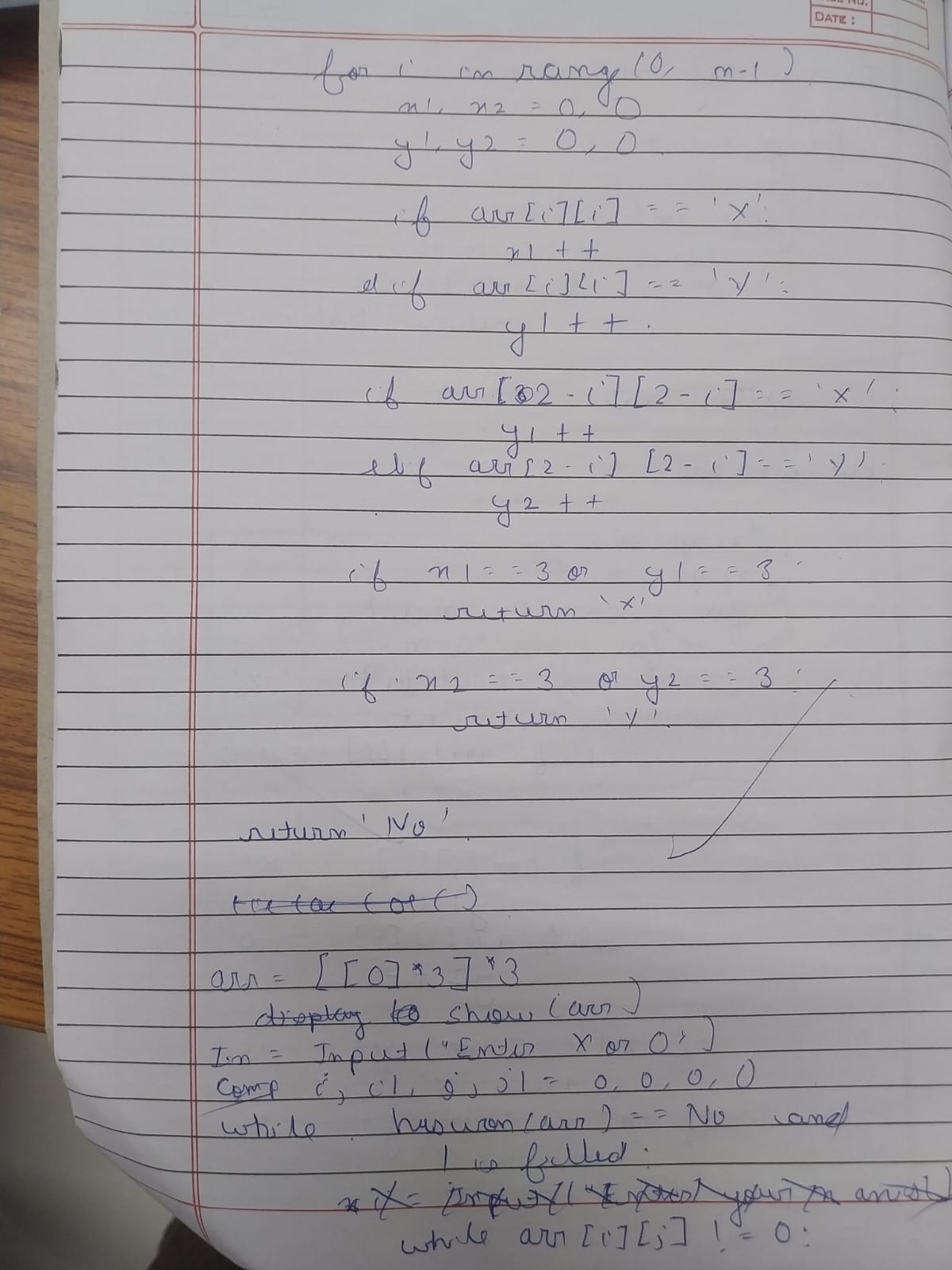
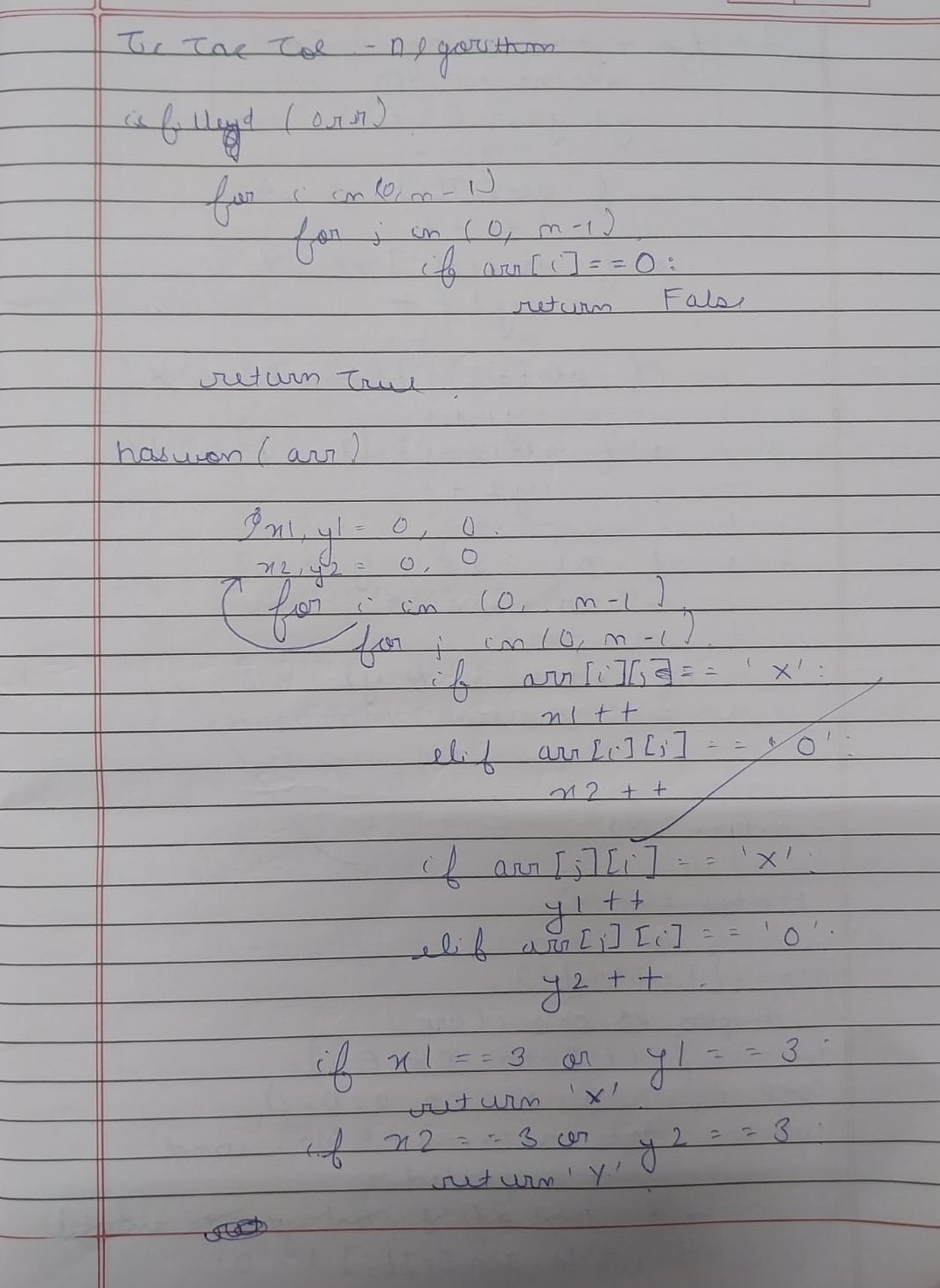
2

**Table of contents**

|  |  |  |
| --- | --- | --- |
| **Program**  **Number** | **Program Title** | **Page Number** |
| **1** | **Tic-Tac-Toe** | **3-10** |
| **2** | **Misplaced tiles – Manhattan distance** | **11-15** |
| **3** | **8-Puzzle IDDFS** | **16-20** |
| **4** | **8-Puzzle A\*** | **21-27** |
| **5** | **Vacuum Cleaner** | **28-32** |
| **6** | **4 queens using hill climbing** | **33-36** |
| **7** | **4 queens using simulated annealing** | **37-41** |
| **8** | **Knowledge Base -Propositional Logic** | **42-47** |
| **9** | **Unification in First**  **Order Logic** | **48-55** |
| **10** | **Resolution Program** | **56-60** |
| **11** | **Forward Chaining inference** | **61-65** |
| **12** | **Alpha Beta Pruning** | **66-70** |

**Program 1 - Tic Tac toe**

**Algorithm**

****

**Code**

import random

def filled(arr):

for i in range(len(arr)):

for j in range(len(arr)):

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

return False

return True

def haswon(arr):

# rows and cols

for i in range(3):

if arr[i][0] == arr[i][1] == arr[i][2] != 0: # row

return arr[i][0]

if arr[0][i] == arr[1][i] == arr[2][i] != 0: # col

return arr[0][i]

# diagonals

if arr[0][0] == arr[1][1] == arr[2][2] != 0:

return arr[0][0]

if arr[0][2] == arr[1][1] == arr[2][0] != 0:

return arr[0][2]

return 'No'

def show(arr):

for row in arr:

for item in row:

if item == 0:

print('\_', end=" ")

else:

print(item, end=" ")

print()

print()

arra = [[0 for \_ in range(3)] for \_ in range(3)]

In = input("Enter X or 0 here : ").upper()

comp = '0' if In == 'X' else 'X'

show(arra)

while haswon(arra) == 'No' and not filled(arra):

# Player move

i = int(input("Enter x axis (0-2): "))

j = int(input("Enter y axis (0-2): "))

while arra[i][j] != 0:

print("Cell already taken, try again.")

i = int(input("Enter x axis (0-2): "))

j = int(input("Enter y axis (0-2): "))

arra[i][j] = In

if haswon(arra) != 'No' or filled(arra):

break

i1, j1 = random.randint(0, 2), random.randint(0, 2)

while arra[i1][j1] != 0:

i1, j1 = random.randint(0, 2), random.randint(0, 2)

arra[i1][j1] = comp

show(arra)

winner = haswon(arra)

show(arra)

if winner == In:

print("Player won ")

elif winner == comp:

print("Player lost ")

else:

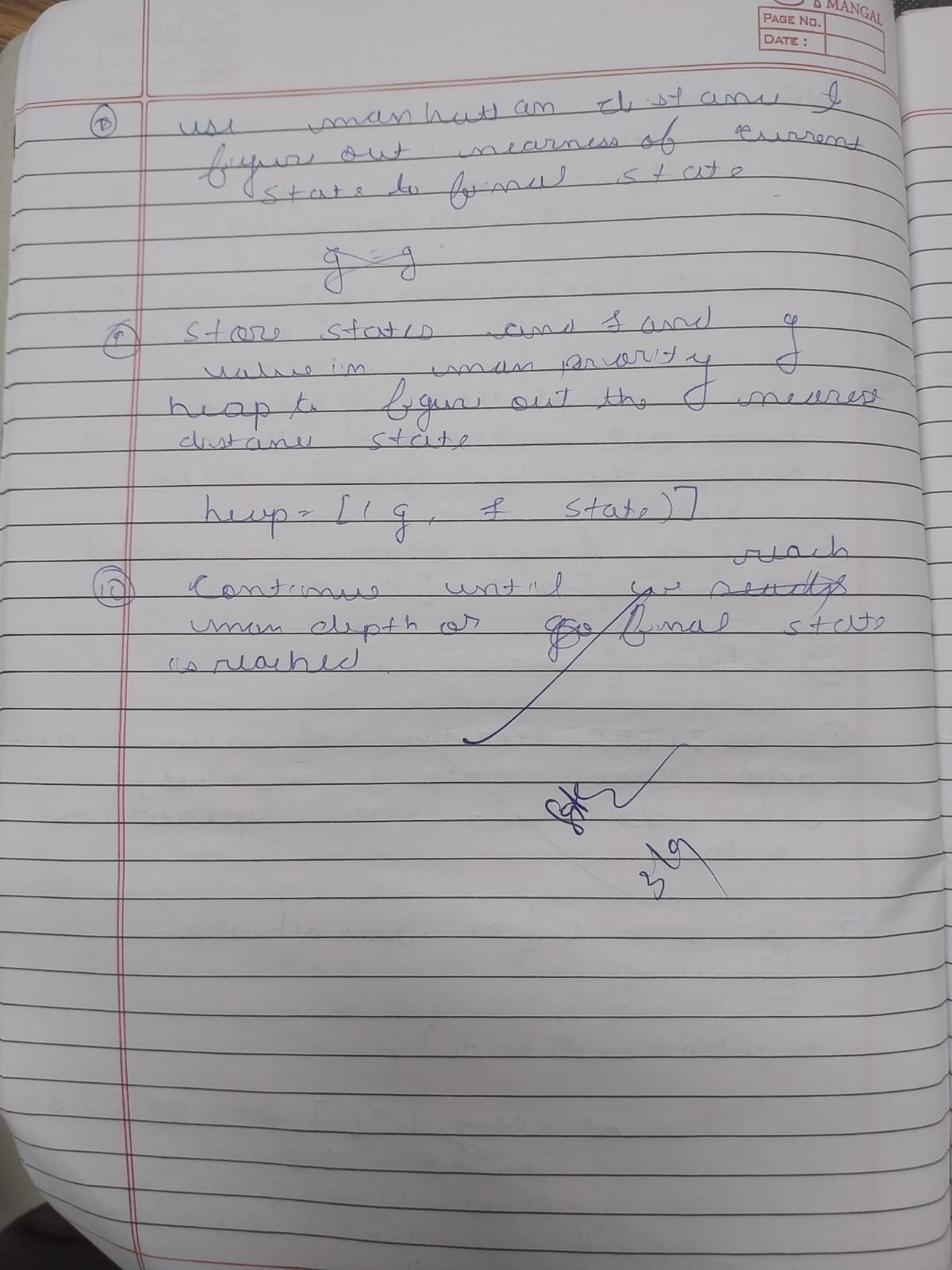
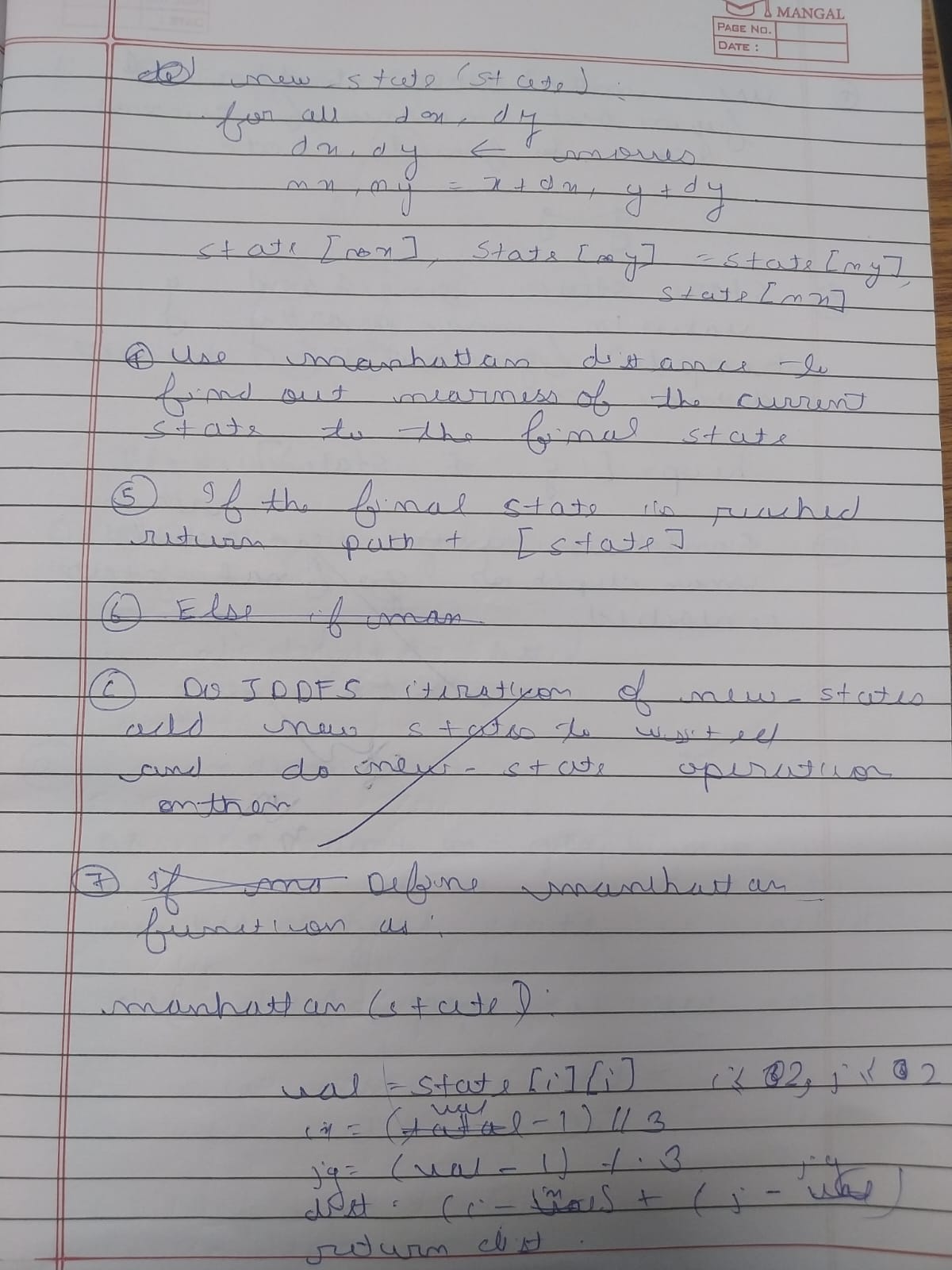
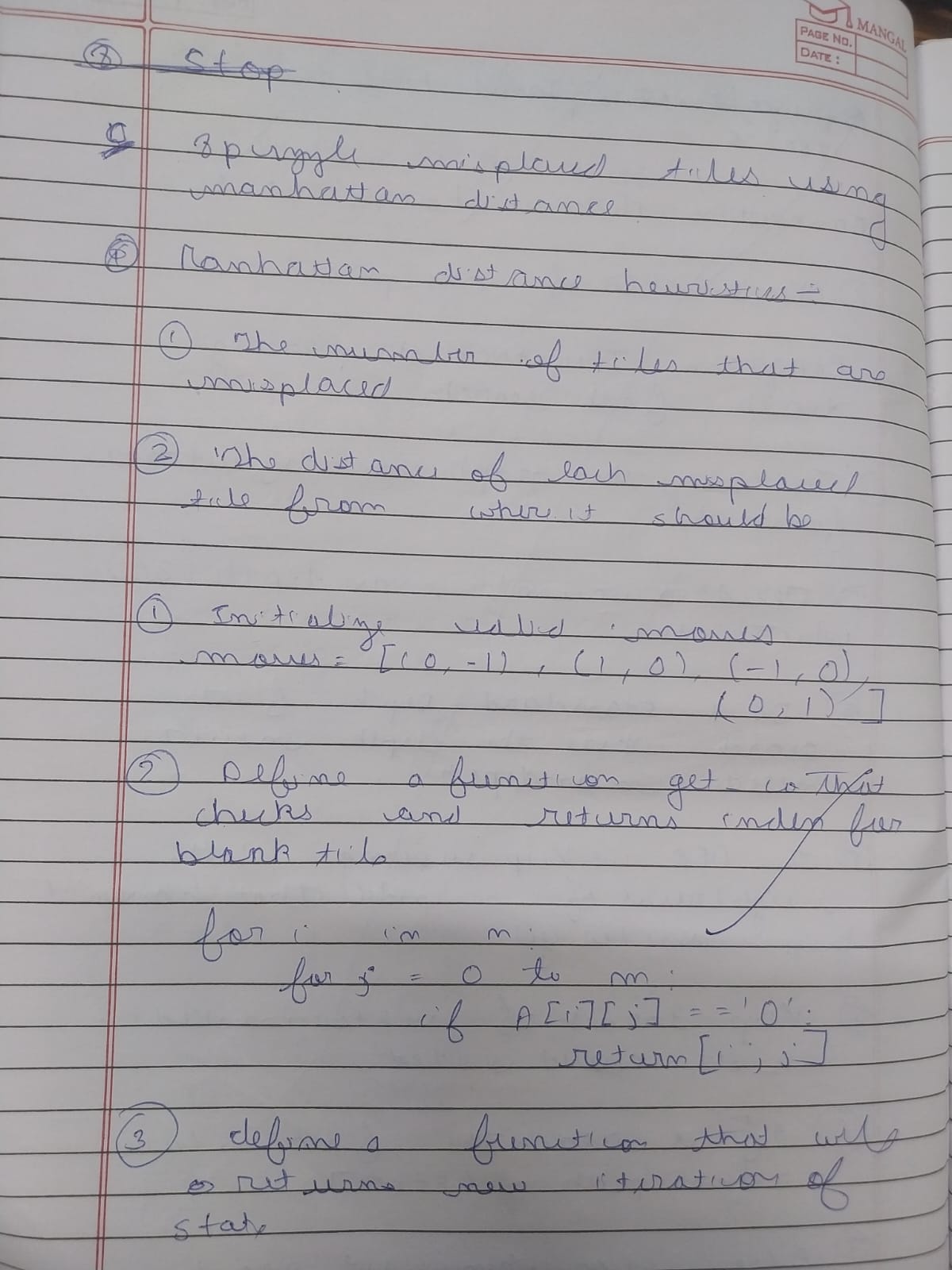
print("It was a draw ")

**Output Snapshot**

****

**Program 2 – Misplaced Tiles Manhattan Distance**

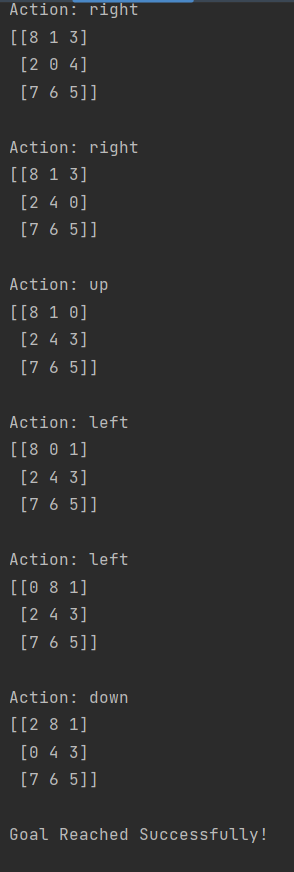
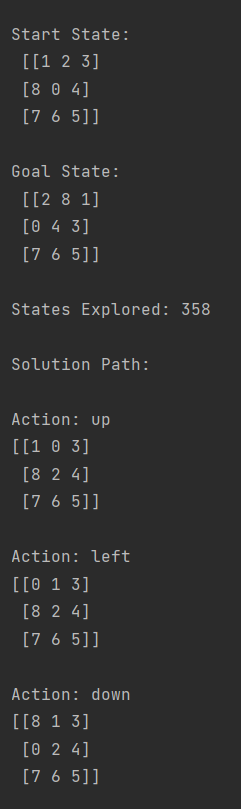
**Algorithm**

****

**Code**

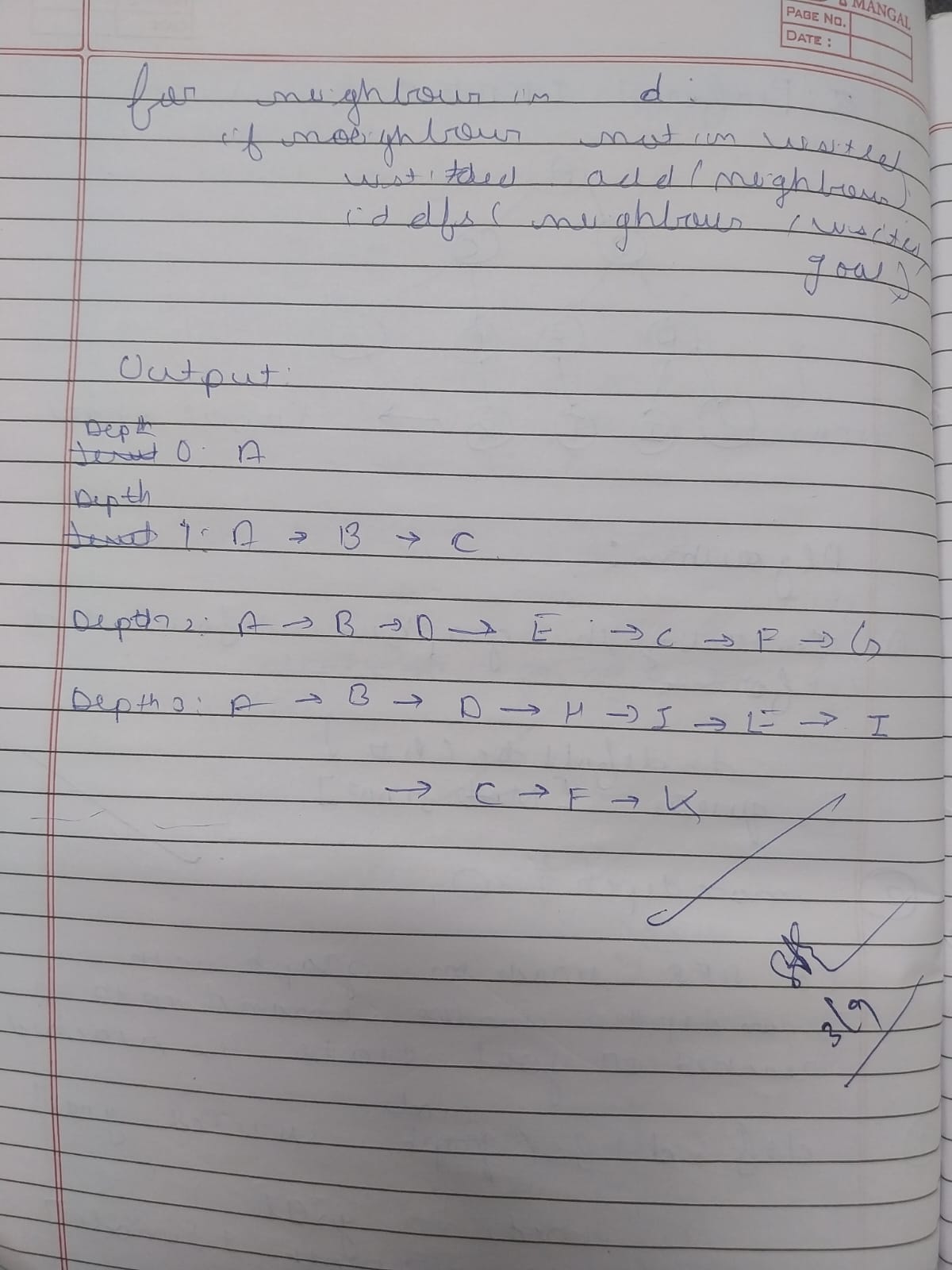
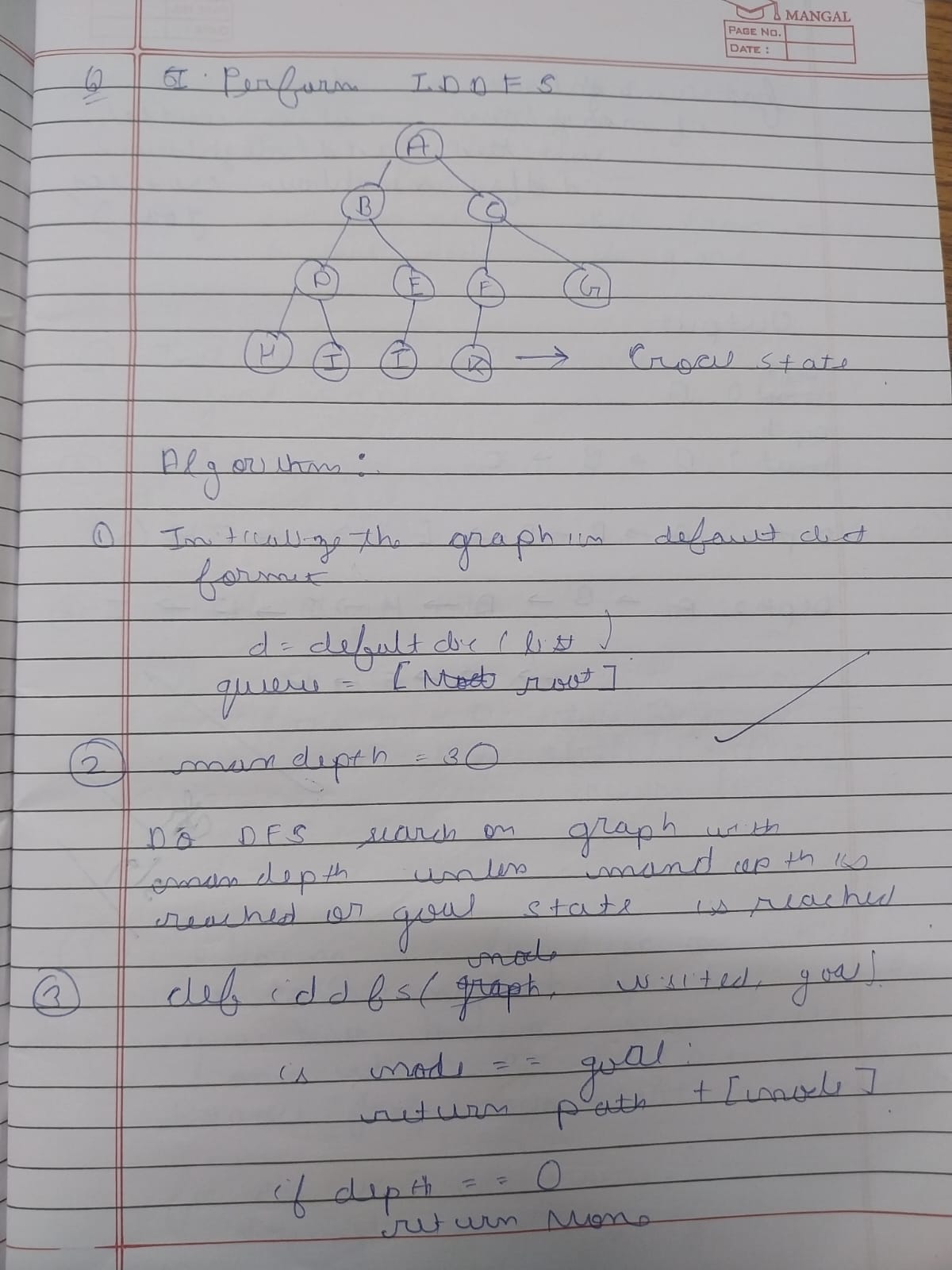
def manhattan(state):  
 dist = 0  
 for i in range(3):  
 for j in range(3):  
 val = state[i][j]  
 if val != 0:  
 target\_x = (val-1) // 3  
 target\_y = (val-1) % 3  
 dist += abs(i - target\_x) + abs(j - target\_y)  
 return dist  
  
def ida\_star(start):  
 def search(path, g, bound):  
 state = path[-1]  
 f = g + manhattan(state)  
 if f > bound:  
 return f  
 if is\_goal(state):  
 return "FOUND"  
 min\_threshold = float("inf")  
 for neighbor in get\_neighbors(state):  
 if neighbor not in path:  
 path.append(neighbor)  
 result = search(path, g+1, bound)  
 if result == "FOUND":  
 return "FOUND"  
 if result < min\_threshold:  
 min\_threshold = result  
 path.pop()  
 return min\_threshold  
 bound = manhattan(start)  
 path = [start]  
 while True:  
 result = search(path, 0, bound)  
 if result == "FOUND":  
 return path  
 if result == float("inf"):  
 return None  
 bound = result

**Output Snapshot**

****

**Program 3 - 8 puzzle using IDDFS**

**Algorithm**

****

**Code**

import copy

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

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

def move(temp, movement):

if movement=="up":

for i in range(3):

for j in range(3):

if(temp[i][j]==-1):

if i!=0:

temp[i][j]=temp[i-1][j]

temp[i-1][j]=-1

return temp

if movement=="down":

for i in range(3):

for j in range(3):

if(temp[i][j]==-1):

if i!=2:

temp[i][j]=temp[i+1][j]

temp[i+1][j]=-1

return temp

if movement=="left":

for i in range(3):

for j in range(3):

if(temp[i][j]==-1):

if j!=0:

temp[i][j]=temp[i][j-1]

temp[i][j-1]=-1

return temp

if movement=="right":

for i in range(3):

for j in range(3):

if(temp[i][j]==-1):

if j!=2:

temp[i][j]=temp[i][j+1]

temp[i][j+1]=-1

return temp

def ids():

global inp

15

global out

global flag

for limit in range(100):

print('LIMIT -> '+str(limit))

stack=[]

inpx=[inp,"none"]

stack.append(inpx)

level=0

while(True):

if len(stack)==0:

break

puzzle=stack.pop(0)

if level<=limit:

print(str(puzzle[1])+" --> "+str(puzzle[0]))

if(puzzle[0]==out):

print("Found")

print('Path cost='+str(level))

flag=True

return

else:

level=level+1

if(puzzle[1]!="down"):

temp=copy.deepcopy(puzzle[0])

up=move(temp, "up")

if(up!=puzzle[0]):

upx=[up,"up"]

stack.insert(0, upx)

if(puzzle[1]!="right"):

temp=copy.deepcopy(puzzle[0])

left=move(temp, "left")

if(left!=puzzle[0]):

leftx=[left,"left"]

stack.insert(0, leftx)

if(puzzle[1]!="up"):

temp=copy.deepcopy(puzzle[0])

down=move(temp, "down")

if(down!=puzzle[0]):

downx=[down,"down"]

stack.insert(0, downx)

if(puzzle[1]!="left"):

temp=copy.deepcopy(puzzle[0])

right=move(temp, "right")

if(right!=puzzle[0]):

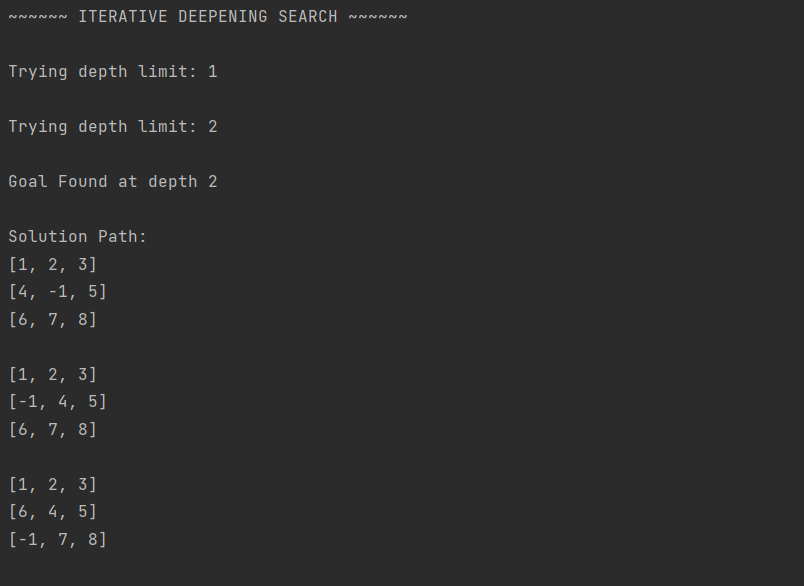
rightx=[right,"right"]

stack.insert(0, rightx)

print('~~~~~~~~~~~~ IDS ~~~~~~~~~~~~')

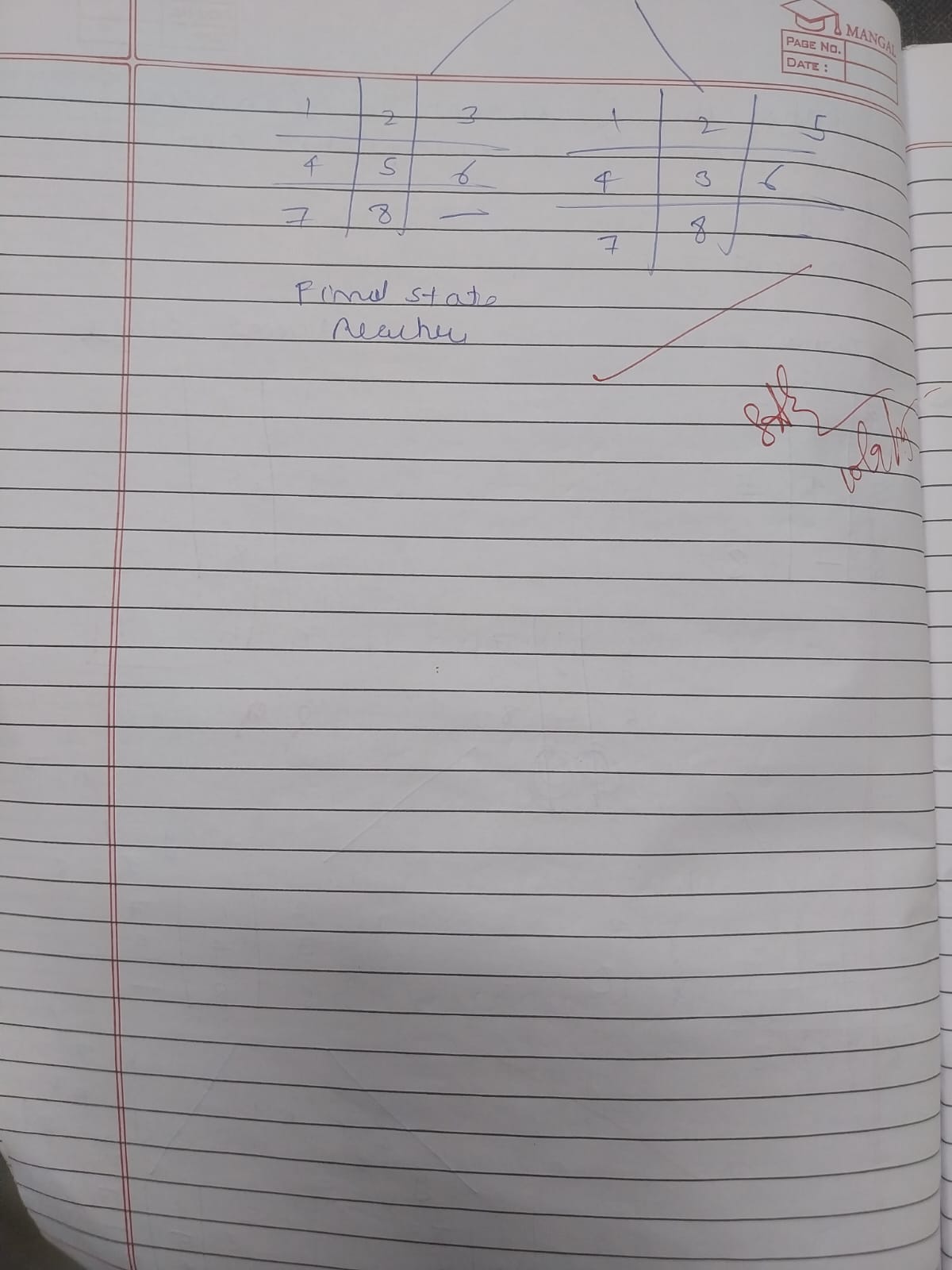
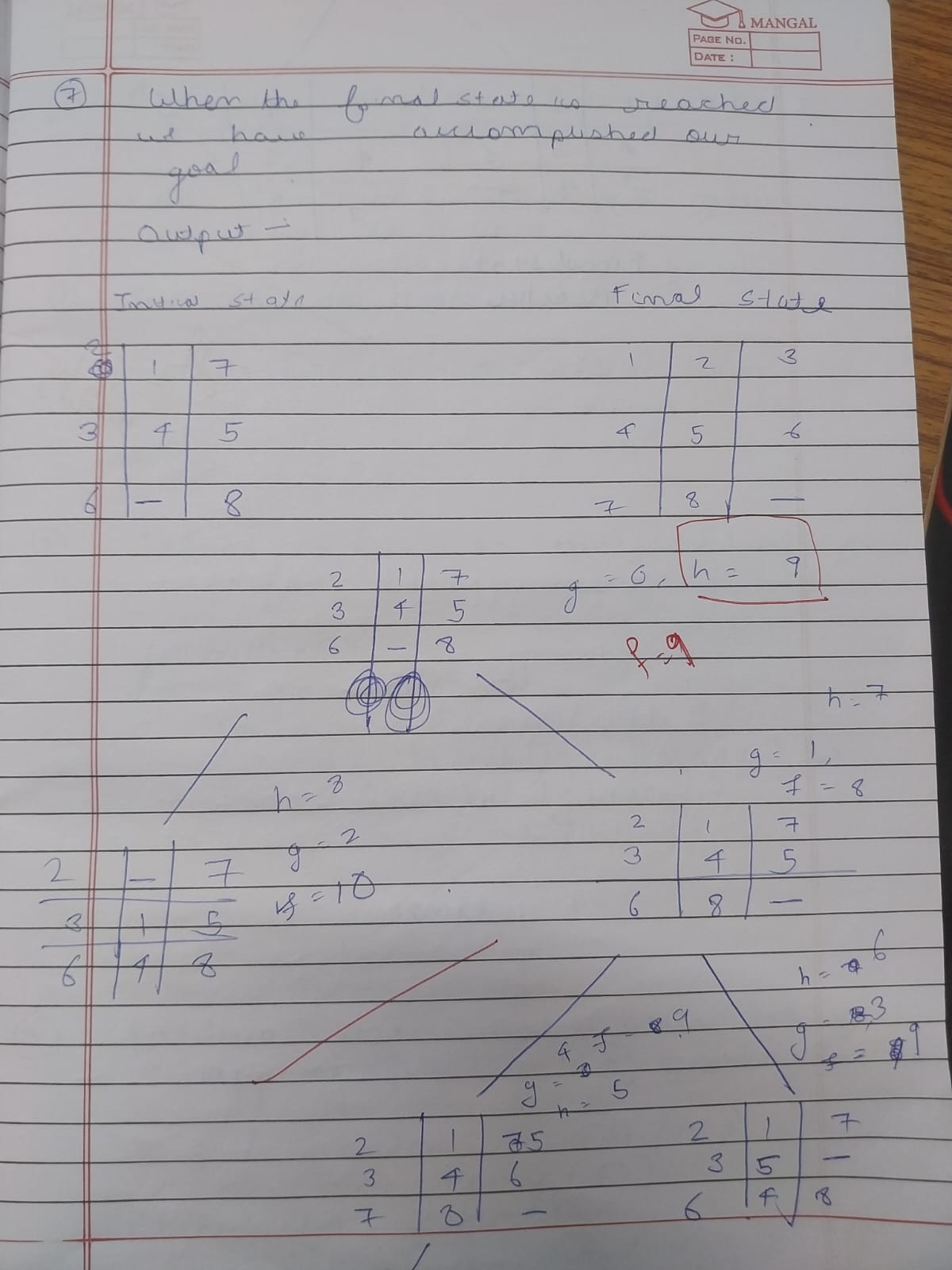
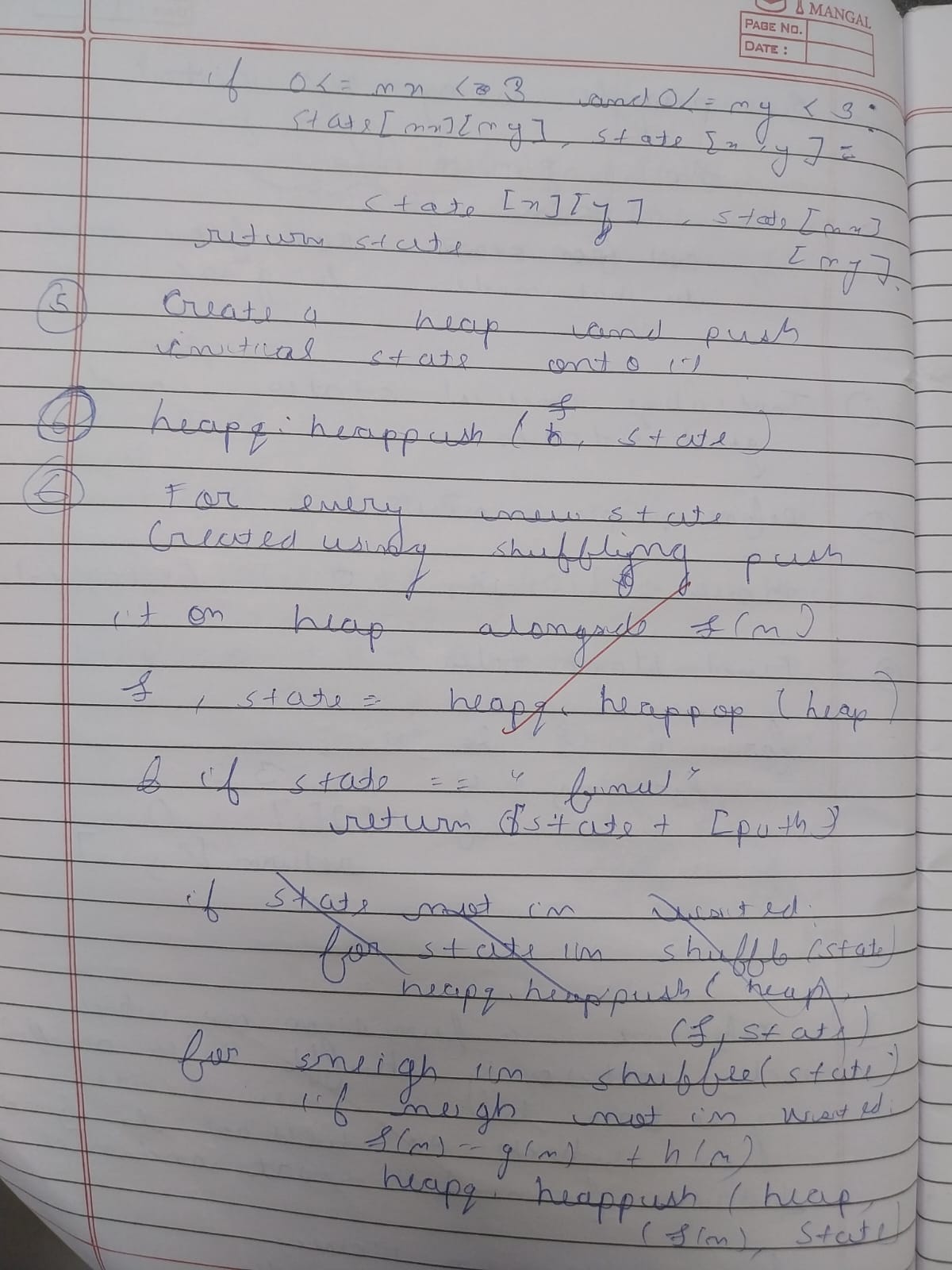
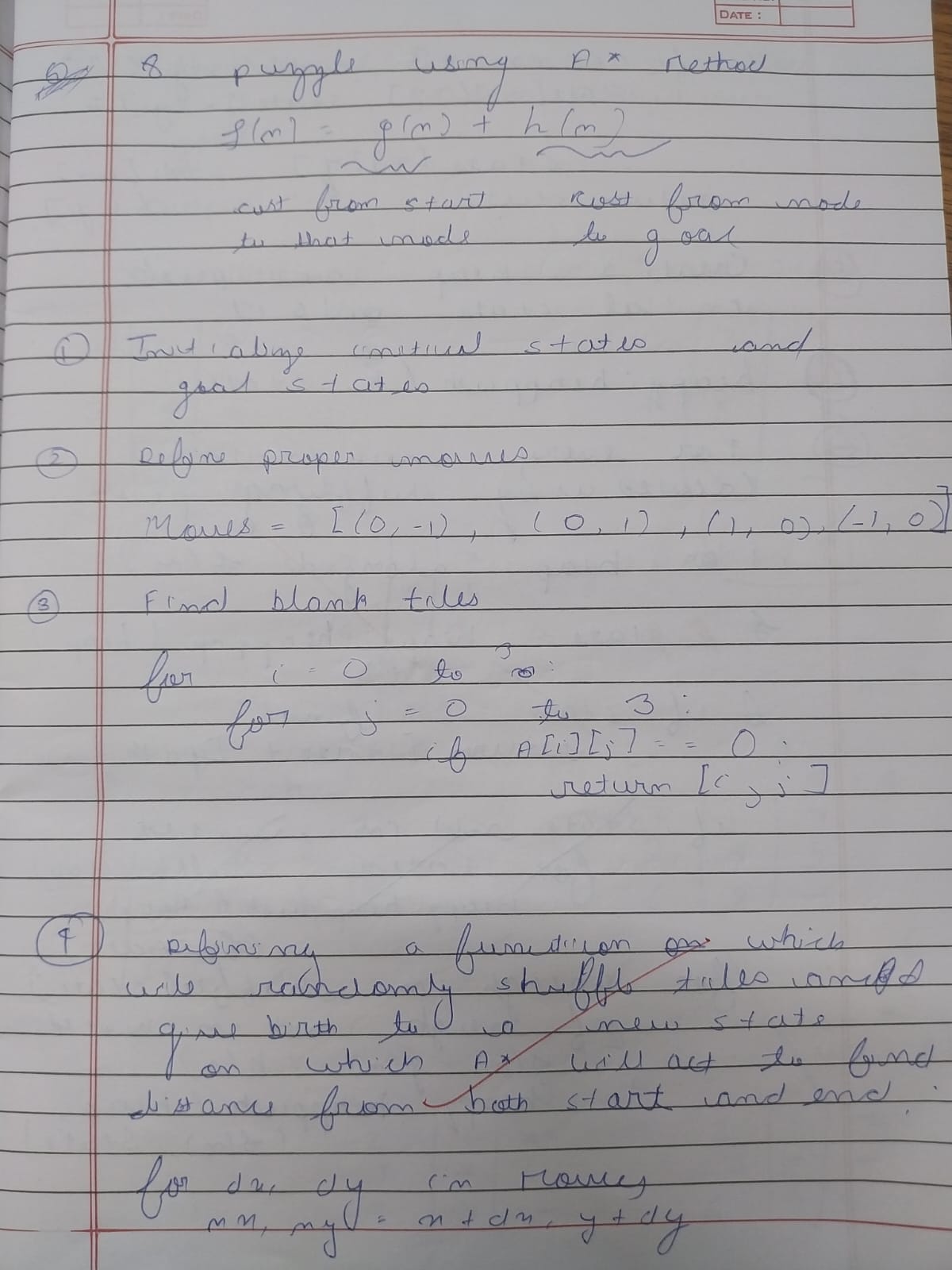
ids()

**Output Snapshot**

****

**Program 04 - 8 Puzzle Using A\***

**Algorithm:**

****

**Code**

def print\_b(src):

state = src.copy()

state[state.index(-1)] = ' '

print(

f"""

{state[0]} {state[1]} {state[2]}

{state[3]} {state[4]} {state[5]}

{state[6]} {state[7]} {state[8]}

“””

)

def h(state, target):

count = 0

i = 0

for j in state:

if state[i] != target[i]:

count = count+1

return count

def astar(state, target):

states = [src]

g = 0

visited\_states = []

while len(states):

print(f"Level: {g}")

moves = []

for state in states:

visited\_states.append(state)

print\_b(state)

if state == target:

print("Success")

return

moves += [move for move in possible\_moves(

state, visited\_states) if move not in moves]

costs = [g + h(move, target) for move in moves]

states = [moves[i]

for i in range(len(moves)) if costs[i] == min(costs)]

g += 1

print("Fail")

def possible\_moves(state, visited\_state):

20

b = state.index(-1)

d = []

if b - 3 in range(9):

d.append('u')

if b not in [0, 3, 6]:

d.append('l')

if b not in [2, 5, 8]:

d.append('r')

if b + 3 in range(9):

d.append('d')

pos\_moves = []

for m in d:

pos\_moves.append(gen(state, m, b))

return [move for move in pos\_moves if move not in visited\_state]

def gen(state, m, b):

temp = state.copy()

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]

if m == 'd':

temp[b + 3], temp[b] = temp[b], temp[b + 3]

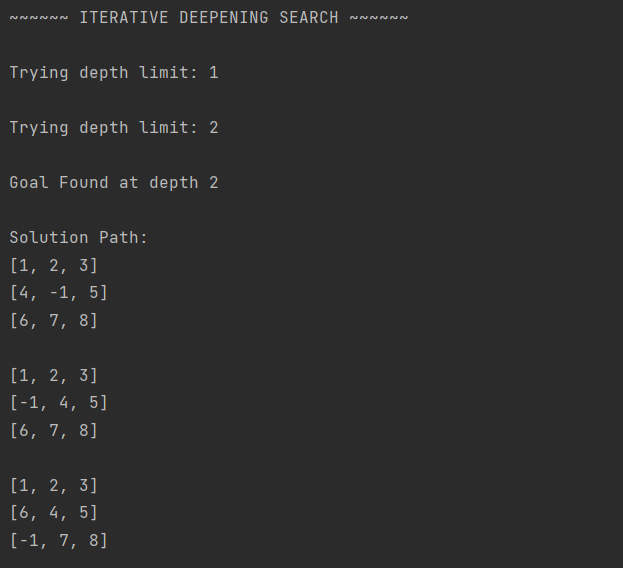
return temp

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

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

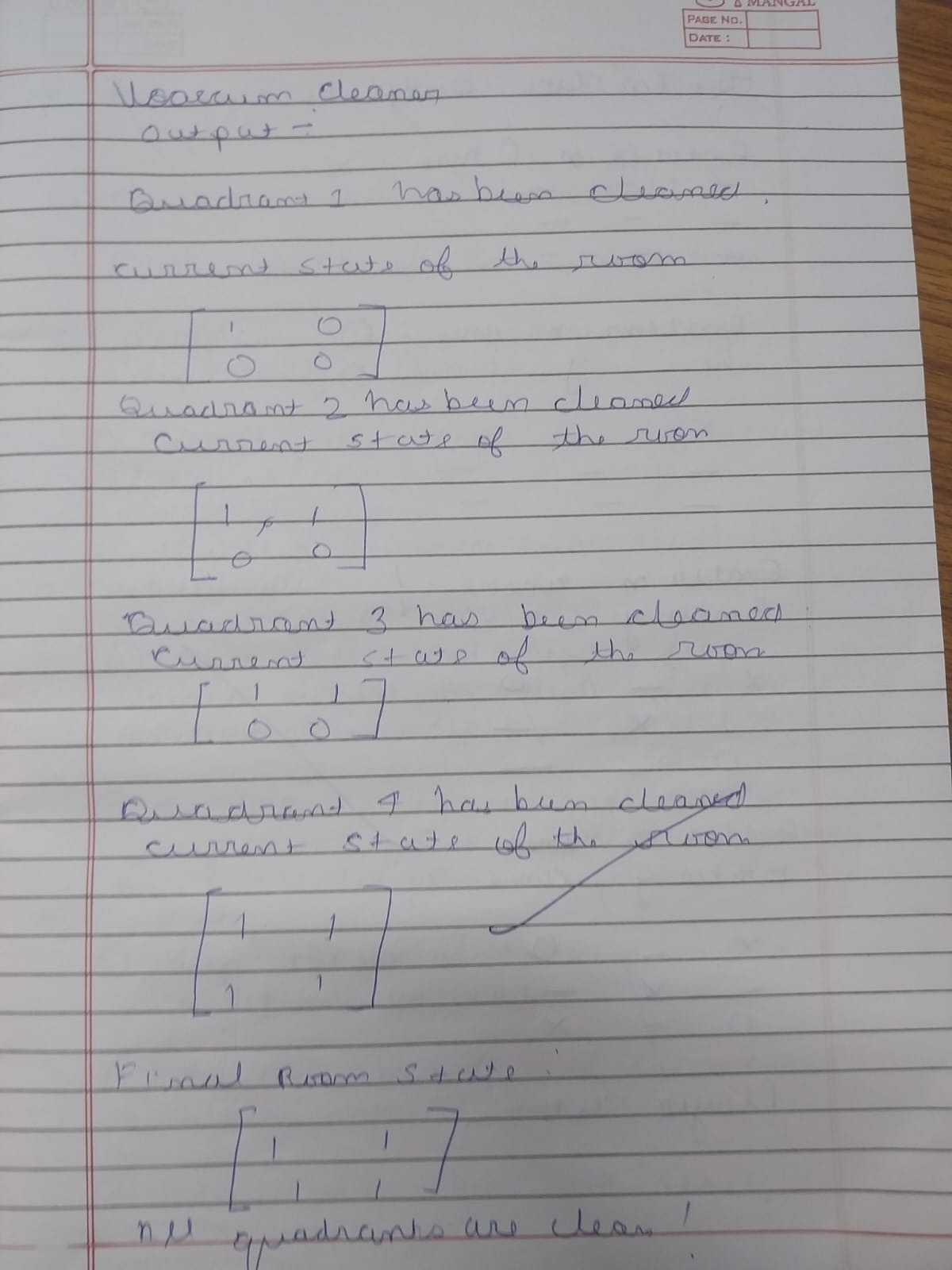
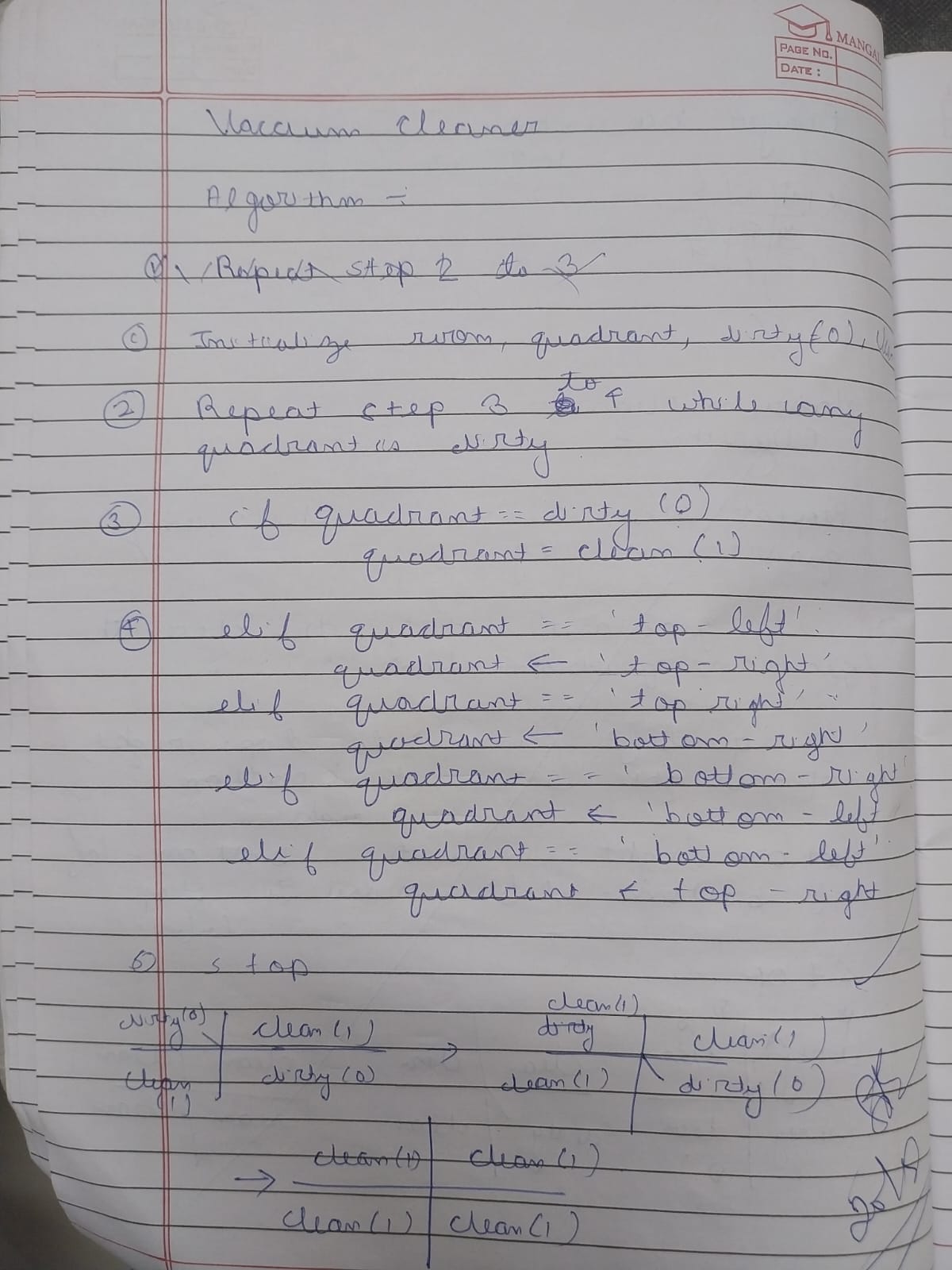
astar(src, target)

**Output Snapshot:**



**Program 5 - Vacuum Cleaner**

**Algorithm**

****

**Code**

def vacuum\_world():

goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum: ")

status\_input = input("Enter status of " + location\_input+ " : ")

status\_input\_complement = input("Enter status of other room : ")

print("Initial Location Condition {A : " + str(status\_input\_complement) + ", B : " + str(status\_input) + " }" )

23

if location\_input == 'A':

print("Vacuum is placed in Location A")

if status\_input == '1':

print("Location A is Dirty.")

goal\_state['A'] = '0'

cost += 1 #cost for suck

print("Cost for CLEANING A " + str(cost))

print("Location A has been Cleaned.")

if status\_input\_complement == '1':

print("Location B is Dirty.")

print("Moving right to the Location B. ")

cost += 1

print("COST for moving RIGHT " + str(cost))

goal\_state['B'] = '0'

cost += 1

print("COST for SUCK " + str(cost))

print("Location B has been Cleaned. ")

else:

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

print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ")

if status\_input\_complement == '1':

print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ")

cost += 1

print("COST for moving RIGHT " + str(cost))

goal\_state['B'] = '0'

cost += 1

print("Cost for SUCK" + str(cost))

print("Location B has been Cleaned. ")

else:

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

print(cost)

print("Location B is already clean.")

else:

24

print("Vacuum is placed in location B")

if status\_input == '1':

print("Location B is Dirty.")

goal\_state['B'] = '0'

cost += 1

print("COST for CLEANING " + str(cost))

print("Location B has been Cleaned.")

if status\_input\_complement == '1':

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1

print("COST for moving LEFT " + str(cost))

goal\_state['A'] = '0'

cost += 1

print("COST for SUCK " + str(cost))

print("Location A has been Cleaned.")

else:

print(cost)

print("Location B is already clean.")

if status\_input\_complement == '1':

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1

print("COST for moving LEFT " + str(cost))

goal\_state['A'] = '0'

cost += 1

print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ")

else:

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

print("Location A is already clean.")

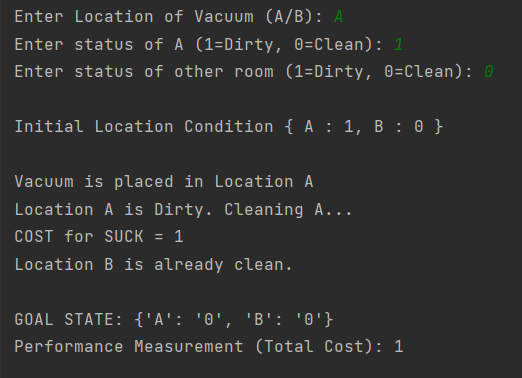
print("GOAL STATE: ")

print(goal\_state)

print("Performance Measurement: " + str(cost))

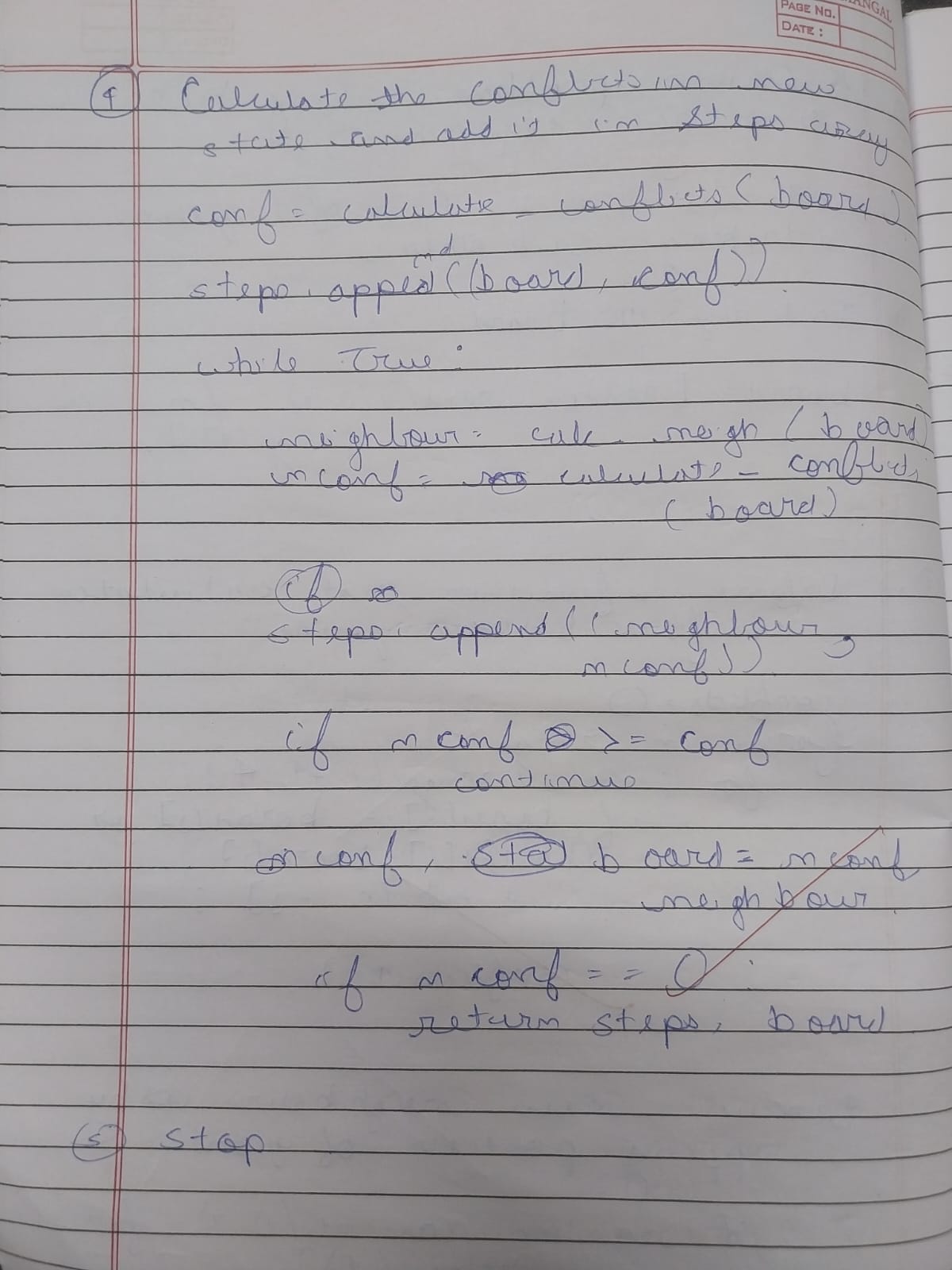
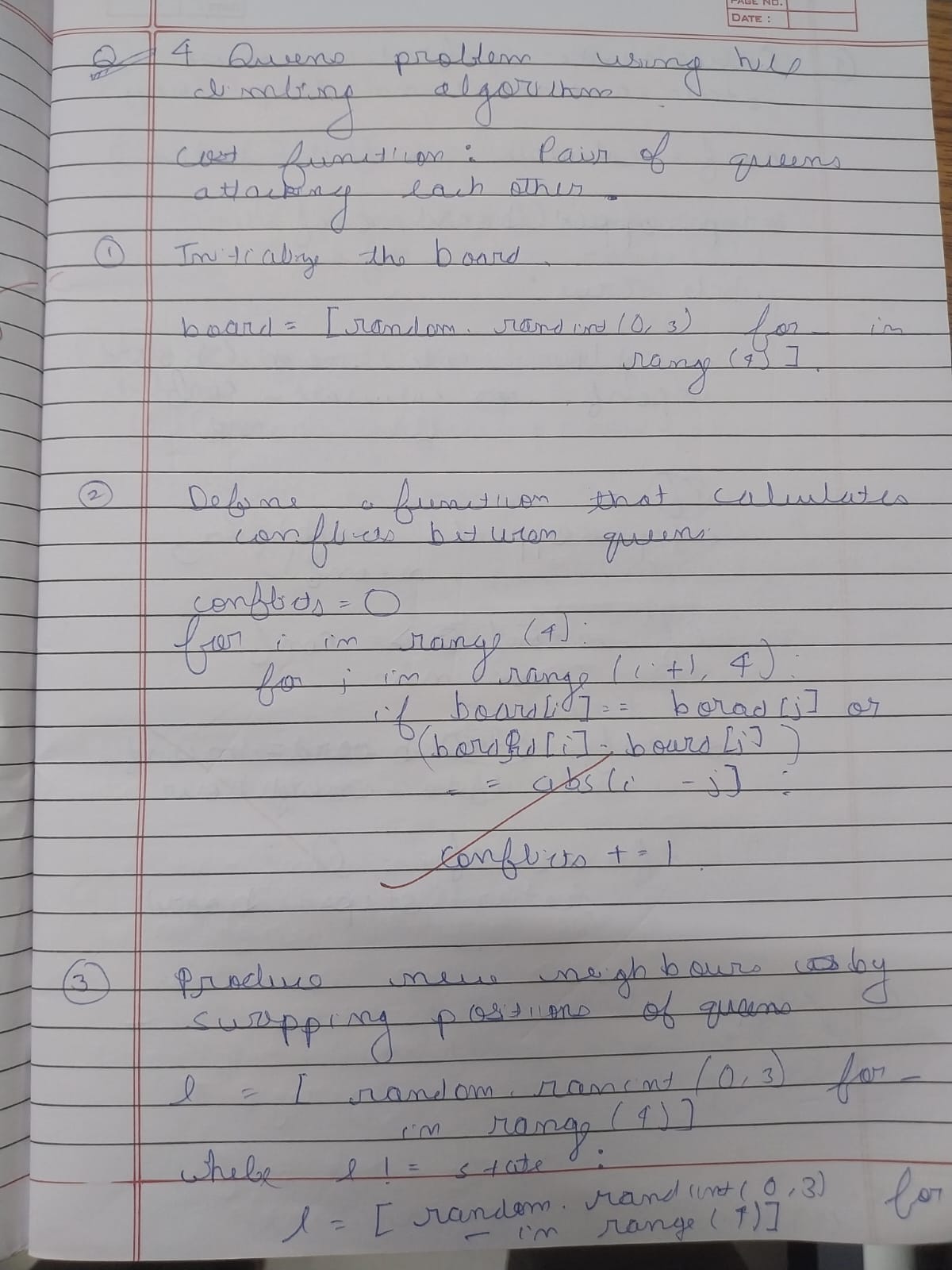
vacuum\_world()

**Output Snapshot**



**Program 6 queens using hill climbing**

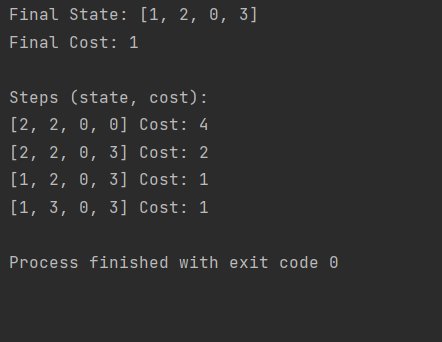
**Algorithm**

****

**Code**

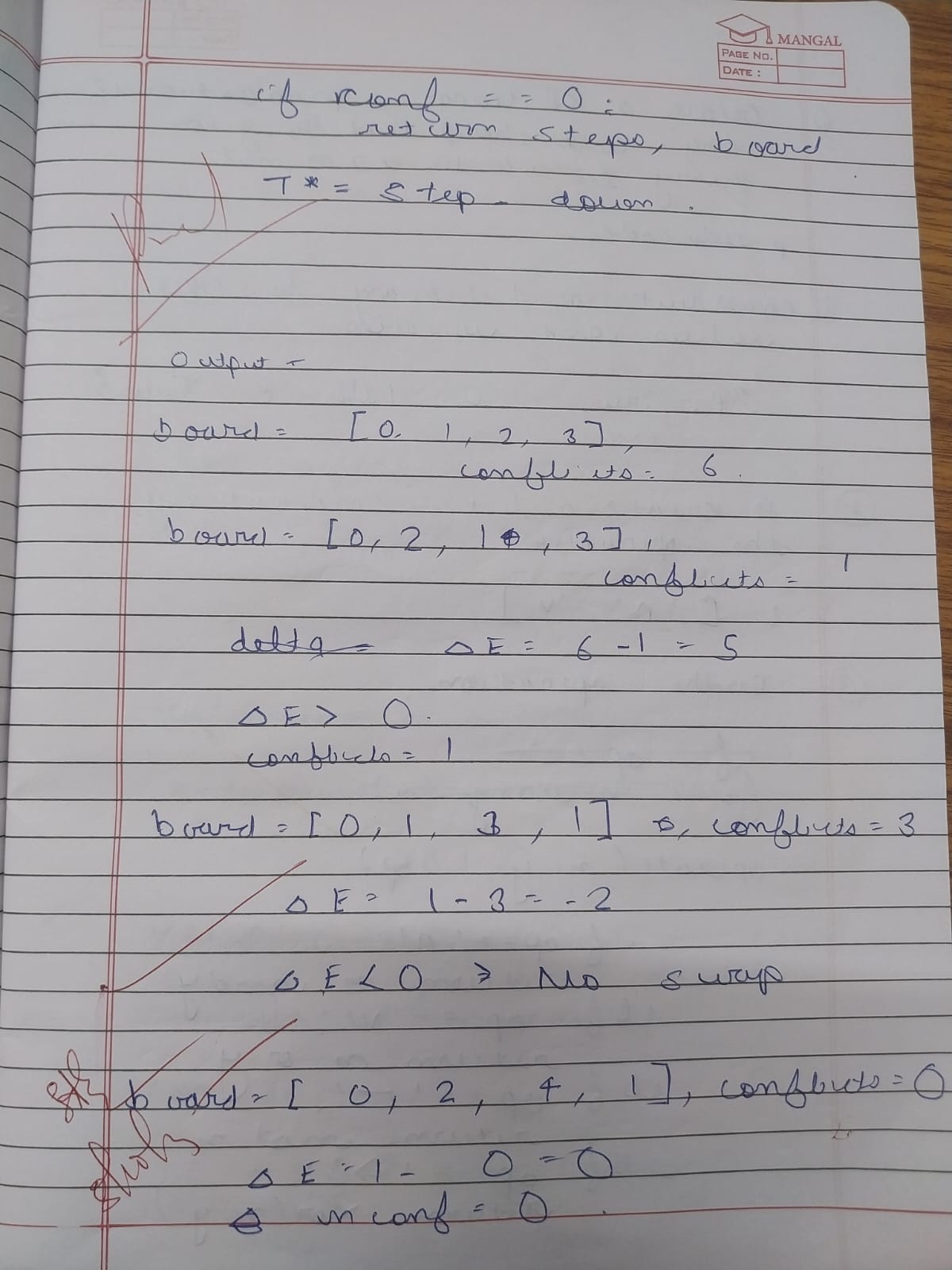
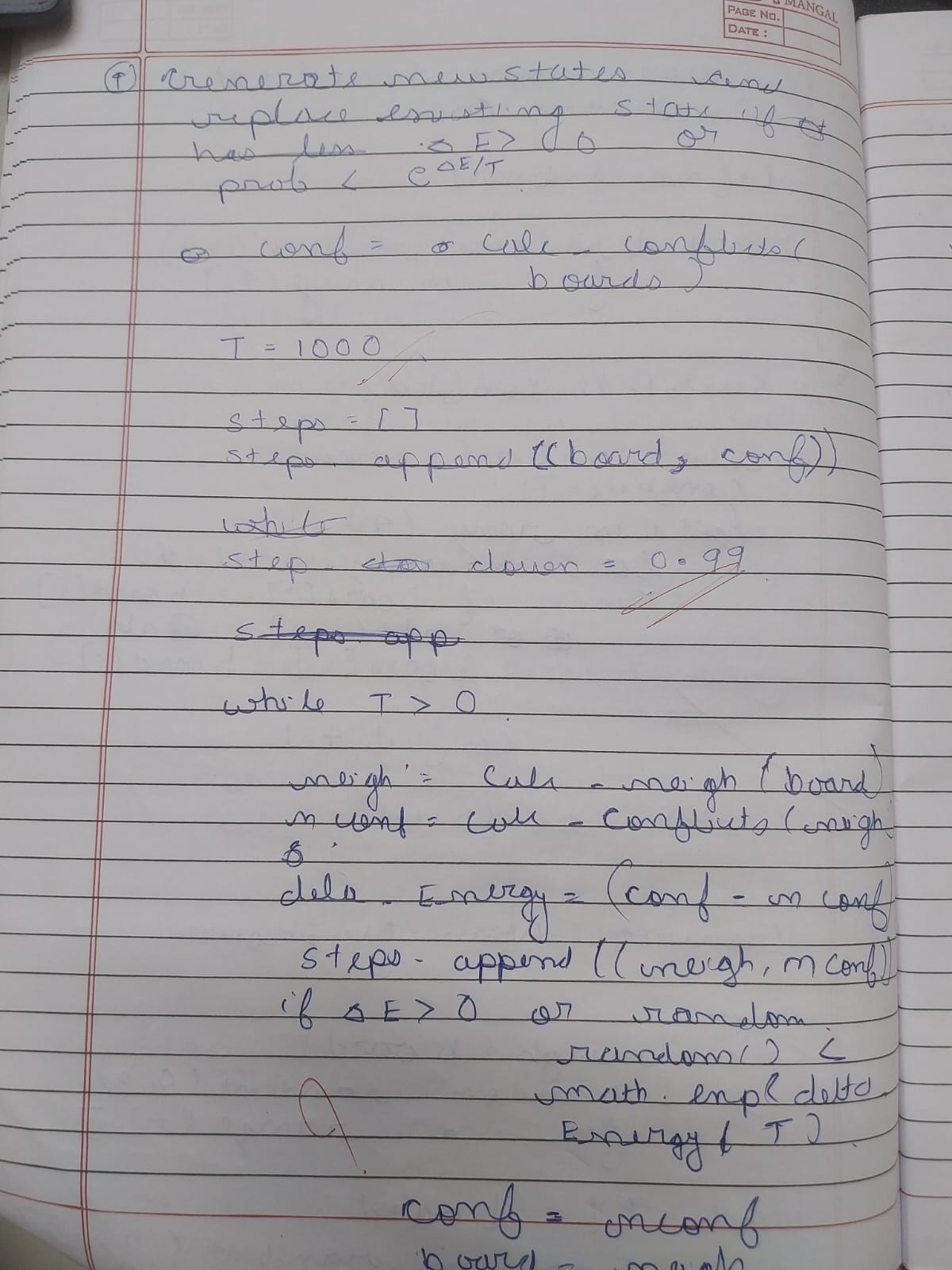
import random  
  
def calculate\_conflicts(state):  
 *"""Heuristic: number of pairs of queens attacking each other."""* conflicts = 0  
 n = len(state)  
 for i in range(n):  
 for j in range(i + 1, n):  
 if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):  
 conflicts += 1  
 return conflicts  
  
def generate\_neighbors(state):  
 *"""Generate all neighbor states by moving one queen in its column."""* neighbors = []  
 n = len(state)  
 for col in range(n):  
 for row in range(n):  
 if state[col] != row:  
 new\_state = state.copy()  
 new\_state[col] = row  
 neighbors.append(new\_state)  
 return neighbors  
  
def hill\_climbing(n=4):  
 # Initial random state  
 state = [random.randint(0, n - 1) for \_ in range(n)]  
 current\_cost = calculate\_conflicts(state)  
  
 steps = []  
 steps.append((state.copy(), current\_cost))  
  
 while True:  
 neighbors = generate\_neighbors(state)  
 best\_neighbor = min(neighbors, key=calculate\_conflicts)  
 best\_cost = calculate\_conflicts(best\_neighbor)  
  
 steps.append((best\_neighbor.copy(), best\_cost))  
  
 if best\_cost >= current\_cost: # Local optimum  
 break  
 state, current\_cost = best\_neighbor, best\_cost  
  
 return state, current\_cost, steps  
  
# Run for 4-Queens  
solution, cost, steps = hill\_climbing(4)  
  
print("Final State:", solution)  
print("Final Cost:", cost)  
print("\nSteps (state, cost):")  
for s, c in steps:  
 print(s, "Cost:", c)

**Output Snapshot**

****

**Program-07 4 queens using simulated annealing**

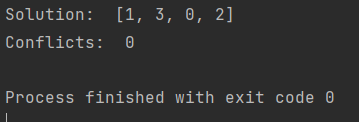
**Algorithm**

****

**Code**

**import random  
import math  
  
# Define the size of the board (4x4)  
N = 4  
  
  
# Generate a random state (a random arrangement of queens on the board)  
def random\_state():  
 return [random.randint(0, N - 1) for \_ in range(N)]  
  
  
# Calculate the number of conflicts (attacking pairs of queens)  
def calculate\_conflicts(state):  
 conflicts = 0  
 for i in range(N):  
 for j in range(i + 1, N):  
 if state[i] == state[j]: # Same column  
 conflicts += 1  
 if abs(state[i] - state[j]) == abs(i - j): # Same diagonal  
 conflicts += 1  
 return conflicts  
  
  
# Perform a random move (a small change to the state)  
def random\_move(state):  
 new\_state = state[:]  
 row = random.randint(0, N - 1)  
 new\_state[row] = random.randint(0, N - 1)  
 return new\_state  
  
  
# Simulated Annealing algorithm  
def simulated\_annealing():  
 current\_state = random\_state()  
 current\_temp = 1000 # Initial temperature  
 min\_temp = 1 # Minimum temperature to stop  
 cooling\_rate = 0.95 # Rate at which the temperature decreases  
  
 # Loop until the temperature is low enough  
 while current\_temp > min\_temp:  
 # Calculate the current number of conflicts  
 current\_conflicts = calculate\_conflicts(current\_state)  
  
 # If no conflicts, we've found a solution  
 if current\_conflicts == 0:  
 return current\_state  
  
 # Generate a new candidate by making a random move  
 new\_state = random\_move(current\_state)  
 new\_conflicts = calculate\_conflicts(new\_state)  
  
 # If the new state is better or if we accept it probabilistically  
 if new\_conflicts < current\_conflicts or random.random() < math.exp(  
 (current\_conflicts - new\_conflicts) / current\_temp):  
 current\_state = new\_state  
  
 # Cool down the temperature  
 current\_temp \*= cooling\_rate  
  
 # Return the best state found  
 return current\_state  
  
  
# Run the simulated annealing algorithm  
solution = simulated\_annealing()  
  
# Display the solution  
print("Solution: ", solution)  
print("Conflicts: ", calculate\_conflicts(solution))**

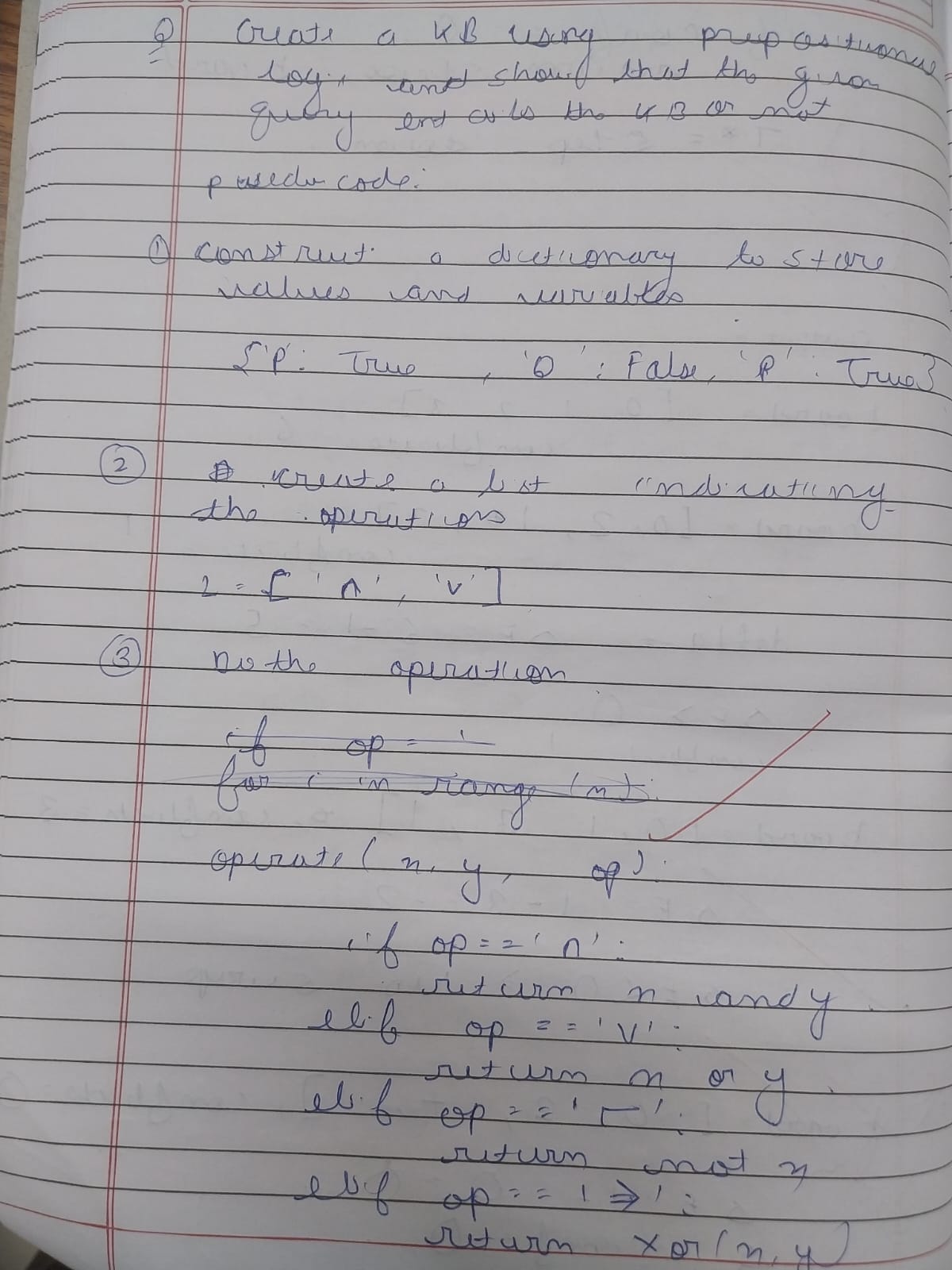
**OUTPUT**

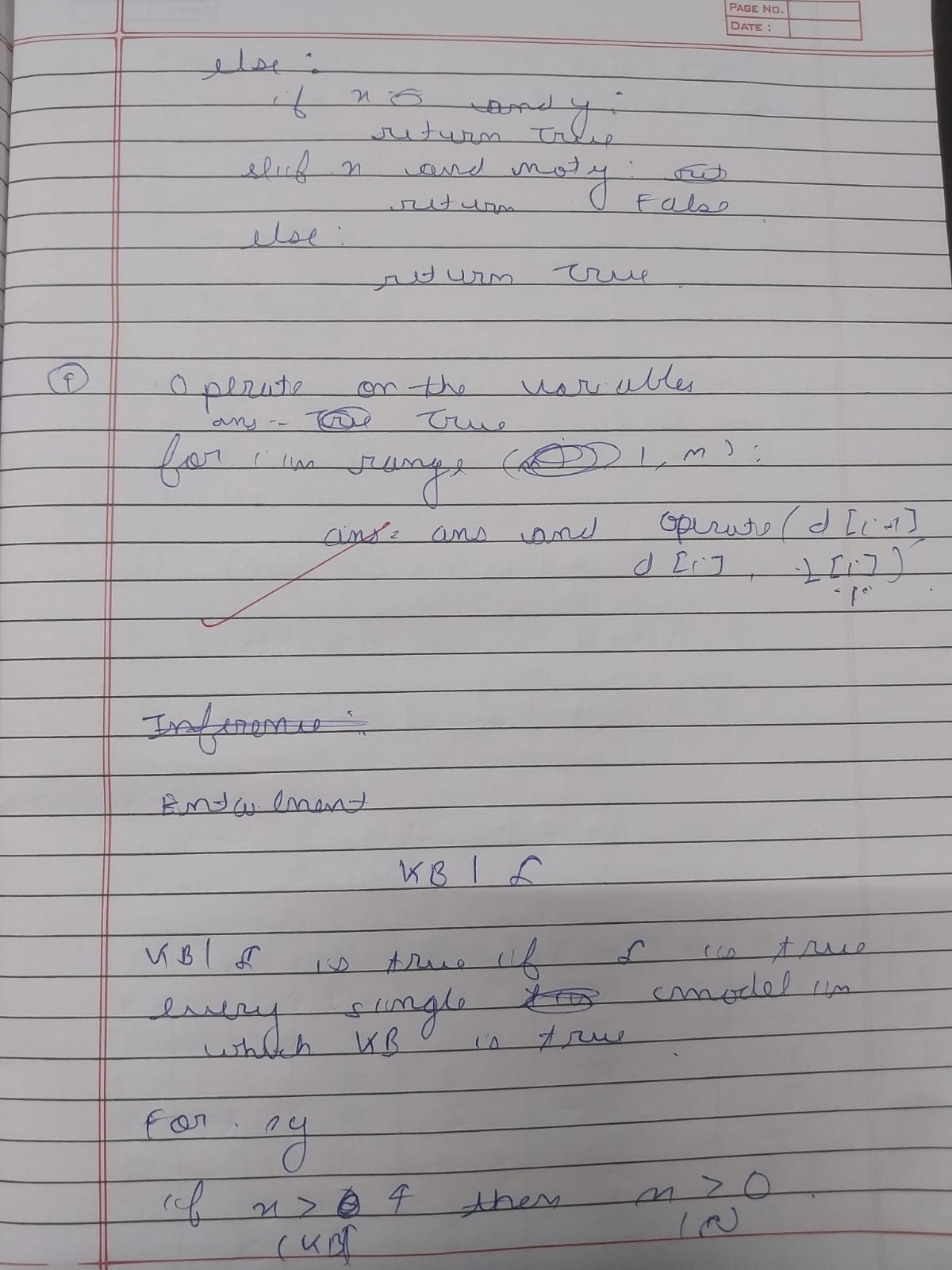
****

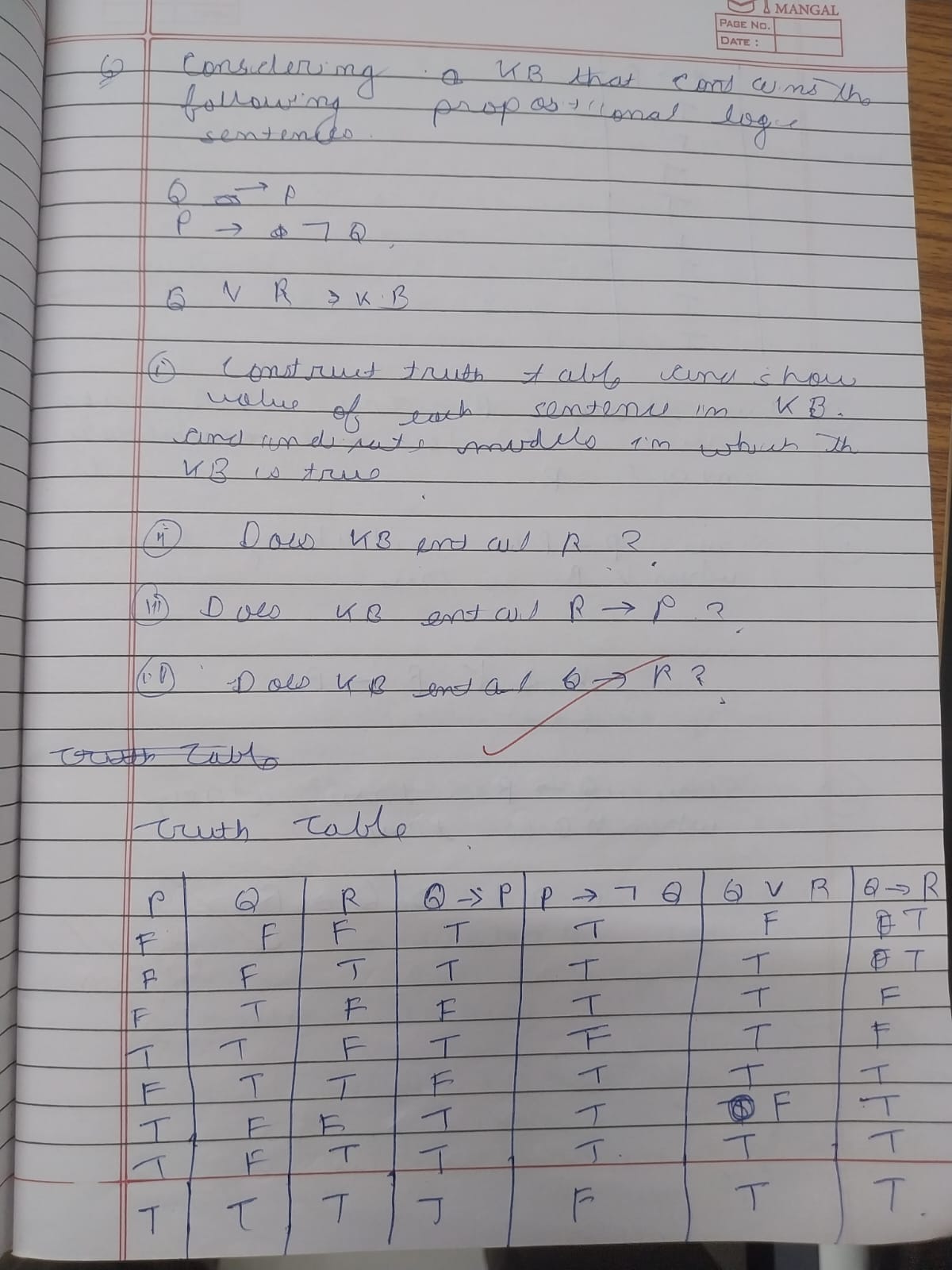
33

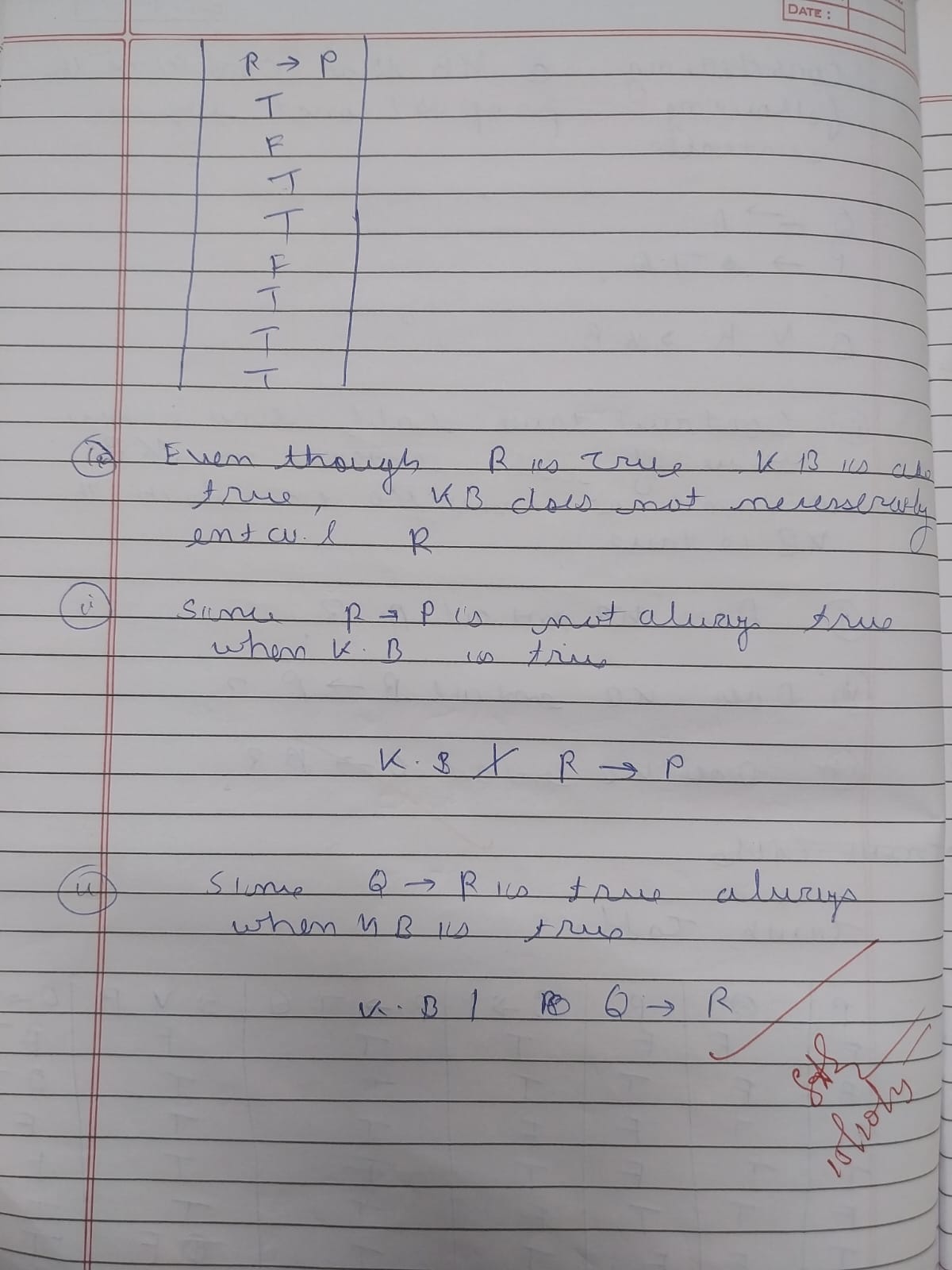
**Program-08 Knowledge Base -Propositional Logic**

**Algorithm**

****

****

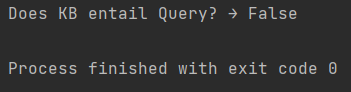
****

****

Code

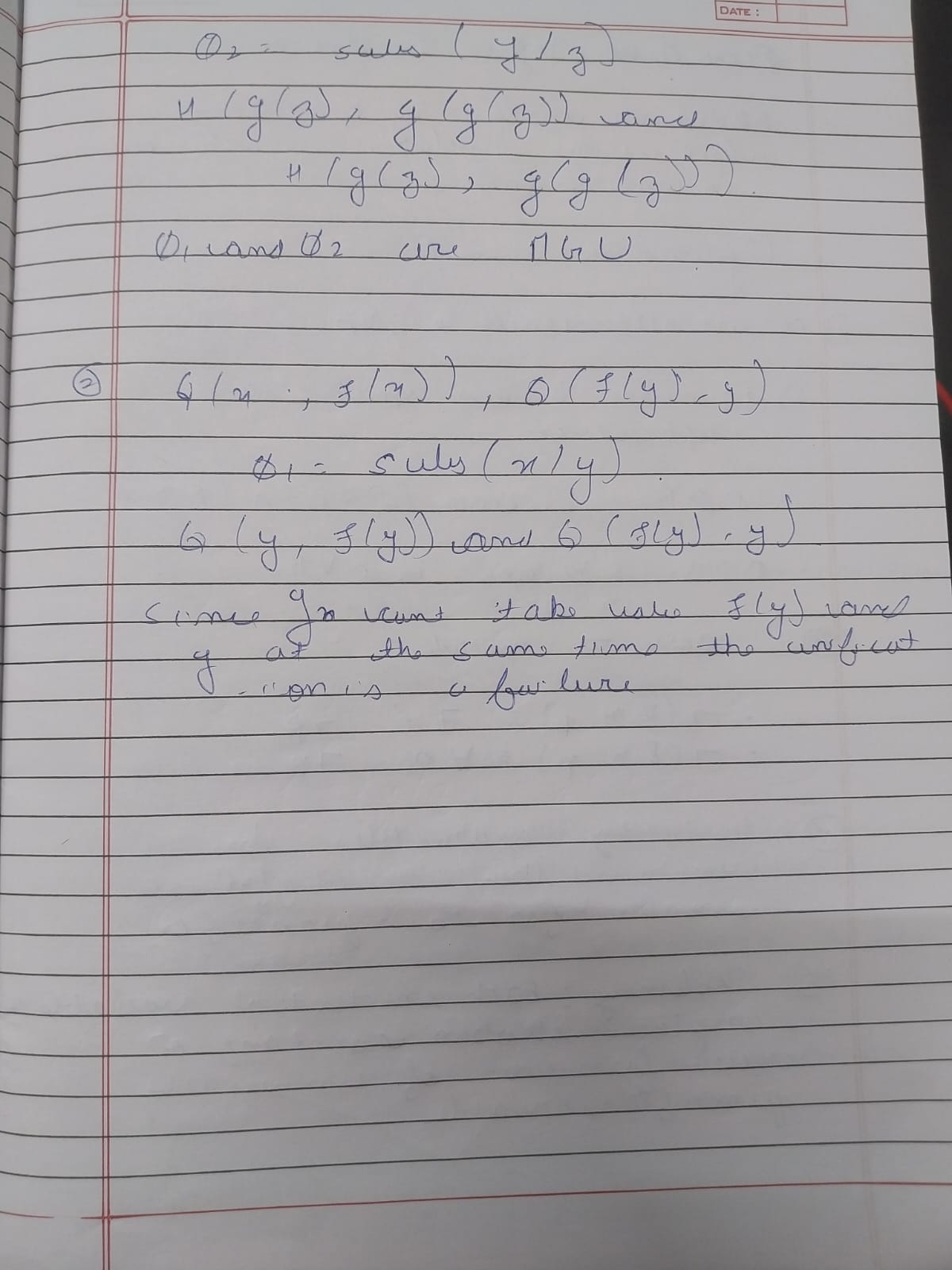
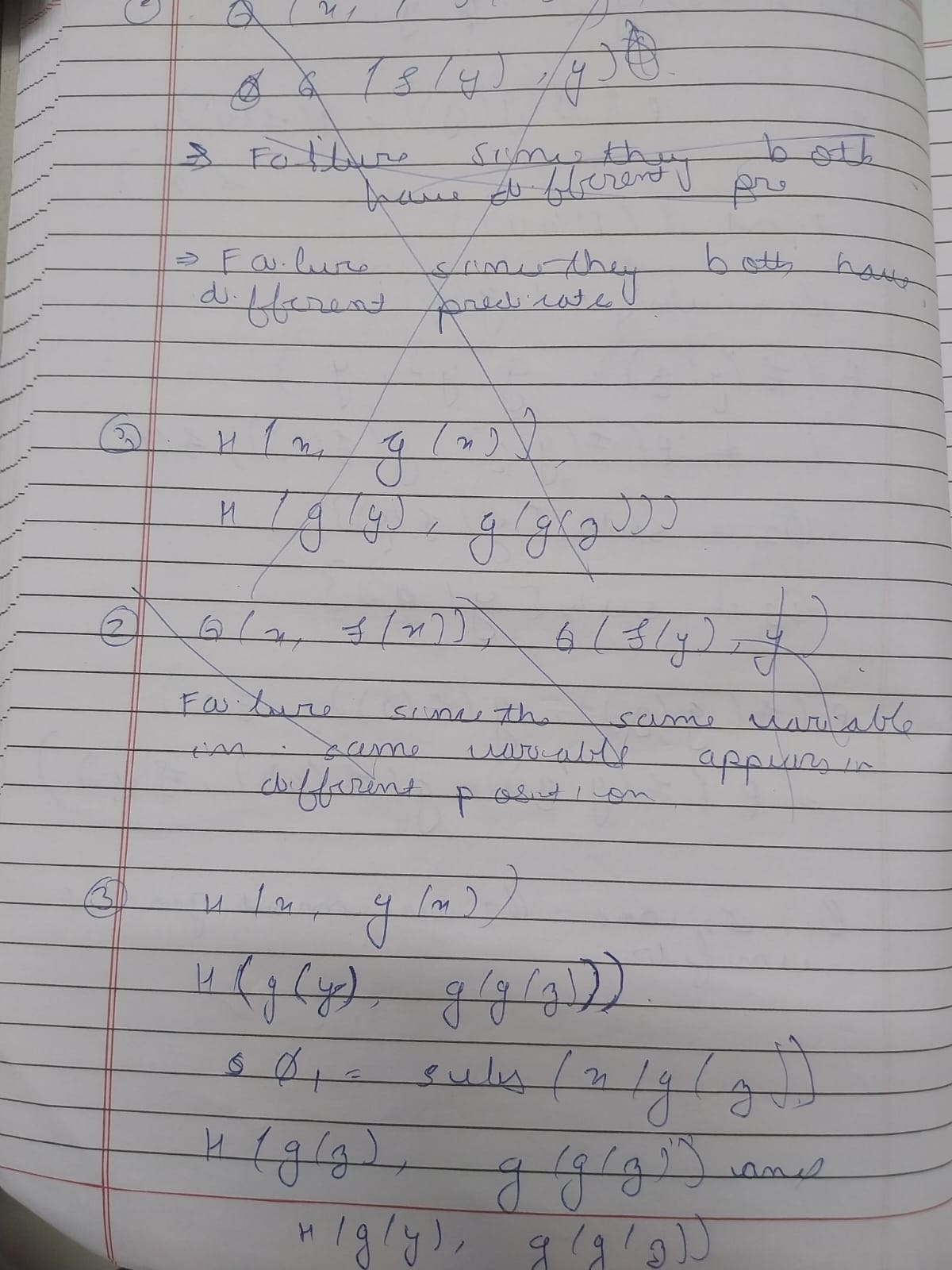
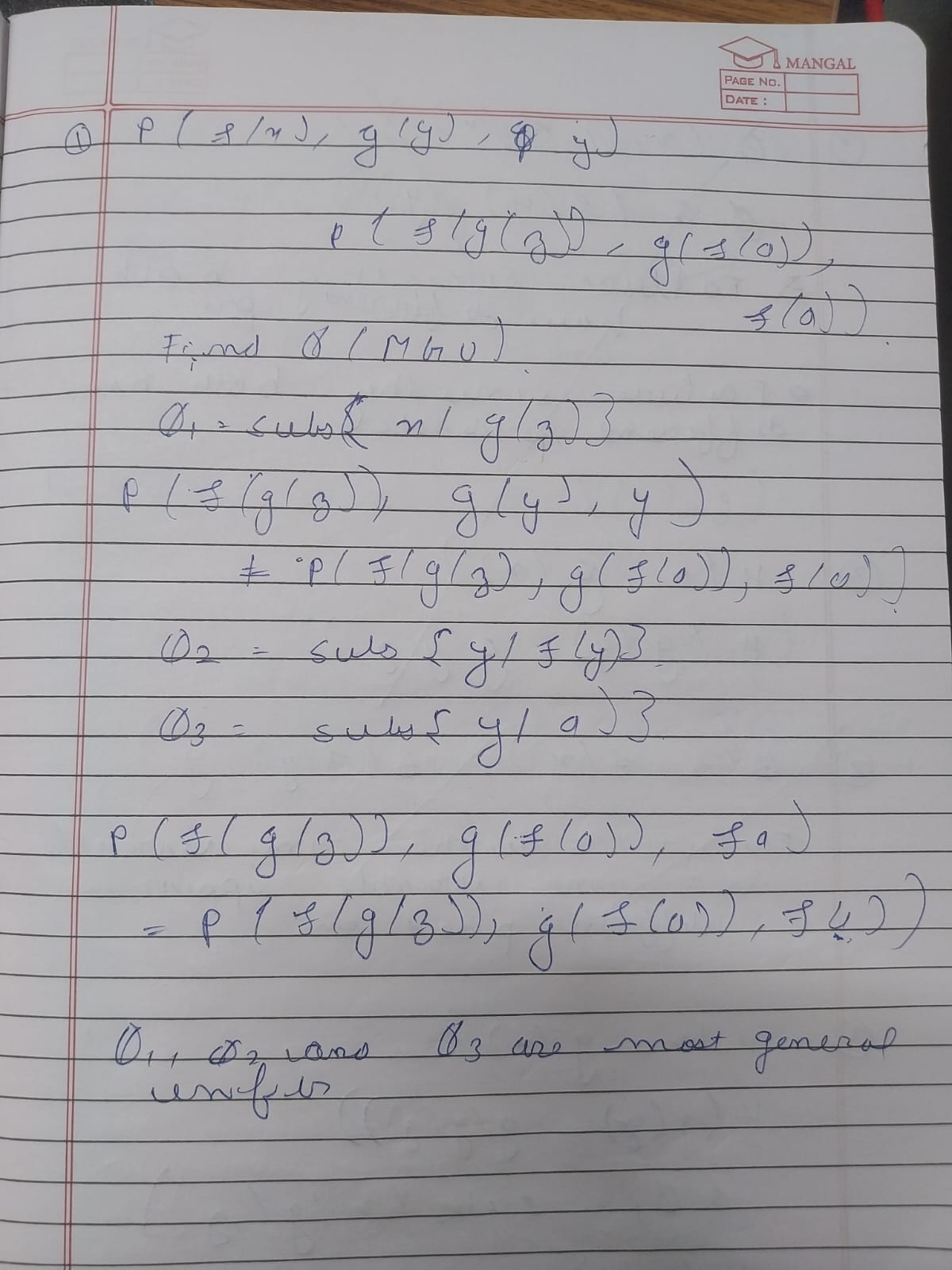
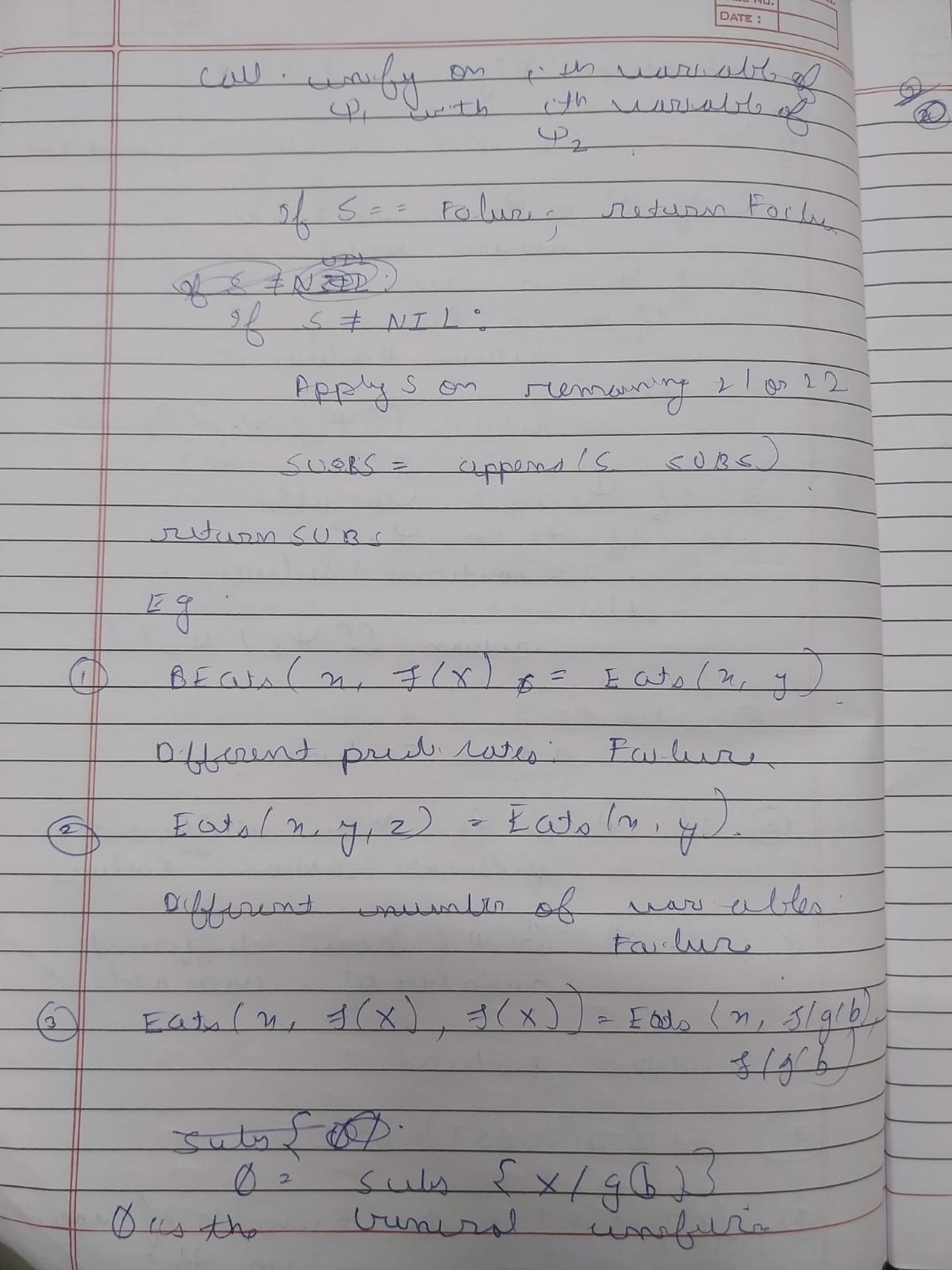
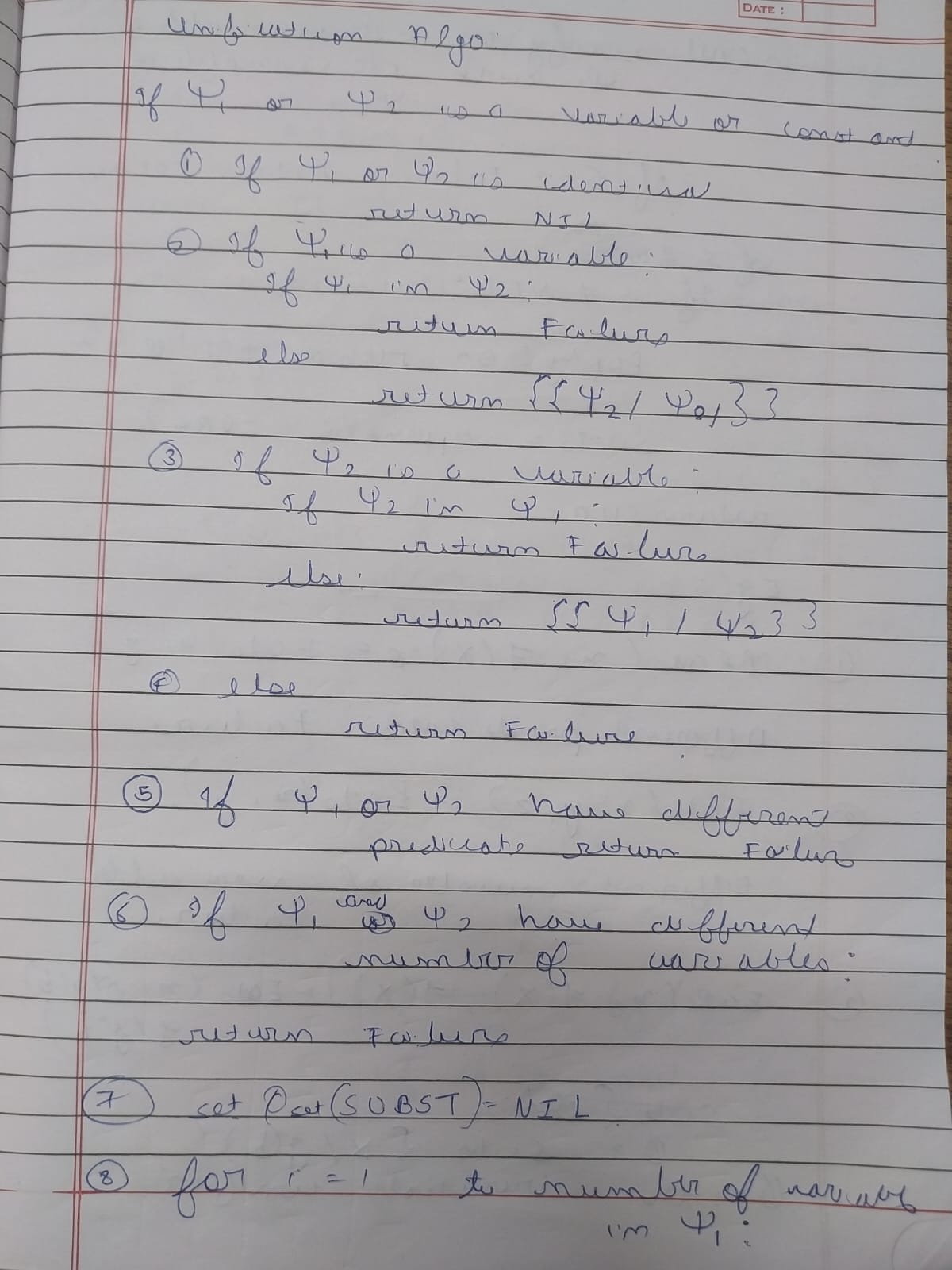
import itertools  
  
  
# Evaluate a propositional logic sentence in a given model (dictionary of symbol → True/False)  
def pl\_true(expr, model):  
 if isinstance(expr, str): # atomic symbol  
 return model[expr]  
 op = expr[0]  
 if op == 'not':  
 return not pl\_true(expr[1], model)  
 elif op == '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 == 'implies':  
 return (not pl\_true(expr[1], model)) or pl\_true(expr[2], model)  
 elif op == 'iff':  
 return pl\_true(expr[1], model) == pl\_true(expr[2], model)  
 else:  
 raise ValueError("Unknown operator: " + op)  
  
  
# The truth table enumeration function  
def tt\_entails(kb, query, symbols):  
 for values in itertools.product([True, False], repeat=len(symbols)):  
 model = dict(zip(symbols, values))  
 if pl\_true(kb, model):  
 if not pl\_true(query, model):  
 return False # Found a counterexample  
 return True  
  
# Example Knowledge Base and Query  
# Example: (P ∧ Q) → R, and (P ∧ Q) is true, does R follow?  
  
  
symbols = ['P', 'Q', 'R']  
  
# Knowledge Base: (P ∧ Q) → R  
kb = ('implies', ('and', 'P', 'Q'), 'R')  
  
# Query: R  
query = 'R'  
  
# Check entailment  
result = tt\_entails(kb, query, symbols)  
  
print("Does KB entail Query? →", result)

Output Snapshot

****

**Program-09 Unification in first order logic**

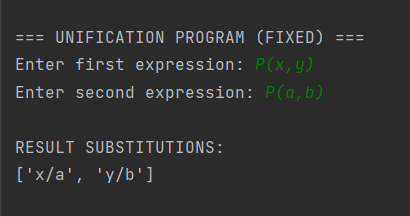
**Algorithm**

****

**Code**

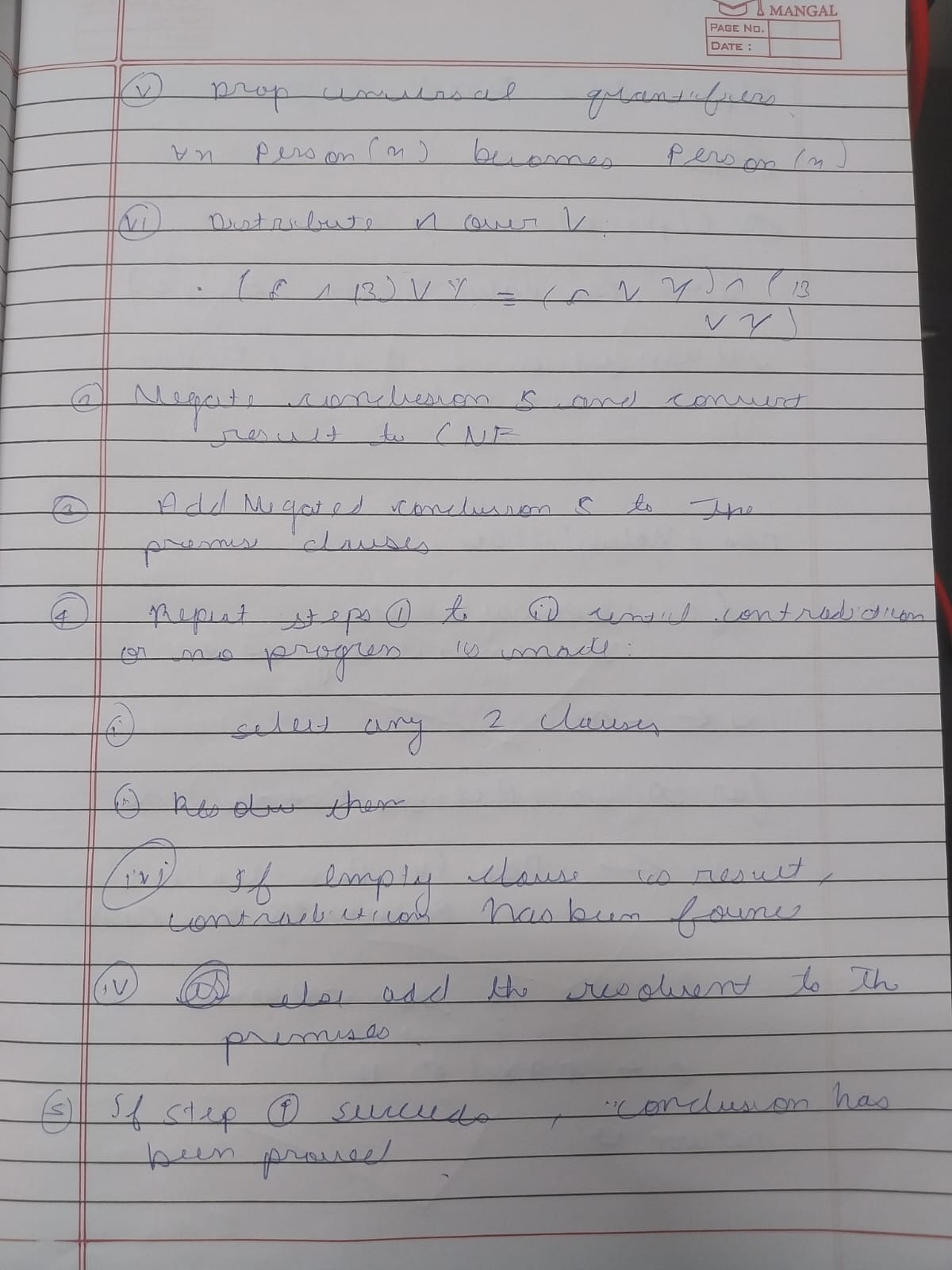
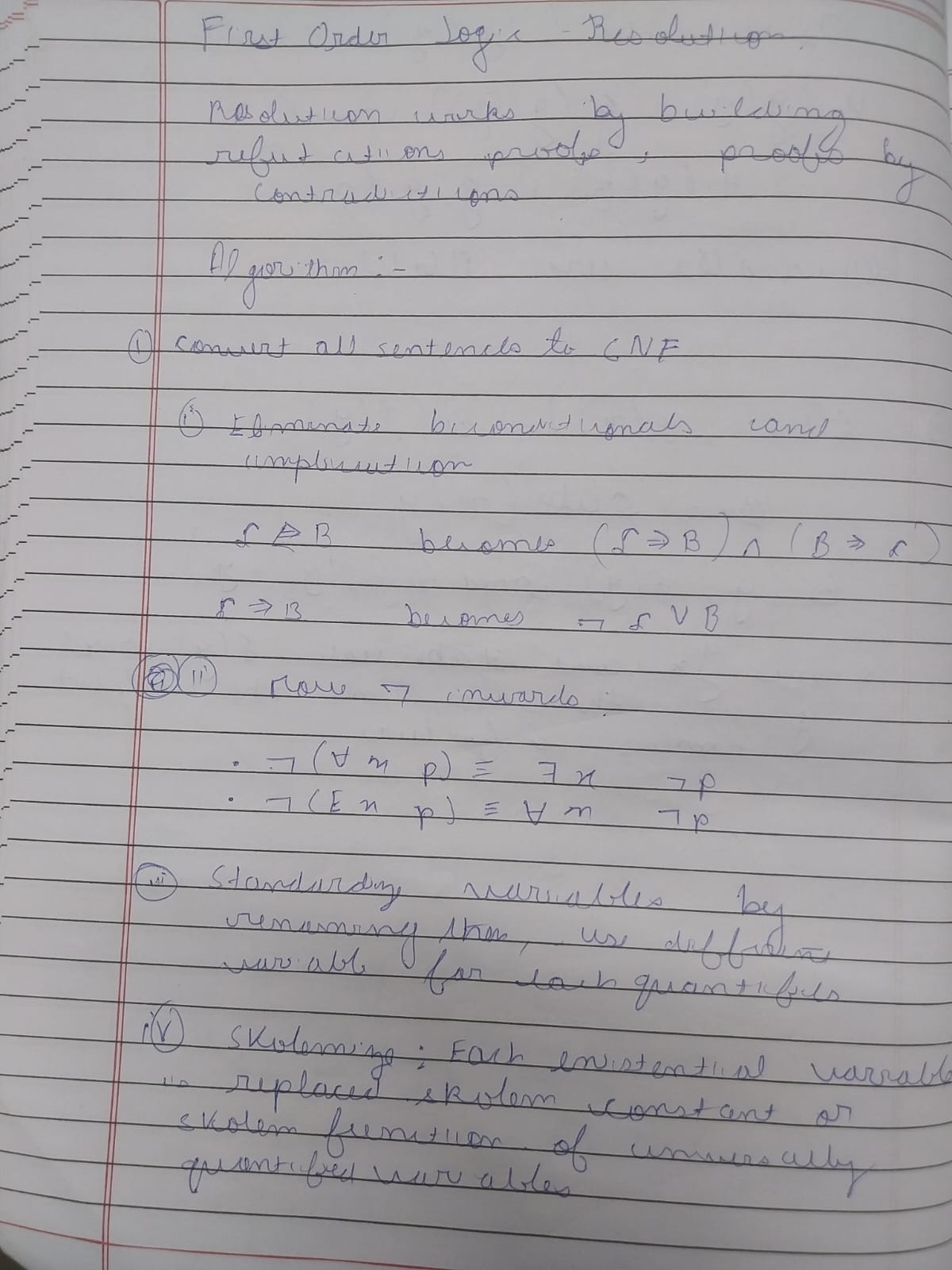
import re  
  
def getAttributes(expression):  
 inside = expression[expression.index("(")+1 : expression.rindex(")")]  
 return inside.split(",")  
  
def getInitialPredicate(expression):  
 return expression.split("(")[0]  
  
def isConstant(x):  
 return len(x) == 1 and x.isupper()  
  
def isVariable(x):  
 return len(x) == 1 and x.islower()  
  
def replaceAttributes(exp, old, new):  
 predicate = getInitialPredicate(exp)  
 attributes = getAttributes(exp)  
 attributes = [new if x == old else x for x in attributes]  
 return f"{predicate}({','.join(attributes)})"  
  
def apply(exp, substitutions):  
 for (new, old) in substitutions:  
 exp = replaceAttributes(exp, old, new)  
 return exp  
  
def checkOccurs(var, exp):  
 return var in exp  
  
def getFirstPart(expression):  
 return getAttributes(expression)[0]  
  
def getRemainingPart(expression):  
 predicate = getInitialPredicate(expression)  
 attributes = getAttributes(expression)[1:]  
 return f"{predicate}({','.join(attributes)})"  
  
def unify(exp1, exp2):  
 # Rule 1: identical  
 if exp1 == exp2:  
 return []  
  
 # Rule 2: constants  
 if isConstant(exp1) and isConstant(exp2):  
 if exp1 != exp2:  
 print(f"{exp1} and {exp2} are different constants → FAIL")  
 return []  
 return []  
  
 # Rule 3: variable cases  
 if isVariable(exp1):  
 return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []  
  
 if isVariable(exp2):  
 return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []  
  
 # Rule 4: predicates must match  
 if getInitialPredicate(exp1) != getInitialPredicate(exp2):  
 print("Predicates do not match → FAIL")  
 return []  
  
 args1 = getAttributes(exp1)  
 args2 = getAttributes(exp2)  
  
 if len(args1) != len(args2):  
 print("Argument lengths do not match → FAIL")  
 return []  
  
 # Recursive unification  
 head1 = args1[0]  
 head2 = args2[0]  
  
 s1 = unify(head1, head2)  
 if s1 == [] and head1 != head2:  
 return []  
  
 if len(args1) == 1:  
 return s1  
  
 t1 = apply(getRemainingPart(exp1), s1)  
 t2 = apply(getRemainingPart(exp2), s1)  
  
 s2 = unify(t1, t2)  
 if s2 == [] and t1 != t2:  
 return []  
  
 return s1 + s2  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("\n=== UNIFICATION PROGRAM (FIXED) ===")  
 e1 = input("Enter first expression: ").strip()  
 e2 = input("Enter second expression: ").strip()  
  
 substitutions = unify(e1, e2)  
  
 print("\nRESULT SUBSTITUTIONS:")  
 if substitutions:  
 print([f"{var}/{val}" for (val, var) in substitutions])  
 else:  
 print("UNIFICATION FAILED")

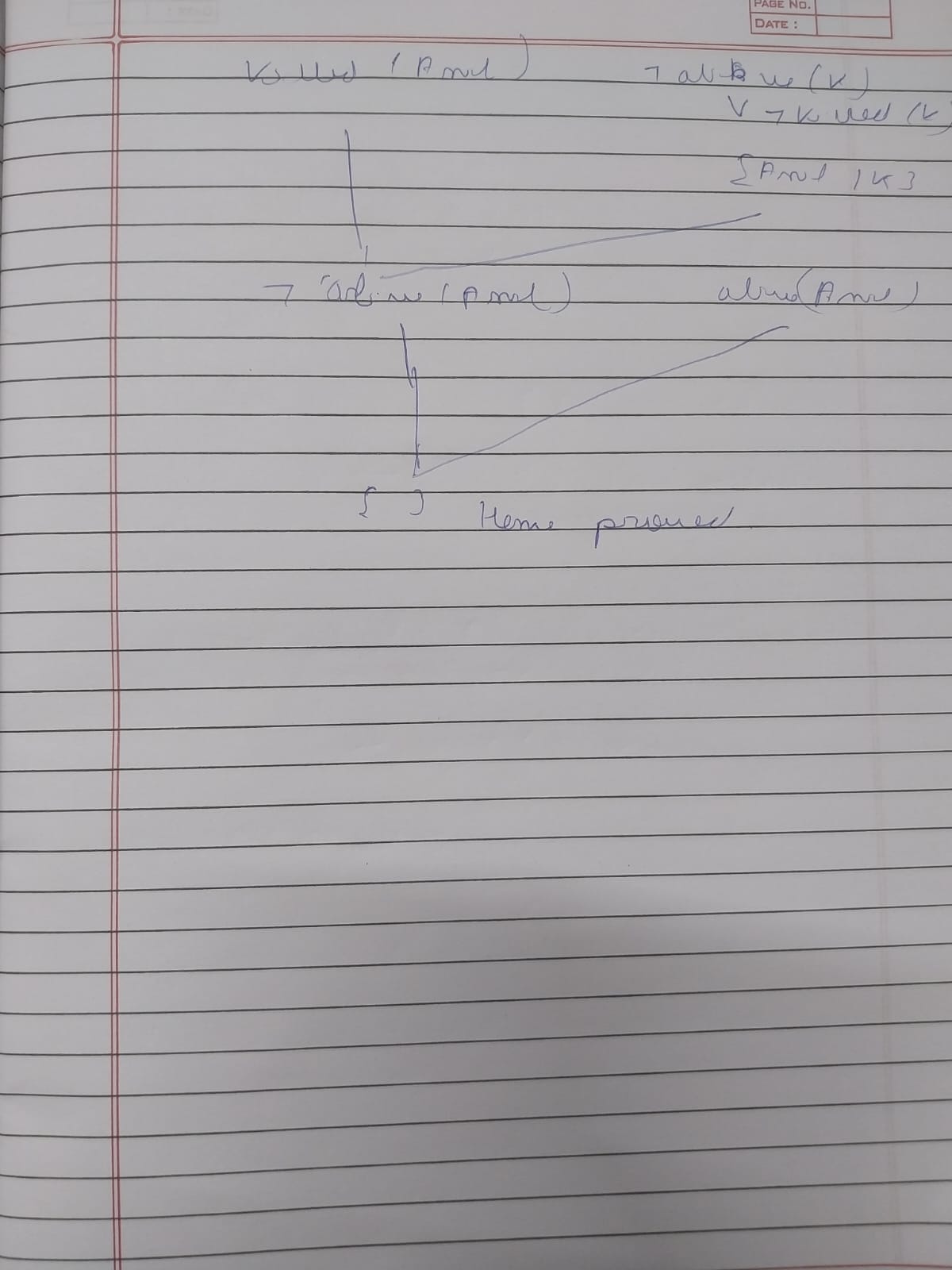
**Output Snapshot**

****

**Program-10 Resolution Program**

**Algorithm:**

****



**Code**

from itertools import combinations

# Utility: negate a literal

def negate(literal):

if literal.startswith('~'):

return literal[1:]

else:

return '~' + literal

# Apply resolution on two clauses

def resolve(ci, cj):

resolvents = set()

for di in ci:

for dj in cj:

if di == negate(dj):

new\_clause = (ci - {di}) | (cj - {dj})

resolvents.add(frozenset(new\_clause))

return resolvents

def fol\_resolution(KB, query):

clauses = set(KB)

clauses.add(frozenset([negate(query)])) # add negation of query

new = set()

while True:

pairs = list(combinations(clauses, 2))

for (ci, cj) in pairs:

resolvents = resolve(ci, cj)

if frozenset() in resolvents:

return True # empty clause found → proved

new = new.union(resolvents)

if new.issubset(clauses):

return False # no new clauses → cannot prove

clauses = clauses.union(new)

# ------------------------

# Knowledge Base in CNF

# ------------------------

KB = [

frozenset(['~Food(x)', 'Likes(John,x)']), # John likes all food

frozenset(['Food(Apple)']),

frozenset(['Food(Vegetables)']),

frozenset(['~Eats(x,y)', 'Killed(x)', 'Food(y)']),

frozenset(['Eats(Anil,Peanuts)']),

frozenset(['Alive(Anil)']),

frozenset(['~Eats(Anil,x)', 'Eats(Harry,x)']),

frozenset(['~Alive(x)', '~Killed(x)']),

frozenset(['Killed(x)', 'Alive(x)'])

]

query = 'Likes(John,Peanuts)'

# Run the resolution

if fol\_resolution(KB, query):

print("Proved:", query)

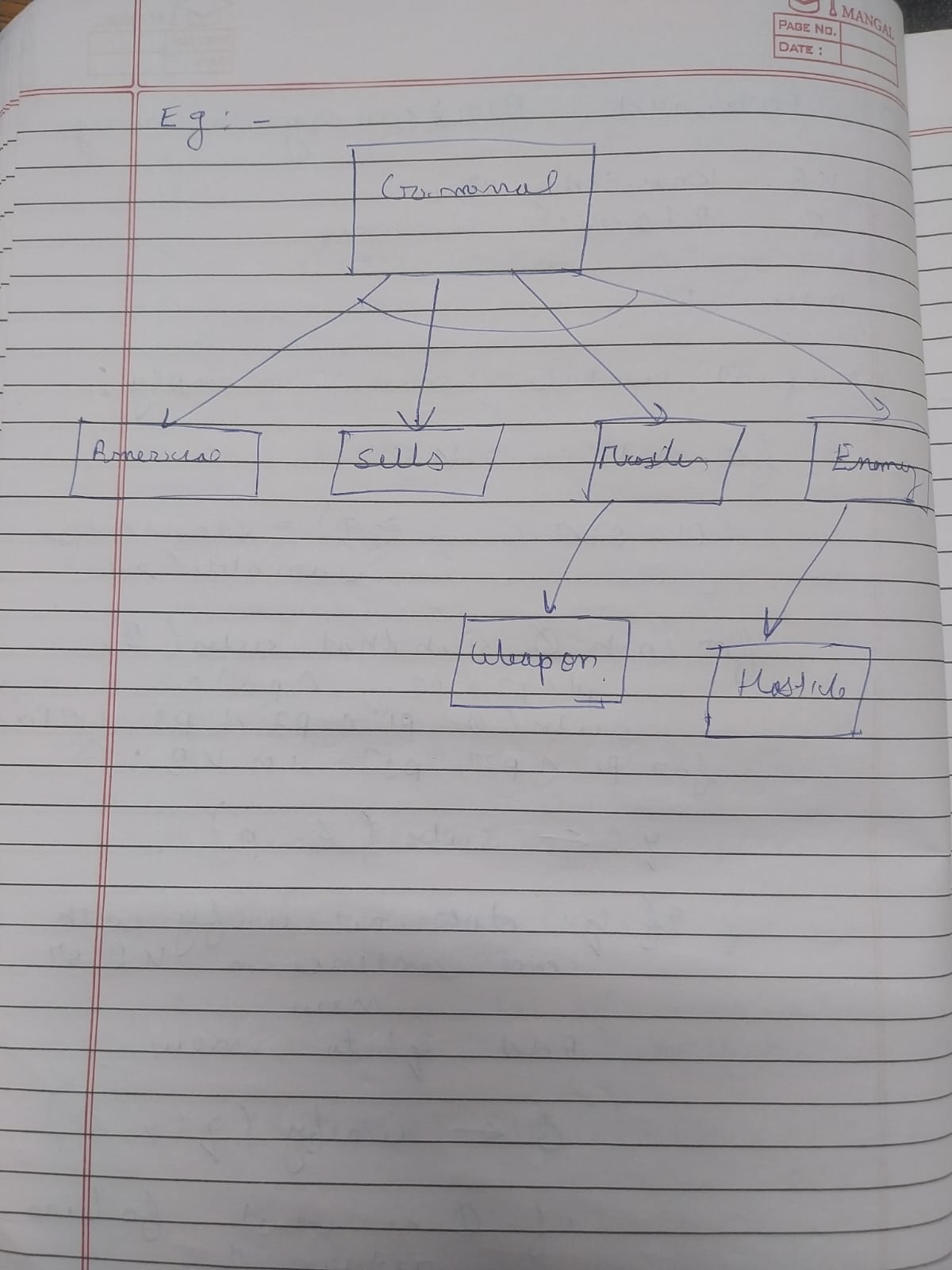
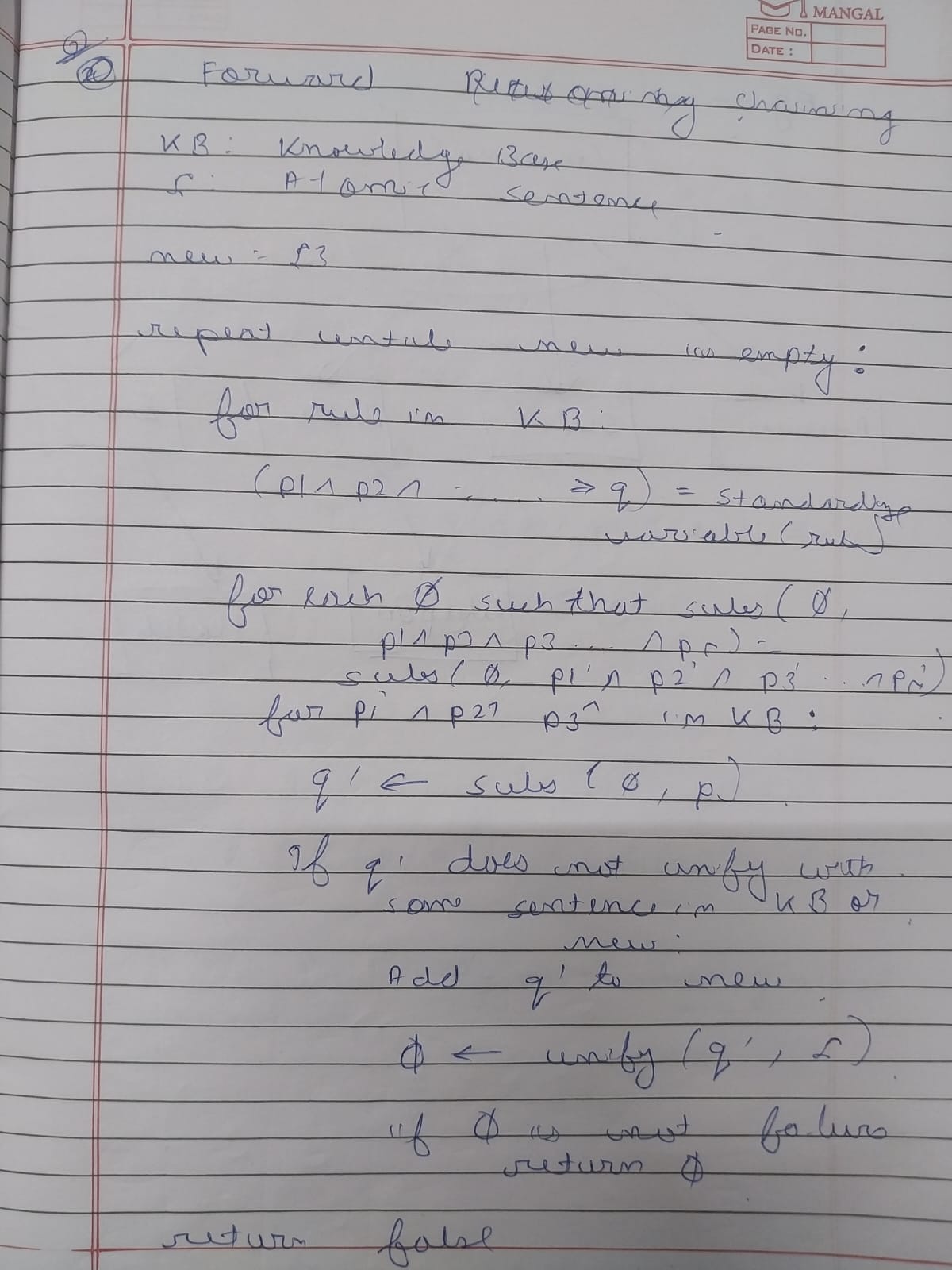
else:

print("Invalid — cannot be proved")

**OutputSnapshot**

****

**Program 11: Forward Chaining inference**



Code-

# Forward Chaining in First-Order Logic (FOL)

# Example: Prove "Criminal(Robert)"

# Knowledge Base (KB)

KB = [

# Rule 1: It is a crime for an American to sell weapons to hostile nations

{"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"], "then": "Criminal(p)"},

# Rule 2: Country A owns some missiles (T1)

{"fact": "Owns(A, T1)"},

{"fact": "Missile(T1)"},

# Rule 3: Missiles are weapons

{"if": ["Missile(x)"], "then": "Weapon(x)"},

# Rule 4: All missiles owned by A were sold by Robert

{"if": ["Missile(x)", "Owns(A, x)"], "then": "Sells(Robert, x, A)"},

# Rule 5: Enemies of America are hostile

{"if": ["Enemy(x, America)"], "then": "Hostile(x)"},

# Rule 6: Robert is an American

{"fact": "American(Robert)"},

# Rule 7: Country A is an enemy of America

{"fact": "Enemy(A, America)"}

]

# Function to extract all current known facts

def get\_facts(kb):

return {rule["fact"] for rule in kb if "fact" in rule}

# Function to perform variable substitution

def substitute(expr, var, val):

return expr.replace(f"({var})", f"({val})").replace(f"{var},", f"{val},").replace(f",{var})", f",{val})")

# Forward chaining algorithm

def forward\_chain(kb, query):

inferred = set()

facts = get\_facts(kb)

new\_inference = True

print("Initial Facts:", facts)

print("Goal:", query)

print("\n--- Forward Chaining Process ---")

while new\_inference:

new\_inference = False

for rule in kb:

if "if" in rule:

conditions = rule["if"]

result = rule["then"]

# Check if all conditions are satisfied

satisfied = True

temp\_result = result

substitution = {}

for cond in conditions:

matched = False

for fact in facts:

if cond.split("(")[0] == fact.split("(")[0]:

# Try variable substitution

c\_args = cond[cond.find("(") + 1:-1].split(",")

f\_args = fact[fact.find("(") + 1:-1].split(",")

if len(c\_args) == len(f\_args):

for i in range(len(c\_args)):

if c\_args[i].islower():

substitution[c\_args[i]] = f\_args[i]

elif c\_args[i] != f\_args[i]:

break

else:

matched = True

break

if not matched:

satisfied = False

break

if satisfied:

for var, val in substitution.items():

temp\_result = substitute(temp\_result, var, val)

if temp\_result not in facts:

facts.add(temp\_result)

inferred.add(temp\_result)

new\_inference = True

print(f"Inferred: {temp\_result}")

if temp\_result == query:

print("\n✅ Goal Reached!")

return True

print("\n❌ Goal Not Reached.")

return False

# Main Execution

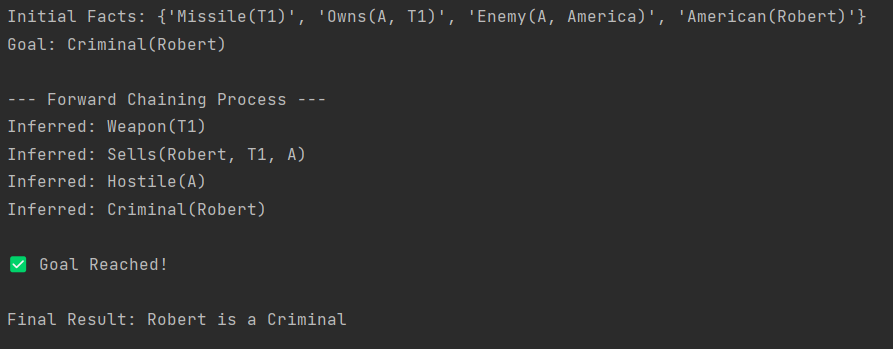
if \_\_name\_\_ == "\_\_main\_\_":

query = "Criminal(Robert)"

result = forward\_chain(KB, query)

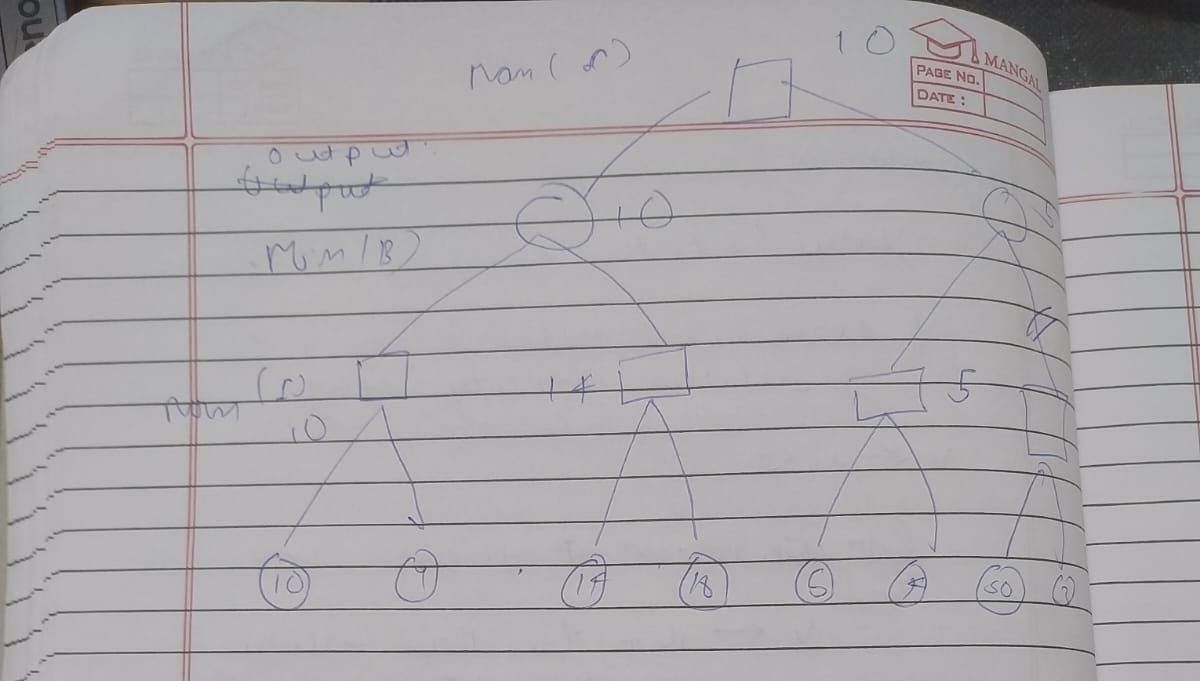
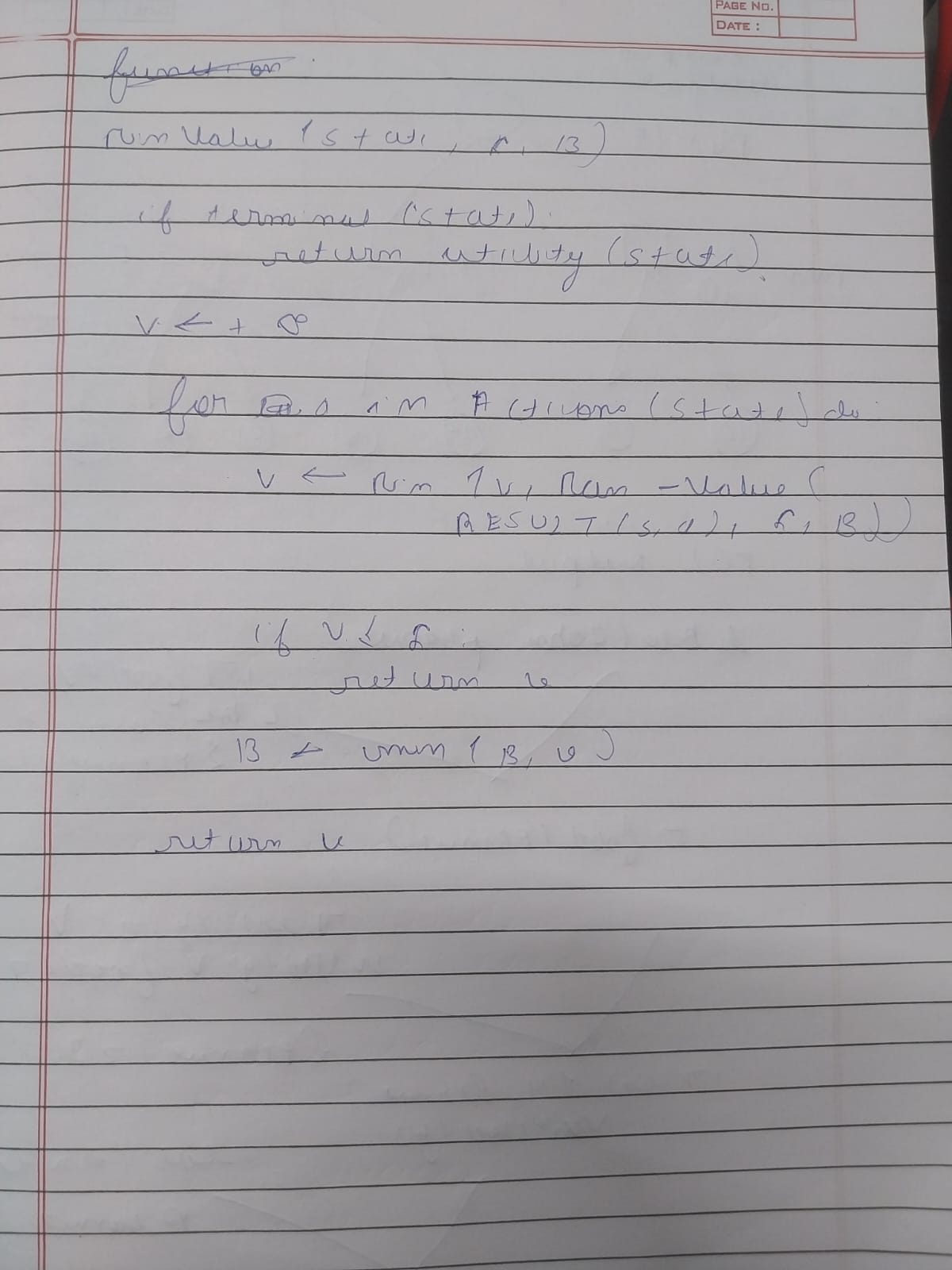
print("\nFinal Result:", "Robert is a Criminal" if result else "Cannot prove Robert is Criminal")

Output-

****

**Program 12 :Alpha Beta Pruning**

**Algorithm-**

****

**Code-**

import math

# Alpha-Beta Pruning Function

def alpha\_beta\_pruning(depth, node\_index, is\_max, scores, alpha, beta, max\_depth, path):

# Base case: return leaf node value

if depth == max\_depth:

path.append(node\_index)

return scores[node\_index]

if is\_max:

best = -math.inf

for i in range(2): # each node has 2 children

val = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, False, scores, alpha, beta, max\_depth, path)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

print(f"Pruned at depth {depth} (MAX node): alpha={alpha}, beta={beta}")

break # Beta cut-off

return best

else:

best = math.inf

for i in range(2):

val = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, True, scores, alpha, beta, max\_depth, path)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

print(f"Pruned at depth {depth} (MIN node): alpha={alpha}, beta={beta}")

break # Alpha cut-off

return best

# -----------------------------

# Example tree for demonstration

# -----------------------------

# Binary tree leaves (depth = 3)

scores = [3, 5, 6, 9, 1, 2, 0, -1]

max\_depth = int(math.log(len(scores), 2))

path = []

print("Leaf node values:", scores)

print("Applying Alpha-Beta Pruning...\n")

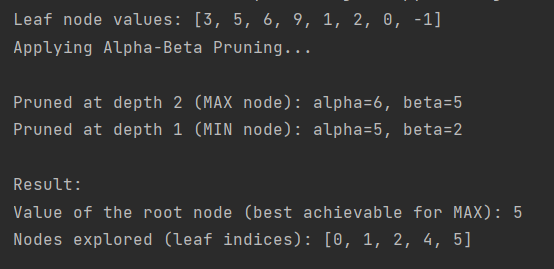
root\_value = alpha\_beta\_pruning(0, 0, True, scores, -math.inf, math.inf, max\_depth, path)

print("\nResult:")

print(f"Value of the root node (best achievable for MAX): {root\_value}")

print(f"Nodes explored (leaf indices): {path}")

**Output -**

****