DAY-1 LAB PROGRAMS

1.8-PUZZLE PROBLEM

import heapq

def manhattan\_distance(state, goal\_state):

distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

x, y = divmod(state[i][j] - 1, 3)

distance += abs(x - i) + abs(y - j)

return distance

def is\_solvable(state):

flat\_state = [num for row in state for num in row if num != 0]

inversions = 0

for i in range(len(flat\_state)):

for j in range(i + 1, len(flat\_state)):

if flat\_state[i] > flat\_state[j]:

inversions += 1

return inversions % 2 == 0

def get\_neighbors(state):

neighbors = []

zero\_row, zero\_col = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)

for move in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

new\_row, new\_col = zero\_row + move[0], zero\_col + move[1]

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_state = [row[:] for row in state]

new\_state[zero\_row][zero\_col], new\_state[new\_row][new\_col] = new\_state[new\_row][new\_col], \

new\_state[zero\_row][zero\_col]

neighbors.append(new\_state)

return neighbors

def solve\_8\_puzzle(start\_state, goal\_state):

if not is\_solvable(start\_state):

print("The given puzzle is not solvable.")

return None

priority\_queue = []

heapq.heappush(priority\_queue, (manhattan\_distance(start\_state, goal\_state), start\_state, []))

visited = set()

visited.add(tuple(tuple(row) for row in start\_state))

while priority\_queue:

cost, current\_state, path = heapq.heappop(priority\_queue)

path = path + [current\_state]

if current\_state == goal\_state:

return path

for neighbor in get\_neighbors(current\_state):

if tuple(tuple(row) for row in neighbor) not in visited:

heapq.heappush(priority\_queue, (len(path) + manhattan\_distance(neighbor, goal\_state), neighbor, path))

visited.add(tuple(tuple(row) for row in neighbor))

return None

start\_state = [

[1, 2, 3],

[4, 0, 6],

[7, 5, 8]

]

goal\_state = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 0]

]

solution\_path = solve\_8\_puzzle(start\_state, goal\_state)

if solution\_path:

print("Solution found!")

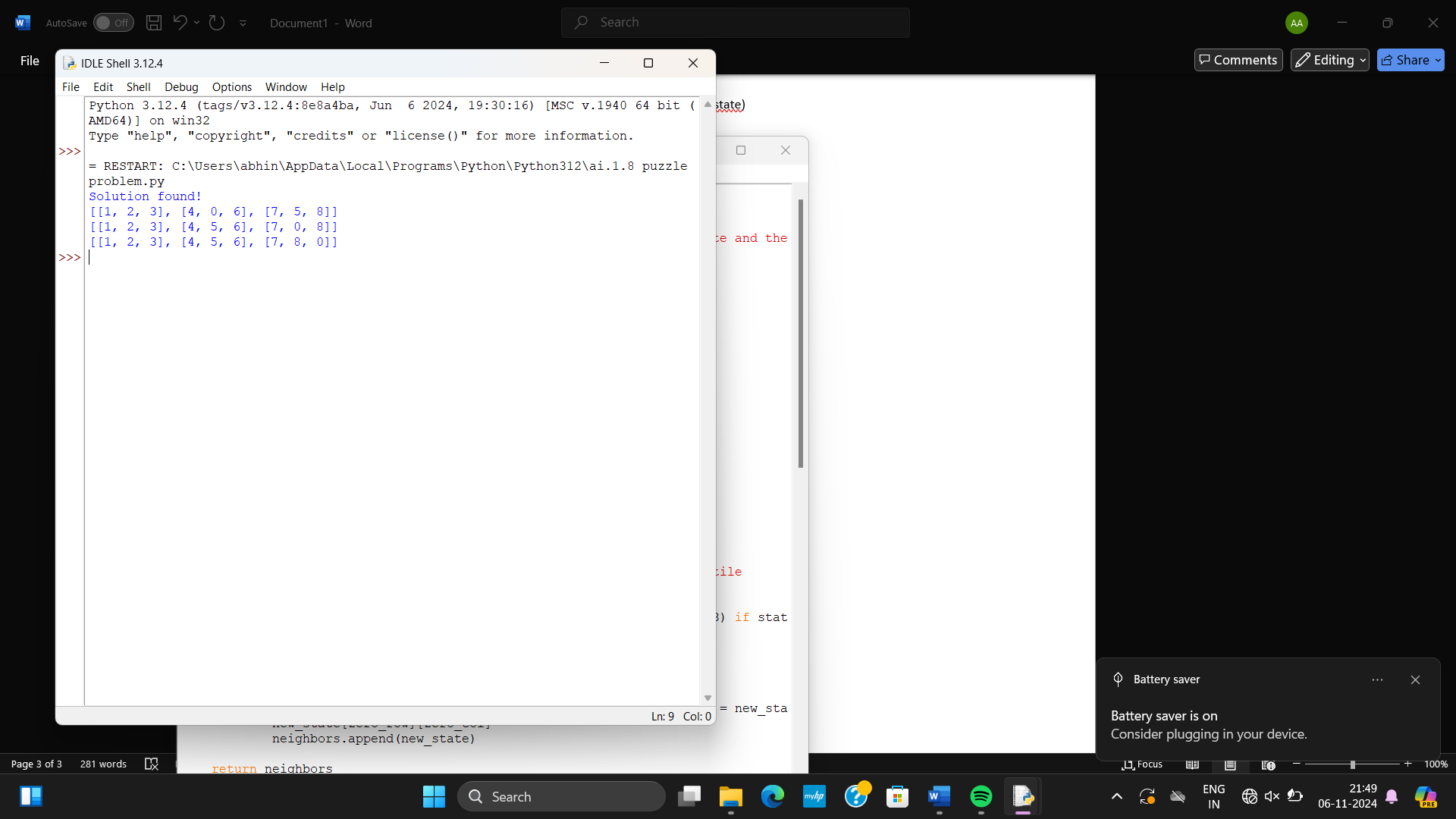
for state in solution\_path:

print(state)

else:

print("No solution exists for the given start state.")

OUTPUT:



2.8-QUEENS PROBLEM

def is\_safe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_queens(board, col):

if col >= len(board):

return True

for i in range(len(board)):

if is\_safe(board, i, col):

board[i][col] = 1

if solve\_queens(board, col + 1):

return True

board[i][col] = 0

return False

def print\_board(board):

for row in board:

print(row)

def solve\_8\_queens():

board = [[0 for \_ in range(8)] for \_ in range(8)]

if not solve\_queens(board, 0):

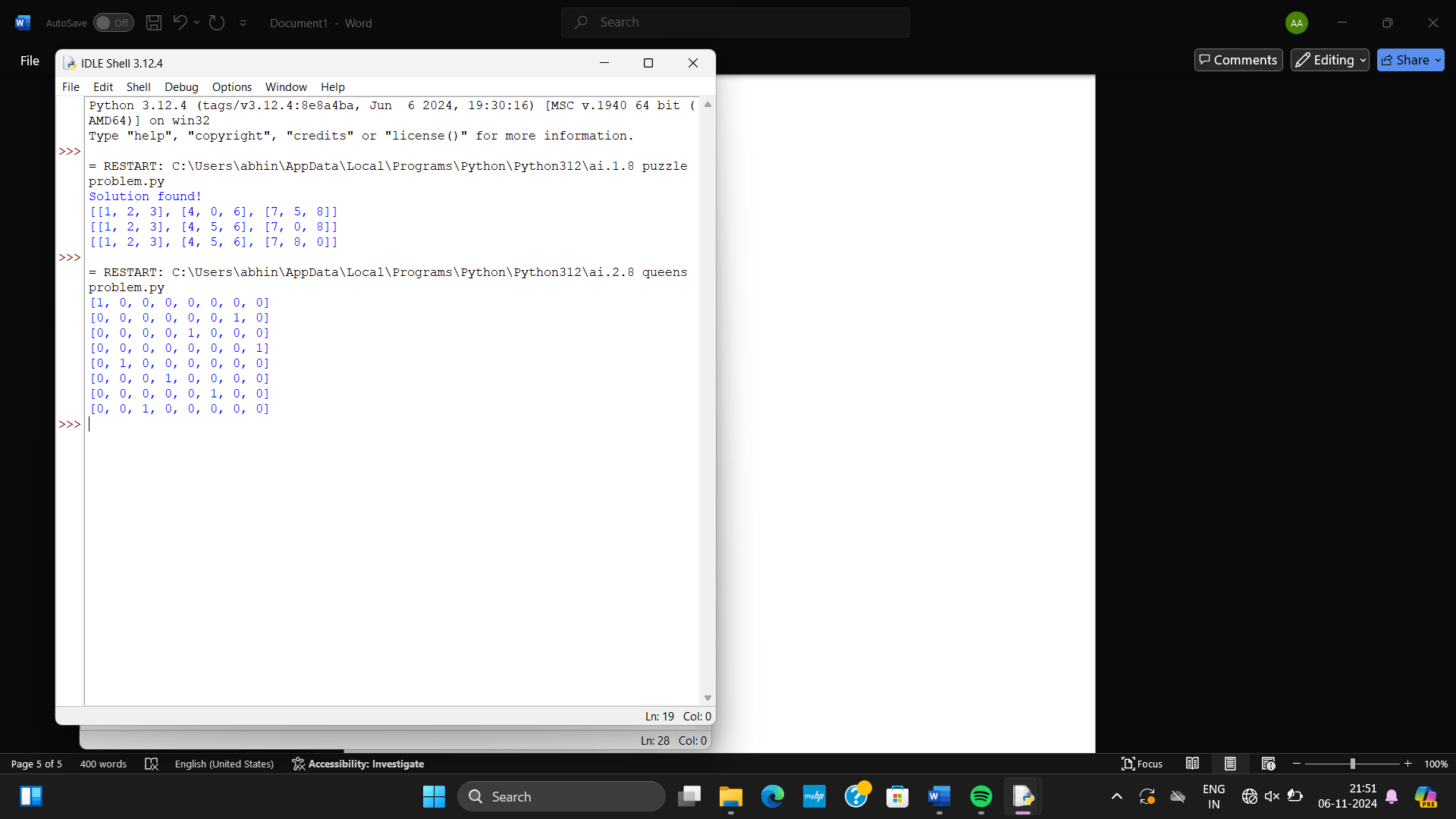
print("No solution found.")

return

print\_board(board)

solve\_8\_queens()

OUTPUT:



3.WATER JUG PROBLEM

from collections import deque

def water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target):

visited = set()

queue = deque()

queue.append((0, 0))

while queue:

current = queue.popleft()

jug1, jug2 = current

print(f"Jug1: {jug1}, Jug2: {jug2}")

if jug1 == target or jug2 == target:

print(f"Found solution: Jug1: {jug1}, Jug2: {jug2}")

return True

if current in visited:

continue

visited.add(current)

queue.append((jug1\_capacity, jug2))

queue.append((jug1, jug2\_capacity))

queue.append((0, jug2))

queue.append((jug1, 0))

pour\_to\_jug2 = min(jug1, jug2\_capacity - jug2)

queue.append((jug1 - pour\_to\_jug2, jug2 + pour\_to\_jug2))

pour\_to\_jug1 = min(jug2, jug1\_capacity - jug1)

queue.append((jug1 + pour\_to\_jug1, jug2 - pour\_to\_jug1))

print("No solution exists.")

return False

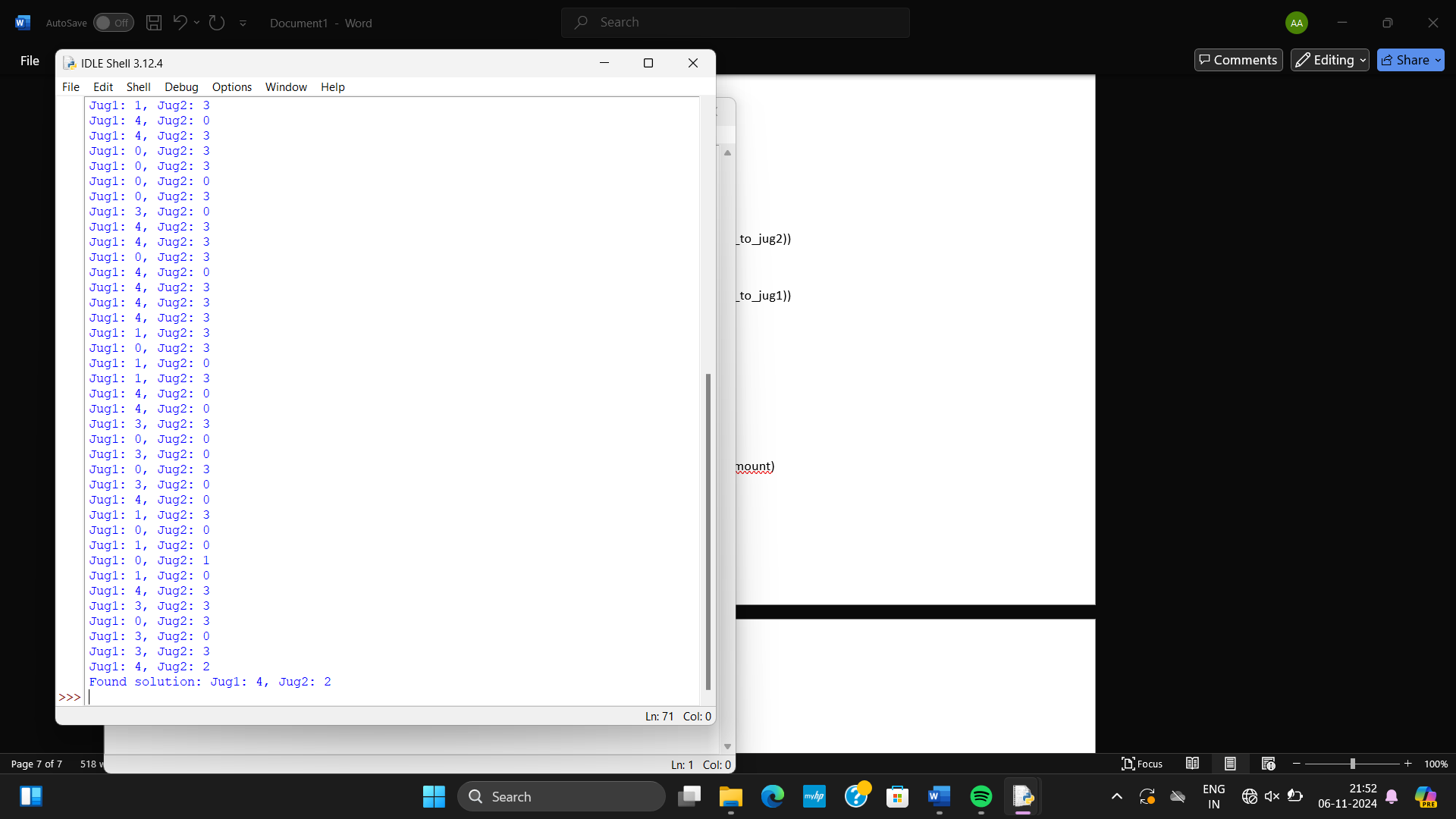
jug1\_capacity = 4 # Capacity of Jug1

jug2\_capacity = 3 # Capacity of Jug2

target\_amount = 2 # The target amount of water

water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target\_amount)

OUTPUT:



4.CRIPT ARITHMETIC PROBLEM

from itertools import permutations

def is\_solution(permutation):

s, e, n, d, m, o, r, y = permutation

send = s \* 1000 + e \* 100 + n \* 10 + d

more = m \* 1000 + o \* 100 + r \* 10 + e

money = m \* 10000 + o \* 1000 + n \* 100 + e \* 10 + y

return send + more == money

def solve\_cryptarithmetic():

letters = 'SENDMORY'

for permutation in permutations(range(10), len(letters))

if permutation[letters.index('M')] == 0:

continue

if is\_solution(permutation):

solution = dict(zip(letters, permutation))

print("Solution found:")

print(f"SEND + MORE = MONEY")

for letter in letters:

print(f"{letter} = {solution[letter]}")

send = solution['S'] \* 1000 + solution['E'] \* 100 + solution['N'] \* 10 + solution['D']

more = solution['M'] \* 1000 + solution['O'] \* 100 + solution['R'] \* 10 + solution['E']

money = solution['M'] \* 10000 + solution['O'] \* 1000 + solution['N'] \* 100 + solution['E'] \* 10 + solution['Y']

print(f"\nNumerical values:")

print(f"SEND = {send}")

print(f"MORE = {more}")

print(f"MONEY = {money}")

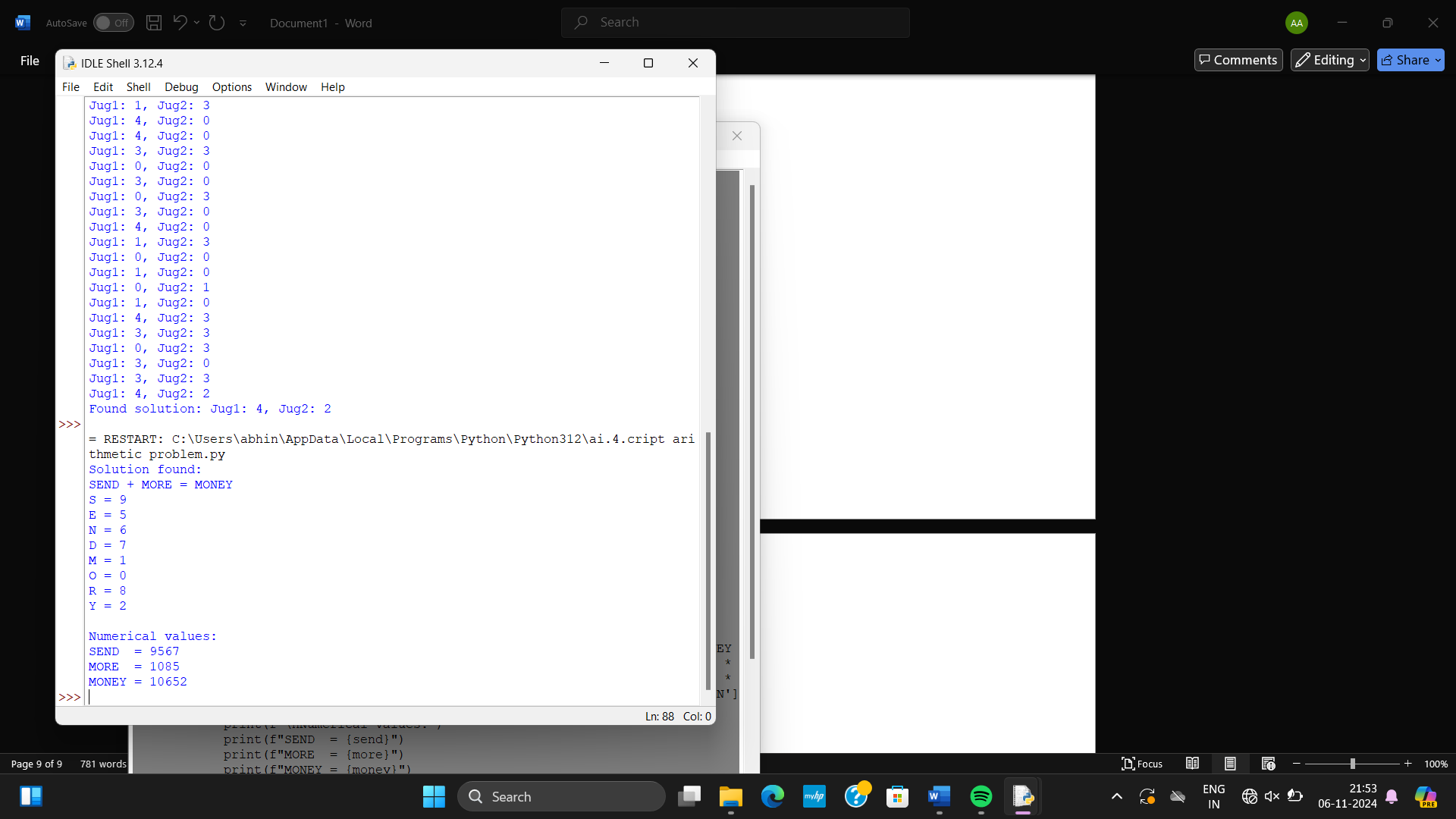
return solution

print("No solution found.")

return None

solve\_cryptarithmetic()

OUTPUT:



5.MISSIONARIES AND CANNIBALS PROBLEM

from collections import deque

def is\_valid\_state(m, c):

if m < 0 or c < 0 or m > 3 or c > 3:

return False

if m > 0 and m < c:

return False

if (3 - m) > 0 and (3 - m) < (3 - c):

return False

return True

def successors(state):

m, c, b = state

moves = []

if b == 1:

possible\_moves = [(-1, 0), (0, -1), (-2, 0), (0, -2), (-1, -1)]

else: possible\_moves = [(1, 0), (0, 1), (2, 0), (0, 2), (1, 1)]

for dm, dc in possible\_moves:

new\_state = (m + dm, c + dc, 1 - b)

if is\_valid\_state(new\_state[0], new\_state[1]):

moves.append(new\_state)

return moves

def bfs():

start\_state = (3, 3, 1)

goal\_state = (0, 0, 0)

visited = set()

queue = deque([(start\_state, [])])

while queue:

state, path = queue.popleft()

if state == goal\_state:

return path + [state]

if state not in visited:

visited.add(state)

for succ in successors(state):

queue.append((succ, path + [state]))

return None

solution = bfs()

if solution:

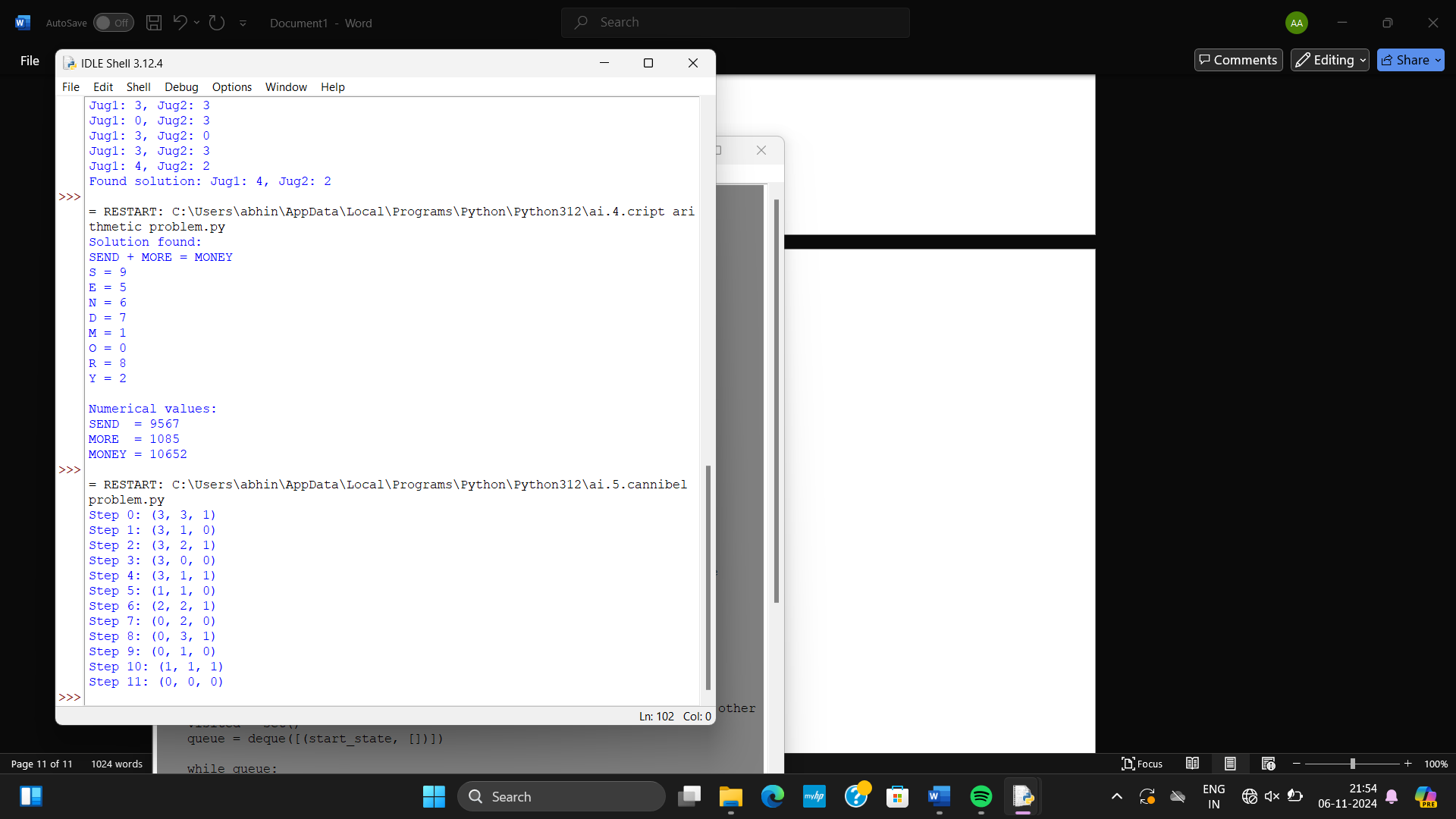
for i, state in enumerate(solution):

print(f"Step {i}: {state}")

else:

print("No solution found.")

OUTPUT:



6.VACCUM CLEANER AGENT PROBLEM

def clean\_room(room):

return room.replace('D', '-')

def vacuum\_cleaner\_world(initial\_room, initial\_position):

current\_room = initial\_room

position = initial\_position

print(f"Initial Room State: {current\_room}")

while 'D' in current\_room:

if current\_room[position] == 'D':

current\_room = clean\_room(current\_room)

print(f"Room Cleaned: {current\_room}")

if position == 0:

position = 1

print("Moving to the right room.")

else:

position = 0

print("Moving to the left room.")

print("All rooms cleaned.")

initial\_room1 = "D-"

initial\_position1 = 0

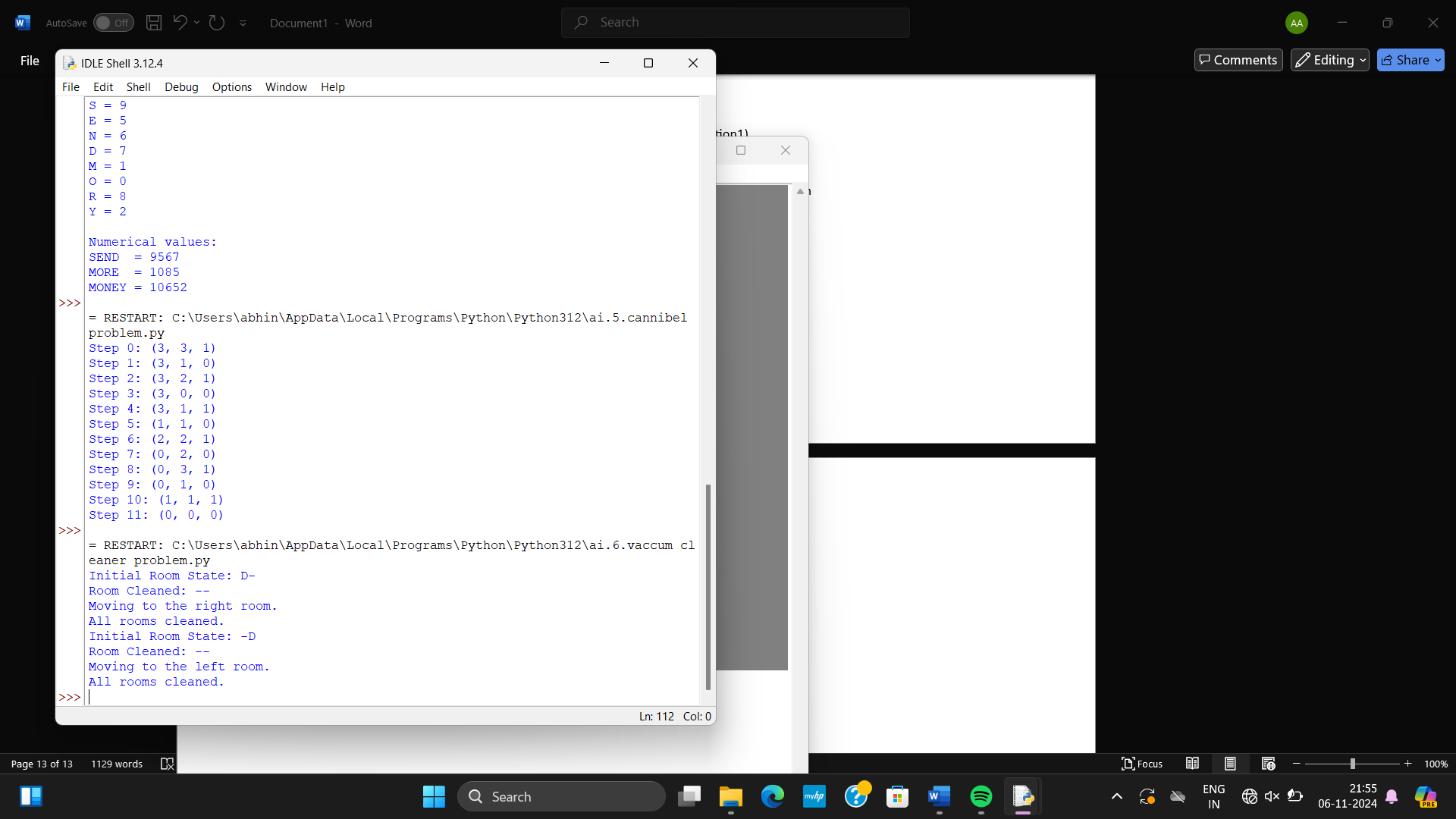
vacuum\_cleaner\_world(initial\_room1, initial\_position1)

initial\_room2 = "-D"

initial\_position2 = 1

vacuum\_cleaner\_world(initial\_room2, initial\_position2)

OUTPUT:



7.BFS PROBLEM

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

visited.add(node)

print(node)

for neighbor in graph[node]:

if neighbor not in visited:

queue.append(neighbor)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D'],

'C': ['A','E','F'],

'D': ['B'],

'E': ['C' ],

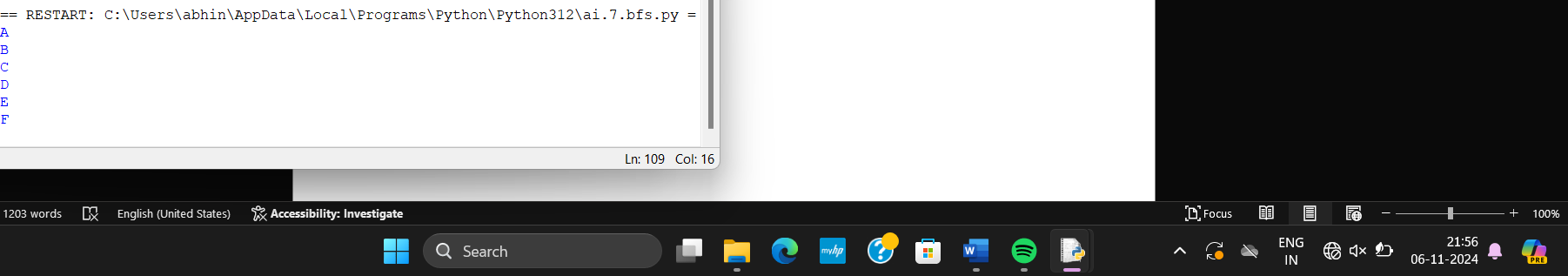
'F': [ 'C']

}

start\_node = 'A'

bfs(graph, start\_node)

OUTPUT:



8.DFS PROBLEM

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start)

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

start\_node = 'A'

dfs(graph, start\_node)

OUTPUT:

