1. Implement Recursive Depth First Search Algorithm. Read the undirected unweighted graph from a .csv file.

CSV Format (example `graph.csv`):

```
A,B
A,C
B,D
C,D
D,E
```

```
import csv


# Function to read graph from CSV

def read_graph_from_csv(file_path):
    graph = {}
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            if not row or ',' not in row[0]:
                continue
            u, v = map(lambda x: x.strip().upper(), row[0].split(','))
            if u not in graph:
                graph[u] = []
            if v not in graph:
                graph[v] = []
            graph[u].append(v)
            graph[v].append(u)
    return graph


graph = read_graph_from_csv(file_path)
print("Graph:", graph)

# Recursive DFS
def recursive_dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node, end=' ')
    for neighbor in graph.get(node, []):
        if neighbor not in visited:
            recursive_dfs(graph, neighbor, visited)

# Provide your CSV file path here
file_path = r"C:\Users\Vraj Shah\OneDrive\Desktop\graph.csv"

# Build the graph
graph = read_graph_from_csv(file_path)

# Optional: print the graph to check its structure
#print("Graph structure:", graph)

# Input starting node and validate
```

```
start_node = input("Enter the starting node for DFS: ").strip().upper()

if start_node not in graph:
    print(f"Error: Node '{start_node}' not found in the graph.")
else:
    print("DFS traversal order:")
    recursive_dfs(graph, start_node)
```

2. Implement Non-Recursive Depth First Search Algorithm. Read the undirected unweighted graph from user.

```
# Non-Recursive DFS for an Undirected Graph

def read_graph():
    graph = {}
    n = int(input("Enter number of edges: "))
    print("Enter each edge as a pair of nodes (e.g., A B):")
    for _ in range(n):
        u, v = input().split()
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append(v)
        graph[v].append(u)  # because the graph is undirected
    return graph

def non_recursive_dfs(graph, start):
    visited = set()
    stack = [start]

    print("DFS traversal order:")
    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            # Add neighbors in reverse sorted order to visit them in lexical order
            for neighbor in sorted(graph[node], reverse=True):
                if neighbor not in visited:
                    stack.append(neighbor)

# Example usage
graph = read_graph()
start_node = input("Enter the starting node for DFS: ")
non_recursive_dfs(graph, start_node)
```

3. Implement Breadth First Search Algorithm. Read the undirected unweighted graph from user.

```
from collections import deque

# Function to read the graph from user
def read_graph_from_user():
    graph = {}
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge in the format 'node1 node2' (space separated):")
    for _ in range(num_edges):
        u, v = input().strip().split()
        u = u.strip().upper()
        v = v.strip().upper()
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append(v)
        graph[v].append(u)  # Undirected graph
    return graph

# BFS Function
def bfs(graph, start_node):
    visited = set()
    queue = deque()

    visited.add(start_node)
    queue.append(start_node)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Main part
graph = read_graph_from_user()
print("\nGraph structure:", graph)

start_node = input("\nEnter the starting node for BFS: ").strip().upper()

if start_node not in graph:
    print(f"Error: Node '{start_node}' not found in the graph.")
else:
```

```
print("\nBFS traversal order:")
bfs(graph, start_node)
```

4. Best First Search Algorithm – Directed| UnWeighted

```
import heapq

def read_graph_from_user():
    graph = {}
    for _ in range(int(input("Enter the number of edges: "))):
        u, v = input("Enter edge (start end): ").strip().upper().split()
        graph.setdefault(u, []).append(v)
    return graph

def read_heuristics():
    heuristics = {}
    for _ in range(int(input("Enter the number of nodes for heuristics: "))):
        node, h = input("Enter node and heuristic (node h): ").strip().upper().split()
        heuristics[node] = int(h)
    return heuristics

def best_first_search(graph, heuristics, start, goal):
    visited = set()
    queue = [(heuristics[start], start)]

    while queue:
        _, node = heapq.heappop(queue)
        if node in visited:
            continue
        print(node, end=' ')
        visited.add(node)
        if node == goal:
            print("\nGoal node reached!")
            return
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(queue, (heuristics[neighbor], neighbor))

    print("\nGoal node not reachable.")

# Main
graph = read_graph_from_user()
heuristics = read_heuristics()

print("\nGraph structure:", graph)
print("Heuristic values:", heuristics)

start = input("\nEnter the starting node: ").strip().upper()
goal = input("Enter the goal node: ").strip().upper()
```

```
if start not in graph:
   print(f"Error: Start node '{start}' not found in graph.")
elif goal not in heuristics:
   print(f"Error: Goal node '{goal}' not found in heuristic values.")
else:
   print("\nBest First Search traversal order:")
   best_first_search(graph, heuristics, start, goal)
```

```
Enter the number of edges:  5
Enter edge (start end):  A B
Enter edge (start end):  A C
Enter edge (start end):  B D
Enter edge (start end):  C E
Enter edge (start end):  D E
Enter the number of nodes for heuristics:  5
Enter node and heuristic (node h):  A 4
Enter node and heuristic (node h):  B 2
Enter node and heuristic (node h):  C 3
Enter node and heuristic (node h):  D 1
Enter node and heuristic (node h):  E 0

Graph structure: {'A': ['B', 'C'], 'B': ['D'], 'C': ['E'], 'D': ['E']}
Heuristic values: {'A': 4, 'B': 2, 'C': 3, 'D': 1, 'E': 0}

Enter the starting node:  A
Enter the goal node:  E

Best First Search traversal order:
A B D E
Goal node reached!
```

5. Best First Search Algorithm – Directed| Weighted

```
# 5) Best FS  Algo - Directed|Weighted

import heapq

# Read graph
def read_graph():
   graph = {}
   for _ in range(int(input("Enter number of edges: "))):
      u, v, w = input("Edge (u v weight): ").upper().split()
      graph.setdefault(u, []).append((v, int(w)))
   return graph

# Read heuristics
```

```
def read_heuristics():
    return {node.upper(): int(h) for node, h in
            (input("Node and Heuristic (node h): ").split() for _ in range(int(input("Enter number of nodes:
"))))}

# Best First Search
def best_first_search(graph, heuristics, start, goal):
    visited, queue = set(), [(heuristics[start], start)]
    while queue:
        _, node = heapq.heappop(queue)
        if node in visited: continue
        print(node, end=' ')
        visited.add(node)
        if node == goal:
            print("\nGoal reached!")
            return
        for neighbor, _ in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(queue, (heuristics[neighbor], neighbor))
    print("\nGoal not reachable.")

# Main
graph = read_graph()
heuristics = read_heuristics()
start, goal = input("Start node: ").strip().upper(), input("Goal node: ").strip().upper()

print("\nBest First Search Traversal:")
best_first_search(graph, heuristics, start, goal)
```

6. Best First Search Algorithm – UnDirected| Weighted

```
import heapq

def read_graph():
    g = {}
    for _ in range(int(input("Edges: "))):
        u, v, w = input().upper().split()
        g.setdefault(u, []).append((v, int(w)))
        g.setdefault(v, []).append((u, int(w)))
    return g

def read_heuristics():
    return {node.upper(): int(h) for node, h in
            (input("Node Heuristic: ").split() for _ in range(int(input("Nodes: "))))}

def best_first_search(g, h, start, goal):
    visited, q = set(), [(h[start], start)]
```

```
    while q:
        _, node = heapq.heappop(q)
        if node in visited: continue
        print(node, end=' ')
        if node == goal:
            print("\nGoal reached!")
            return
        visited.add(node)
        for neighbor, _ in g.get(node, []):
            if neighbor not in visited:
                heapq.heappush(q, (h[neighbor], neighbor))
    print("\nGoal not reachable.")

g = read_graph()
h = read_heuristics()
start, goal = input("Start: ").upper(), input("Goal: ").upper()
print("\nBest First Search Traversal:")
best_first_search(g, h, start, goal)
```

```
Edges:  5
 A B 4
 A C 2
 B D 5
 C D 8
 C E 10
Nodes:  5
Node Heuristic:  A 7
Node Heuristic:  B 6
Node Heuristic:  C 2
Node Heuristic:  D 1
Node Heuristic:  E 0
Start:  A
Goal:  E

Best First Search Traversal:
A C E
Goal reached!
```

7. Best First Search Algorithm – UnDirected| Weighted

```
# 7) Best FS Algo - undirected| Unweighted

import heapq

def read_graph():
    g = {}
    for _ in range(int(input("Edges: "))):
```

```
        u, v = input("Edge (u v): ").upper().split()
        g.setdefault(u, []).append(v)
        g.setdefault(v, []).append(u)
    return g

def read_heuristics():
    return {node.upper(): int(h) for node, h in
            (input("Node Heuristic: ").split() for _ in range(int(input("Nodes: "))))}

def best_first_search(g, h, start, goal):
    visited, q = set(), [(h[start], start)]
    while q:
        _, node = heapq.heappop(q)
        if node in visited: continue
        print(node, end=' ')
        if node == goal:
            print("\nGoal reached!")
            return
        visited.add(node)
        for neighbor in g.get(node, []):
            if neighbor not in visited:
                heapq.heappush(q, (h[neighbor], neighbor))
    print("\nGoal not reachable.")

g = read_graph()
h = read_heuristics()
start, goal = input("Start: ").strip().upper(), input("Goal: ").strip().upper()
print("\nBest First Search Traversal:")
best_first_search(g, h, start, goal)
```

```
Edges:  5
Edge (u v):  A B
Edge (u v):  A C
Edge (u v):  B D
Edge (u v):  C D
Edge (u v):  C E
Nodes:  5
Node Heuristic:  A 7
Node Heuristic:  B 6
Node Heuristic:  C 2
Node Heuristic:  D 1
Node Heuristic:  E 0
Start:  A
Goal:  E

Best First Search Traversal:
A C E
Goal reached!
```

8. A* Algo – Directed|Weighted from CSV file

From	To	Weight
A	B	1
A	C	3
B	D	1
C	D	1
D	E	5
A		7
B		6
C		2
D		1
E		0

```
import csv, heapq

def read_graph_and_heuristics(path):
    g, h = {}, {}
    with open(path) as f:
        next(f)
        for row in csv.reader(f):
            row = [x.strip() for x in row if x.strip() != '']  # Clean empty cells
            if not row:
                continue
            if len(row) == 3:
                u, v, w = row[0].upper(), row[1].upper(), int(row[2])
                g.setdefault(u, []).append((v, w))
            elif len(row) == 2:
                node, heur = row[0].upper(), int(row[1])
                h[node] = heur
    return g, h

def a_star(g, h, start, goal):
    queue, visited = [(h.get(start, 0), 0, start)], set()
    while queue:
        _, g_val, node = heapq.heappop(queue)
        if node in visited: continue
        print(node, end=' ')
        if node == goal: return print("\nGoal reached!")
        visited.add(node)
        for nei, cost in g.get(node, []):
            if nei not in visited:
                heapq.heappush(queue, (g_val + cost + h.get(nei, 0), g_val + cost, nei))
    print("\nGoal not reachable.")

# Main
path = r"C:\Users\Vraj Shah\Downloads\sample_graph_exp 8.csv"  # Your CSV path
graph, heuristics = read_graph_and_heuristics(path)

start = input("Enter start node: ").strip().upper()
goal = input("Enter goal node: ").strip().upper()

print("\nA* Traversal:")
a_star(graph, heuristics, start, goal)
```

9. A* Algo – Directed | Wighted from user

```
import heapq

def read_graph():
    g = {}; n = int(input("Edges? "))
    for _ in range(n):
        u, v, w = input("From, To, Weight: ").upper().split()
        g.setdefault(u, []).append((v, int(w)))
    return g

def read_heuristics():
    return {input("Node: ").upper(): int(input("Heuristic: ")) for _ in range(int(input("Nodes? ")))}

def a_star(g, h, start, goal):
    q, v = [(h.get(start, 0), 0, start)], set()
    while q:
        _, g_val, node = heapq.heappop(q)
        if node in v: continue
        print(node, end=' ')
        if node == goal: return print("\nGoal reached!")
        v.add(node)
        for nei, cost in g.get(node, []):
            if nei not in v:
                heapq.heappush(q, (g_val + cost + h.get(nei, 0), g_val + cost, nei))
    print("\nGoal not reachable.")

# Main
g, h = read_graph(), read_heuristics()
start, goal = input("Start: ").upper(), input("Goal: ").upper()

print("\nA* Traversal:")
a_star(g, h, start, goal)
```

```
Edges?  5
From, To, Weight:  A B 1
From, To, Weight:  A C 3
From, To, Weight:  B D 1
From, To, Weight:  C D 1
From, To, Weight:  D E 5
Nodes?  5
Node:  A 7
Heuristic:  7
Node:  B
Heuristic:  6
Node:  C
Heuristic:  2
Node:  D
Heuristic:  1
```

Node: E
Heuristic: 0
Start: A
Goal: E

A* Traversal:
A C D B E
Goal reached!

10. A* algo – undircted | weighted from CSV

From	To	Weight	Heuristic
A	B	1	7
A	C	3	
B	D	1	6
C	D	1	2
D	E	5	1
E			0

```
import csv, heapq

def read_graph_heuristics(file):
    g, h = {}, {}
    with open(file) as f:
        for r in csv.DictReader(f):
            u = r['From'].strip().upper()
            if r['Heuristic']: h[u] = int(r['Heuristic'])
            if r['To']:
                v, w = r['To'].strip().upper(), int(r['Weight'])
                g.setdefault(u, []).append((v, w))
                g.setdefault(v, []).append((u, w))  # Undirected
    return g, h

def astar(g, h, start, goal):
    q, seen = [(h.get(start, 0), 0, start)], set()
    while q:
        _, g_val, u = heapq.heappop(q)
        if u in seen: continue
        print(u, end=' ')
        if u == goal: return print("\nGoal reached!")
        seen.add(u)
        for v, cost in g.get(u, []):
            if v not in seen:
                heapq.heappush(q, (g_val + cost + h.get(v, 0), g_val + cost, v))
    print("\nGoal not reachable.")

# === Main ===
file = r"C:\Users\Vraj Shah\Downloads\sample_astar_graph_exp 10.csv" # <- Put your file path here!
g, h = read_graph_heuristics(file)
start = input("Enter start node: ").strip().upper()
```

```
goal = input("Enter goal node: ").strip().upper()

print("\nA* Traversal:")
astar(g, h, start, goal)
```

11. A* Algo - Undirected|Weighted from User

```
import heapq

g, h = {}, {}
for _ in range(int(input("Edges: "))):
    u, v, w = input("From: ").upper(), input("To: ").upper(), int(input("Weight: "))
    g.setdefault(u, []).append((v, w))
    g.setdefault(v, []).append((u, w))

for _ in range(int(input("Heuristic nodes: "))):
    n = input("Node: ").upper()
    h[n] = int(input(f"Heuristic {n}: "))

def astar(g, h, start, goal):
    q, seen = [(h.get(start,0), 0, start)], set()
    while q:
        _, gval, u = heapq.heappop(q)
        if u in seen: continue
        print(u, end=' ')
        if u == goal: return print("\nGoal reached!")
        seen.add(u)
        for v, cost in g.get(u, []):
            if v not in seen:
                heapq.heappush(q, (gval+cost+h.get(v,0), gval+cost, v))
    print("\nGoal not reachable.")

start = input("Start: ").strip().upper()
goal = input("Goal: ").strip().upper()
print("\nA* Traversal:")
astar(g, h, start, goal)
```

```
Edges:  5
From:  A
To:  B
Weight:  1
From:  A
To:  C
Weight:  3
From:  B
To:  D
Weight:  1
From:  C
To:  D
Weight:  1
From:  D
To:  E
```

```
Weight:  5
Heuristic nodes:  5
Node:  A
Heuristic A:  7
Node:  B
Heuristic B:  6
Node:  C
Heuristic C:  2
Node:  D
Heuristic D:  1
Node:  E
Heuristic E:  0
Start:  A
Goal:  E

A* Traversal:
A C D B E
Goal reached!
```

12. Implement Fuzzy set operations – union, intersection and complement. Demonstrate these operations with 3 fuzzy sets.

```
# Define 3 fuzzy sets
A = {'x': 0.2, 'y': 0.5, 'z': 0.8}
B = {'x': 0.6, 'y': 0.4, 'z': 0.3}
C = {'x': 0.9, 'y': 0.7, 'z': 0.1}

# Union of two fuzzy sets
def fuzzy_union(set1, set2):
   return {k: max(set1.get(k,0), set2.get(k,0)) for k in set(set1) | set(set2)}

# Intersection of two fuzzy sets
def fuzzy_intersection(set1, set2):
   return {k: min(set1.get(k,0), set2.get(k,0)) for k in set(set1) | set(set2)}

# Complement of a fuzzy set
def fuzzy_complement(set1):
   return {k: round(1 - v, 2) for k, v in set1.items()}

# Display function
def display(title, fz_set):
   print(f"\n{title}:")
   for k, v in fz_set.items():
      print(f"{k}: {v}")

# Perform operations
union_AB = fuzzy_union(A, B)
intersection_BC = fuzzy_intersection(B, C)
complement_C = fuzzy_complement(C)

# Display results
display("Union of A and B", union_AB)
display("Intersection of B and C", intersection_BC)
display("Complement of C", complement_C)
```

13. Implement Fuzzy set operations – union, intersection and complement. Demonstrate De Morgan's Law (Complement of Union) with 2 fuzzy sets.

```
# Define 2 fuzzy sets
A = {'x': 0.3, 'y': 0.6, 'z': 0.8}
B = {'x': 0.7, 'y': 0.4, 'z': 0.5}

# Fuzzy Operations
def fuzzy_union(set1, set2):
   return {k: max(set1.get(k,0), set2.get(k,0)) for k in set(set1) | set(set2)}

def fuzzy_intersection(set1, set2):
   return {k: min(set1.get(k,0), set2.get(k,0)) for k in set(set1) | set(set2)}

def fuzzy_complement(set1):
   return {k: round(1 - v, 2) for k, v in set1.items()}

def display(title, fz_set):
   print(f"\n{title}:")
   for k, v in fz_set.items():
      print(f"{k}: {v}")

# Operations
union_AB = fuzzy_union(A, B)
complement_union = fuzzy_complement(union_AB)

complement_A = fuzzy_complement(A)
complement_B = fuzzy_complement(B)
intersection_complements = fuzzy_intersection(complement_A, complement_B)

# Display
display("Set A", A)
display("Set B", B)
display("Union of A and B", union_AB)
display("Complement of (A ∪ B)", complement_union)
display("Complement of A", complement_A)
display("Complement of B", complement_B)
display("Intersection of complements (¬A ∩ ¬B)", intersection_complements)

# Verify De Morgan's Law
print("\nDe Morgan's Law Verified:", complement_union == intersection_complements)
```

14. Implement Fuzzy set operations – union, intersection and complement. Demonstrate De Morgan's Law (Complement of Intersection) with 2 fuzzy sets.

```
# Fuzzy Set Operations (Short Version)

A = {'x1': 0.2, 'x2': 0.7, 'x3': 1.0}
B = {'x1': 0.5, 'x2': 0.4, 'x3': 0.9}

print("Set A:", A)
print("Set B:", B)

# Operations
union = {x: max(A.get(x, 0), B.get(x, 0)) for x in set(A) | set(B)}
intersection = {x: min(A.get(x, 0), B.get(x, 0)) for x in set(A) | set(B)}
complement_A = {x: 1 - A[x] for x in A}
complement_B = {x: 1 - B[x] for x in B}
```

```
print("\nUnion (A ∪ B):", union)
print("Intersection (A ∩ B):", intersection)
print("Complement of A (A'):", complement_A)
print("Complement of B (B'):", complement_B)

# De Morgan's Law
comp_intersection = {x: 1 - intersection[x] for x in intersection}
union_complements = {x: max(complement_A.get(x, 0), complement_B.get(x, 0)) for x in set(complement_A) |
set(complement_B)}

print("\nComplement of (A ∩ B):", comp_intersection)
print("Union of (A' ∪ B'):", union_complements)

print("\n De Morgan's Law Verified!" if comp_intersection == union_complements else "\n❌ De Morgan's Law Failed!")
```

15. Implement any two-player game (Modified Tic-Tac-Toe, Nim Game, Connect Four Game or Gomoku Game) using min-max algorithm such that in every play either computer wins or it is a draw.

```
# Modified Tic-Tac-Toe (Minimax AI)

import math

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board):
    for line in board + list(zip(*board)) + [[board[i][i] for i in range(3)], [board[i][2-i] for i in range(3)]]:
        if all(cell == 'X' for cell in line):
            return 'X'
        if all(cell == 'O' for cell in line):
            return 'O'
    return None if any(cell == ' ' for row in board for cell in row) else 'Draw'

def minimax(board, is_maximizing):
    winner = check_winner(board)
    if winner == 'O': return 1
    if winner == 'X': return -1
    if winner == 'Draw': return 0

    best_score = -math.inf if is_maximizing else math.inf

    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O' if is_maximizing else 'X'
                score = minimax(board, not is_maximizing)
                board[i][j] = ' '
                best_score = max(score, best_score) if is_maximizing else min(score, best_score)

    return best_score

def best_move(board):
    move = None
    best_score = -math.inf

    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
```

```
            board[i][j] = 'O'
            score = minimax(board, False)
            board[i][j] = ' '
            if score > best_score:
                best_score = score
                move = (i, j)
    return move

# Main Game Loop
board = [[' ']*3 for _ in range(3)]

print("Welcome to Modified Tic-Tac-Toe!")
print_board(board)

while True:
    # Player move
    try:
        row, col = map(int, input("\nEnter your move (row and column 0-2): ").split())
        if board[row][col] != ' ':
            print("Invalid move! Cell occupied.")
            continue
        board[row][col] = 'X'
    except (ValueError, IndexError):
        print("Invalid input! Enter row and column between 0-2.")
        continue

    print_board(board)
    if (result := check_winner(board)):
        print("\nResult:", result)
        break

    # Computer move
    i, j = best_move(board)
    board[i][j] = 'O'
    print("\nComputer's move:")
    print_board(board)

    if (result := check_winner(board)):
        print("\nResult:", result)
        break
```

16. Implement any two-player game (Modified Tic-Tac-Toe, Nim Game, Connect Four Game or Gomoku Game) using min-max algorithm such that in every play either computer loses or it is a draw.

```
import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board):
    lines = board + list(zip(*board)) + [[board[i][i] for i in range(3)], [board[i][2-i] for i in range(3)]]
    for line in lines:
        if all(cell == 'X' for cell in line):
            return 'X'
        if all(cell == 'O' for cell in line):
            return 'O'
    return None if any(cell == ' ' for row in board for cell in row) else 'Draw'
```

```
# BAD Minimax: Computer makes random valid moves
def bad_move(board):
    empty = [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']
    return random.choice(empty) if empty else None

# Main Game
board = [[' ']*3 for _ in range(3)]

print("Welcome to Modified Tic-Tac-Toe (Computer loses or draws)!")
print_board(board)

while True:
    # Player move
    try:
        row, col = map(int, input("\nEnter your move (row and column 0-2): ").split())
        if board[row][col] != ' ':
            print("Invalid move! Cell occupied.")
            continue
        board[row][col] = 'X'
    except (ValueError, IndexError):
        print("Invalid input! Enter row and column between 0-2.")
        continue

    print_board(board)
    if (result := check_winner(board)):
        print("\nResult:", result)
        break

    # Computer bad move
    move = bad_move(board)
    if move:
        i, j = move
        board[i][j] = 'O'
        print("\nComputer's move:")
        print_board(board)

    if (result := check_winner(board)):
        print("\nResult:", result)
        break
```

17. Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one binary output. Display the final weight matrices, bias values and the number of steps. Note that random values are assigned to weight matrices and bias in each step.

```
import numpy as np

# Activation function: Sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of Sigmoid (for backpropagation)
def sigmoid_derivative(x):
    return x * (1 - x)

# Function to create a simple MLP
def simple_mlp(N, max_steps=10000, learning_rate=0.1):
    # Generate random binary inputs (X) and expected outputs (y)
```

```
    X = np.random.randint(0, 2, (N, N))  # N samples, N features
    y = np.random.randint(0, 2, (N, 1))  # N samples, 1 output

    # Randomly initialize weights and biases
    np.random.seed()
    w1 = np.random.uniform(-1, 1, (N, N))  # Input to hidden1
    b1 = np.random.uniform(-1, 1, (1, N))

    w2 = np.random.uniform(-1, 1, (N, N))  # hidden1 to hidden2
    b2 = np.random.uniform(-1, 1, (1, N))

    w3 = np.random.uniform(-1, 1, (N, 1))  # hidden2 to output
    b3 = np.random.uniform(-1, 1, (1, 1))

    steps = 0
    for step in range(max_steps):
        steps += 1

        # Forward Pass
        z1 = np.dot(X, w1) + b1
        a1 = sigmoid(z1)

        z2 = np.dot(a1, w2) + b2
        a2 = sigmoid(z2)

        z3 = np.dot(a2, w3) + b3
        output = sigmoid(z3)

        # Compute Error
        error = y - output
        if np.mean(np.abs(error)) < 0.01:  # Stop if error is small enough
            break

        # Backward Pass
        d_output = error * sigmoid_derivative(output)
        d_hidden2 = d_output.dot(w3.T) * sigmoid_derivative(a2)
        d_hidden1 = d_hidden2.dot(w2.T) * sigmoid_derivative(a1)

        # Update Weights and Biases
        w3 += a2.T.dot(d_output) * learning_rate
        b3 += np.sum(d_output, axis=0, keepdims=True) * learning_rate

        w2 += a1.T.dot(d_hidden2) * learning_rate
        b2 += np.sum(d_hidden2, axis=0, keepdims=True) * learning_rate

        w1 += X.T.dot(d_hidden1) * learning_rate
        b1 += np.sum(d_hidden1, axis=0, keepdims=True) * learning_rate

    # Display final results
    print("Final Weights and Biases after training:\n")
    print(f"w1 (Input -> Hidden Layer 1):\n{w1}\n")
    print(f"b1 (Hidden Layer 1 bias):\n{b1}\n")

    print(f"w2 (Hidden Layer 1 -> Hidden Layer 2):\n{w2}\n")
    print(f"b2 (Hidden Layer 2 bias):\n{b2}\n")

    print(f"w3 (Hidden Layer 2 -> Output):\n{w3}\n")
    print(f"b3 (Output bias):\n{b3}\n")

    print(f"Total number of steps taken: {steps}")

# Example: Let's run with N=4 inputs
simple_mlp(N=4)
```

18. Implement a simple Multi-Layer Perceptron with 4 binary inputs, one hidden layer and two binary outputs. Display the final weight matrices, bias values and the number of steps. Note that random values are assigned to weight matrices and bias in each step.

```
import numpy as np

# Step activation function
def step_function(x):
    return np.where(x >= 0, 1, 0)

# Define MLP parameters
input_size = 4
hidden_size = 5   # You can choose any number for hidden neurons
output_size = 2

# Random weight initialization
W1 = np.random.randn(input_size, hidden_size)
b1 = np.random.randn(hidden_size)
W2 = np.random.randn(hidden_size, output_size)
b2 = np.random.randn(output_size)

# Dummy input data (4 binary inputs)
X = np.random.randint(0, 2, (1, input_size))

steps = 0
output = None

while True:
    steps += 1

    # Forward pass
    hidden_input = np.dot(X, W1) + b1
    hidden_output = step_function(hidden_input)

    final_input = np.dot(hidden_output, W2) + b2
    final_output = step_function(final_input)

    # If final output is binary (0 or 1) for both outputs, break
    if np.all((final_output == 0) | (final_output == 1)):
        output = final_output
        break
    else:
        # Randomize weights and biases again
        W1 = np.random.randn(input_size, hidden_size)
        b1 = np.random.randn(hidden_size)
        W2 = np.random.randn(hidden_size, output_size)
        b2 = np.random.randn(output_size)

# Display results
print("Input X:\n", X)
print("\nFinal hidden layer weights W1:\n", W1)
print("\nFinal hidden layer bias b1:\n", b1)
print("\nFinal output layer weights W2:\n", W2)
print("\nFinal output layer bias b2:\n", b2)
print("\nFinal output:\n", output)
print("\nTotal steps taken:", steps)
```

19. Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one output. Use backpropagation and Sigmoid function as activation function.

```
import numpy as np

# Sigmoid activation and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define MLP structure
N = 4  # Number of inputs (you can change N easily)
hidden1_size = 5
hidden2_size = 3
output_size = 1

# Random weight initialization
W1 = np.random.randn(N, hidden1_size)
b1 = np.random.randn(hidden1_size)

W2 = np.random.randn(hidden1_size, hidden2_size)
b2 = np.random.randn(hidden2_size)

W3 = np.random.randn(hidden2_size, output_size)
b3 = np.random.randn(output_size)

# Dummy training data (binary inputs and binary output)
X = np.random.randint(0, 2, (10, N))  # 10 samples
y = np.random.randint(0, 2, (10, 1))

# Training parameters
epochs = 5000
learning_rate = 0.1

# Training using Backpropagation
for epoch in range(epochs):
    # Forward pass
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)

    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    z3 = np.dot(a2, W3) + b3
    output = sigmoid(z3)

    # Compute error
    error = y - output

    # Backward pass
    d_output = error * sigmoid_derivative(output)

    d_hidden2 = np.dot(d_output, W3.T) * sigmoid_derivative(a2)

    d_hidden1 = np.dot(d_hidden2, W2.T) * sigmoid_derivative(a1)

    # Update weights and biases
    W3 += learning_rate * np.dot(a2.T, d_output)
```

```
    b3 += learning_rate * np.sum(d_output, axis=0)

    W2 += learning_rate * np.dot(a1.T, d_hidden2)
    b2 += learning_rate * np.sum(d_hidden2, axis=0)

    W1 += learning_rate * np.dot(X.T, d_hidden1)
    b1 += learning_rate * np.sum(d_hidden1, axis=0)

# Final Results
print("\nTraining complete!")
print("\nFinal Input X:\n", X)
print("\nExpected Output y:\n", y)
print("\nFinal Output after training:\n", np.round(output))
print("\nFinal Weights and Biases:")
print("\nW1:\n", W1)
print("\nb1:\n", b1)
print("\nW2:\n", W2)
print("\nb2:\n", b2)
print("\nW3:\n", W3)
print("\nb3:\n", b3)
```

20. Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one output. Use backpropagation and ReLU function as activation function.

```
import numpy as np

# ReLU and its derivative
relu = lambda x: np.maximum(0, x)
relu_deriv = lambda x: (x > 0).astype(float)

# Setup
N, h1, h2, out = 4, 5, 3, 1
X = np.random.randint(0, 2, (10, N))
y = np.random.randint(0, 2, (10, 1))

# Random weights and biases
W1, b1 = np.random.randn(N, h1), np.random.randn(h1)
W2, b2 = np.random.randn(h1, h2), np.random.randn(h2)
W3, b3 = np.random.randn(h2, out), np.random.randn(out)

# Training
for _ in range(5000):
    a1 = relu(X @ W1 + b1)
    a2 = relu(a1 @ W2 + b2)
    out_pred = relu(a2 @ W3 + b3)

    error = y - out_pred
    d_out = error * relu_deriv(out_pred)
    d_h2 = (d_out @ W3.T) * relu_deriv(a2)
    d_h1 = (d_h2 @ W2.T) * relu_deriv(a1)

    W3 += 0.01 * a2.T @ d_out; b3 += 0.01 * d_out.sum(0)
    W2 += 0.01 * a1.T @ d_h2; b2 += 0.01 * d_h2.sum(0)
    W1 += 0.01 * X.T @ d_h1; b1 += 0.01 * d_h1.sum(0)

# Output
print("\nInput X:\n", X)
```

```
print("\nTarget y:\n", y)
print("\nPredicted Output:\n", np.round(out_pred))
print("\nWeights and Biases:")
print("\nW1:\n", W1, "\nb1:\n", b1)
print("\nW2:\n", W2, "\nb2:\n", b2)
print("\nW3:\n", W3, "\nb3:\n", b3)
```

21. Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one output. Use backpropagation and Tanh function as activation function.

```
import numpy as np

# Tanh and its derivative
tanh = lambda x: np.tanh(x)
tanh_deriv = lambda x: 1 - np.tanh(x)**2

# Setup
N, h1, h2, out = 4, 5, 3, 1
X = np.random.randint(0, 2, (10, N))
y = np.random.randint(0, 2, (10, 1))

# Random weights and biases
W1, b1 = np.random.randn(N, h1), np.random.randn(h1)
W2, b2 = np.random.randn(h1, h2), np.random.randn(h2)
W3, b3 = np.random.randn(h2, out), np.random.randn(out)

# Training
for _ in range(5000):
    a1 = tanh(X @ W1 + b1)
    a2 = tanh(a1 @ W2 + b2)
    out_pred = tanh(a2 @ W3 + b3)

    error = y - out_pred
    d_out = error * tanh_deriv(out_pred)
    d_h2 = (d_out @ W3.T) * tanh_deriv(a2)
    d_h1 = (d_h2 @ W2.T) * tanh_deriv(a1)

    W3 += 0.01 * a2.T @ d_out; b3 += 0.01 * d_out.sum(0)
    W2 += 0.01 * a1.T @ d_h2; b2 += 0.01 * d_h2.sum(0)
    W1 += 0.01 * X.T @ d_h1; b1 += 0.01 * d_h1.sum(0)

# Output
print("\nInput X:\n", X)
print("\nTarget y:\n", y)
print("\nPredicted Output:\n", np.round(out_pred))
print("\nWeights and Biases:")
print("\nW1:\n", W1, "\nb1:\n", b1)
print("\nW2:\n", W2, "\nb2:\n", b2)
print("\nW3:\n", W3, "\nb3:\n", b3)
```

22. Write a program to read a text file with at least 30 sentences and 200 words and perform the following tasks in the given sequence.
a. Text cleaning by removing punctuation/special characters, numbers and extra white spaces. Use regular expression for the same.
b. Convert text to lowercase

c. Tokenization
d. Remove stop words
e. Correct misspelled words.

```
!pip install re
!pip install nltk
!pip install textblob
import nltk
nltk.download('punkt')
nltk.download('stopwords')

import re
import nltk
from nltk.corpus import stopwords
from textblob import TextBlob

nltk.download('stopwords')

file_path = r"C:\Users\Vraj Shah\OneDrive\Desktop\The sun rises in the east. Birds ch.txt"
with open(file_path, 'r', encoding='utf-8') as f:
    text = f.read()

# Cleaning
text = re.sub(r'[^a-zA-Z\s]', '', text)
text = re.sub(r'\s+', ' ', text)
text = text.lower()

# Tokenization (simple way)
tokens = text.split()

# Remove Stopwords
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]

# Correct Spelling
corrected_tokens = [str(TextBlob(word).correct()) for word in filtered_tokens]

# Final output
print("\nCleaned and Corrected Text:\n")
print(' '.join(corrected_tokens))
```

23. Write a program to read a text file with at least 30 sentences and 200 words and perform the following tasks in the given sequence.
a. Text cleaning by removing punctuation/special characters, numbers and extra white spaces. Use regular expression for the same.
b. Convert text to lowercase
c. Stemming and Lemmatization
d. Create a list of 3 consecutive words after lemmatization.

```
import re
import nltk
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('wordnet')
nltk.download('omw-1.4')
```

```
# Step 1: Read File
file_path = r"C:\Users\Vraj Shah\OneDrive\Desktop\The sun rises in the east. Birds ch.txt" #
<<< Change accordingly
with open(file_path, 'r', encoding='utf-8') as f:
    text = f.read()

# Step 2: Text Cleaning
text = re.sub(r'[^a-zA-Z\s]', '', text)
text = re.sub(r'\s+', ' ', text)

# Step 3: Lowercase
text = text.lower()

# Step 4: Tokenization (Simple Split)
tokens = text.split()

# Step 5: Stemming
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in tokens]

# Step 6: Lemmatization
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(word) for word in stemmed_tokens]

# Step 7: 3-Consecutive Words (triplets)
triplets = [' '.join(lemmatized_tokens[i:i+3]) for i in range(len(lemmatized_tokens)-2)]

# Final Outputs
print("\nLemmatized Tokens:\n", lemmatized_tokens)
print("\nList of 3-Consecutive Words:\n", triplets)
```

24. Write a program to read a 3 text files on any technical concept with at least 20 sentences and 150 words. Implement one-hot encoding.

```
import re

# Function to read and clean text
def read_and_clean(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read()
    text = text.lower()
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    text = re.sub(r'\s+', ' ', text)
    return text

# Function to create vocabulary
def build_vocab(texts):
    vocab = set()
    for text in texts:
        vocab.update(text.split())
    vocab = sorted(list(vocab))  # Sort for consistent order
```

```
    return vocab

# Function to one-hot encode a text
def one_hot_encode(text, vocab):
    words = text.split()
    word_to_index = {word: idx for idx, word in enumerate(vocab)}
    one_hot_vectors = []
    for word in words:
        vector = [0] * len(vocab)
        if word in word_to_index:
            vector[word_to_index[word]] = 1
        one_hot_vectors.append(vector)
    return one_hot_vectors

# Step 1: Read and Clean the 3 files
file1 = r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 24 - Artificial Intelligence, often abbr.txt" # Change this
file2 = r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 24 - Blockchain is a distributed ledger.txt" # Change this
file3 = r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 24 - Cloud computing is the delivery of.txt"  # Change this

text1 = read_and_clean(file1)
text2 = read_and_clean(file2)
text3 = read_and_clean(file3)

texts = [text1, text2, text3]

# Step 2: Build vocabulary from all files
vocab = build_vocab(texts)
print("Vocabulary Size:", len(vocab))
print("Vocabulary List:\n", vocab)

# Step 3: One-Hot Encode each text
encoded_texts = [one_hot_encode(text, vocab) for text in texts]

# Step 4: Display results
for i, encoded in enumerate(encoded_texts, start=1):
    print(f"\nOne-Hot Encoding for File {i}:")
    for vector in encoded[:10]:  # Show only first 10 vectors for readability
        print(vector)
```

25. Write a program to read a 3 text files on a movie review with at least 20 sentences and 150 words. Implement bag of words.

```
!pip install scikit-learn

from sklearn.feature_extraction.text import CountVectorizer
import os

# Step 1: Read the contents of the 3 movie review files
file_names = [
    r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 25 - The Dark Knight iReview.txt",
    r"C:\Users\Vraj Shah\OneDrive\Desktop\exp 25 - Interstellar review.txt",
    r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 25 - Inception Review.txt"
]

documents = []

for file_name in file_names:
    with open(file_name, 'r', encoding='utf-8') as file:
        documents.append(file.read())

# Step 2: Initialize CountVectorizer (Bag of Words)
```

```
vectorizer = CountVectorizer()

# Step 3: Fit and Transform the documents
X = vectorizer.fit_transform(documents)

# Step 4: Display the vocabulary
print("\nVocabulary (Words extracted):")
print(vectorizer.get_feature_names_out())

# Step 5: Display the Bag of Words Matrix
print("\nBag of Words Matrix (Word Counts):")
print(X.toarray())
```

26. Write a program to read a 3 text files a tourist place with at least 20 sentences and 150 words. Implement TF-IDF.

```
import math
import os

# Read files
files = [r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 26 - The Eiffel Tower.txt", r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp
26 - The Statue of Liberty.txt",r"C:\Users\Vraj Shah\OneDrive\Desktop\Exp 26 - The Taj Mahal.txt"]
documents = []
for file in files:
    with open(file, 'r', encoding='utf-8') as f:
        documents.append(f.read().lower())

# Tokenize
def tokenize(text):
    return text.replace('.', '').replace(',', '').split()

tokenized_docs = [tokenize(doc) for doc in documents]

# Build Vocabulary
vocab = sorted(set(word for doc in tokenized_docs for word in doc))

# Term Frequency (TF)
def compute_tf(doc):
    tf = {}
    for word in vocab:
        tf[word] = doc.count(word) / len(doc)
    return tf

tfs = [compute_tf(doc) for doc in tokenized_docs]

# Inverse Document Frequency (IDF)
def compute_idf():
    idf = {}
    N = len(tokenized_docs)
    for word in vocab:
        df = sum(word in doc for doc in tokenized_docs)
        idf[word] = math.log((N + 1) / (df + 1)) + 1  # smoothing
    return idf

idf = compute_idf()

# TF-IDF
tfidfs = []
for tf in tfs:
    tfidf = {word: tf[word] * idf[word] for word in vocab}
```

```
    tfidfs.append(tfidf)

# Display results
for i, tfidf in enumerate(tfidfs):
    print(f"\nTF-IDF for {files[i]}:")
    sorted_words = sorted(tfidf.items(), key=lambda x: x[1], reverse=True)
    for word, score in sorted_words[:10]:  # Top 10 words
        print(f"{word}: {score:.4f}")
```