```
# ==============================================
# EXPERIMENT 1: RECURSIVE DFS (KEYWORD: REC_DFS)
# ==============================================
# Theory:
# Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible
# along each branch before backtracking. In the recursive implementation:
# 1. Start at a node and mark it as visited
# 2. Recursively visit all unvisited neighbors
# 3. Backtrack when no unvisited neighbors remain
#
# Algorithm:
# 1. Create a recursive function that takes the graph, current node, and visited set
# 2. Mark current node as visited and process it
# 3. For each neighbor of current node:
#    a. If neighbor is not visited, recursively call the function for that neighbor
# 4. Return when all neighbors are processed

import pandas as pd

def dfs_recursive(graph, node, visited):
    visited.add(node)
    print(node, end=" ")  # Process the node
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

# Read graph from CSV
df = pd.read_csv('a.csv', header=None)
graph = {}
for row in df.values:
    u, v = row
    graph.setdefault(u, []).append(v)
    graph.setdefault(v, []).append(u)

# Perform DFS
start_node = list(graph.keys())[0]  # Start from the first node
visited = set()
dfs_recursive(graph, start_node, visited)


# ==============================================
# EXPERIMENT 2: NON-RECURSIVE DFS (KEYWORD: STACK_DFS)
# ==============================================
# Theory:
# Non-recursive DFS uses a stack data structure instead of the call stack.
# This approach is more memory-efficient for large graphs as it avoids
# stack overflow errors that might occur with the recursive approach.
#
# Algorithm:
# 1. Create a stack and push the starting node
# 2. Create a visited set to track visited nodes
# 3. While stack is not empty:
#    a. Pop a node from the stack
#    b. If node is not visited, mark it as visited and process it
#    c. Push all unvisited neighbors to the stack
# 4. Return when stack is empty

def dfs_non_recursive(graph, start):
    stack = [start]
    visited = set()
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            # Add neighbors in reverse order to maintain left-to-right traversal
            stack.extend(reversed(graph[node]))

# Input graph from user
graph = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v = input("Enter edge (u v): ").split()
    graph.setdefault(u, []).append(v)
    graph.setdefault(v, []).append(u)

# Perform DFS
start_node = input("Enter start node: ")
dfs_non_recursive(graph, start_node)
```

```
# ===============================================
# EXPERIMENT 3: BFS (KEYWORD: BFS)
# ===============================================
# Theory:
# Breadth-First Search (BFS) traverses a graph level by level. It visits all
# neighbors at the current level before moving to nodes at the next level.
# BFS is useful for finding the shortest path in unweighted graphs.
#
# Algorithm:
# 1. Create a queue and enqueue the starting node
# 2. Create a visited set to track visited nodes
# 3. While queue is not empty:
#    a. Dequeue a node from the queue
#    b. If node is not visited, mark it as visited and process it
#    c. Enqueue all unvisited neighbors
# 4. Return when queue is empty

from collections import deque

def bfs(graph, start):
    queue = deque([start])
    visited = set()
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Input graph from user
graph = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v = input("Enter edge (u v): ").split()
    graph.setdefault(u, []).append(v)
    graph.setdefault(v, []).append(u)

# Perform BFS
start_node = input("Enter start node: ")
bfs(graph, start_node)


# ===============================================
# EXPERIMENT 4: BEST FIRST SEARCH - DIRECTED UNWEIGHTED (KEYWORD: BFS_DIR_UNW)
# ===============================================
# Theory:
# Best First Search is a search algorithm that selects the path which appears
# best based on a heuristic or evaluation function. It uses a priority queue
# where nodes with better heuristic values are explored first.
#
# Algorithm:
# 1. Create a priority queue and add the starting node with its heuristic value
# 2. Create a visited set to track visited nodes
# 3. While priority queue is not empty:
#    a. Remove node with the best heuristic value
#    b. If node is not visited, mark it as visited and process it
#    c. Add all unvisited neighbors to the priority queue with their heuristic values
# 4. Return when priority queue is empty

import heapq

def best_first_search_directed_unweighted(graph, start, heuristic):
    priority_queue = [(heuristic[start], start)]
    visited = set()
    while priority_queue:
        _, node = heapq.heappop(priority_queue)
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            for neighbor in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (heuristic[neighbor], neighbor))

# Input directed unweighted graph and heuristic values from user
graph = {}
heuristic = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v = input("Enter edge (u v): ").split()
    graph.setdefault(u, []).append(v)
    # Note: No reverse edge since it's directed

nodes = set(graph.keys())
```

```
for node in nodes:
    heuristic[node] = float(input(f"Enter heuristic value for {node}: "))

# Perform Best First Search
start_node = input("Enter start node: ")
best_first_search_directed_unweighted(graph, start_node, heuristic)



# ===============================================
# EXPERIMENT 5: BEST FIRST SEARCH – UNDIRECTED WEIGHTED (KEYWORD: BFS_UNDIR_W)
# ===============================================
# Theory:
# This version of Best First Search works with undirected weighted graphs.
# The weights don't affect the search order directly (that would be Dijkstra's),
# but we store them for potential use in applications.
#
# Algorithm:
# Same as Experiment 4, but we add edges in both directions for undirected graph
# and store weights (though they don't affect the search order).

import heapq

def best_first_search_undirected_weighted(graph, start, heuristic):
    priority_queue = [(heuristic[start], start)]
    visited = set()
    while priority_queue:
        _, node = heapq.heappop(priority_queue)
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            for neighbor, weight in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (heuristic[neighbor], neighbor))

# Input undirected weighted graph and heuristic values from user
graph = {}
heuristic = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v, weight = input("Enter edge (u v weight): ").split()
    weight = float(weight)
    graph.setdefault(u, []).append((v, weight))
    graph.setdefault(v, []).append((u, weight))  # Add reverse edge for undirected

nodes = set()
for node in graph:
    nodes.add(node)
    for neighbor, _ in graph[node]:
        nodes.add(neighbor)

for node in nodes:
    heuristic[node] = float(input(f"Enter heuristic value for {node}: "))

# Perform Best First Search
start_node = input("Enter start node: ")
best_first_search_undirected_weighted(graph, start_node, heuristic)


# ===============================================
# EXPERIMENT 6: BEST FIRST SEARCH – UNDIRECTED UNWEIGHTED (KEYWORD: BFS_UNDIR_UNW)
# ===============================================
# Theory:
# Best First Search for undirected unweighted graphs is a simpler version
# where we just consider connectivity without weights.
#
# Algorithm:
# Similar to Experiment 4, but with edges in both directions for undirected graph.

import heapq

def best_first_search_undirected_unweighted(graph, start, heuristic):
    priority_queue = [(heuristic[start], start)]
    visited = set()
    while priority_queue:
        _, node = heapq.heappop(priority_queue)
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            for neighbor in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (heuristic[neighbor], neighbor))
```

```
# Input undirected unweighted graph and heuristic values from user
graph = {}
heuristic = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v = input("Enter edge (u v): ").split()
    graph.setdefault(u, []).append(v)
    graph.setdefault(v, []).append(u)  # Add reverse edge for undirected

nodes = set(graph.keys())
for node in nodes:
    heuristic[node] = float(input(f"Enter heuristic value for {node}: "))

# Perform Best First Search
start_node = input("Enter start node: ")
best_first_search_undirected_unweighted(graph, start_node, heuristic)



# ===============================================
# EXPERIMENT 7: BEST FIRST SEARCH - DIRECTED WEIGHTED (KEYWORD: BFS_DIR_W)
# ===============================================
# Theory:
# Best First Search for directed weighted graphs considers direction and
# stores weights (though weights don't affect search order in basic Best First Search).
#
# Algorithm:
# Similar to Experiment 5, but without adding reverse edges.

import heapq

def best_first_search_directed_weighted(graph, start, heuristic):
    priority_queue = [(heuristic[start], start)]
    visited = set()
    while priority_queue:
        _, node = heapq.heappop(priority_queue)
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            for neighbor, weight in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (heuristic[neighbor], neighbor))

# Input directed weighted graph and heuristic values from user
graph = {}
heuristic = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v, weight = input("Enter edge (u v weight): ").split()
    weight = float(weight)
    graph.setdefault(u, []).append((v, weight))
    # No reverse edge since it's directed

nodes = set()
for node in graph:
    nodes.add(node)
    for neighbor, _ in graph[node]:
        nodes.add(neighbor)

for node in nodes:
    heuristic[node] = float(input(f"Enter heuristic value for {node}: "))

# Perform Best First Search
start_node = input("Enter start node: ")
best_first_search_directed_weighted(graph, start_node, heuristic)



# ===============================================
# EXPERIMENT 8: A* ALGORITHM - DIRECTED WEIGHTED FROM CSV (KEYWORD: ASTAR_DIR_W_CSV)
# ===============================================
# Theory:
# A* algorithm combines the advantages of Dijkstra's algorithm and Best First Search.
# It uses both the cost to reach a node (g(n)) and the heuristic estimate to the goal (h(n))
# to determine the order of node exploration: f(n) = g(n) + h(n).
#
# Algorithm:
# 1. Create a priority queue and add the starting node with f(n) = g(n) + h(n)
# 2. Create a visited set to track visited nodes
# 3. While priority queue is not empty:
#    a. Remove node with the lowest f(n) value
#    b. If node is goal, return success
#    c. If node is not visited, mark it as visited and process it
```

```
#    d. For each neighbor, calculate g(neighbor) = g(current) + cost(current, neighbor)
#    e. Add neighbor to priority queue with f(n) = g(neighbor) + h(neighbor)
# 4. Return failure if goal not found

import heapq
import pandas as pd

def a_star_directed_weighted_csv(graph, start, goal, heuristic):
    priority_queue = [(0 + heuristic[start], 0, start)]  # (f(n), g(n), node)
    visited = set()
    while priority_queue:
        f_n, g_n, node = heapq.heappop(priority_queue)
        if node == goal:
            print(f"Found goal: {goal}")
            return
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            for neighbor, cost in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (g_n + cost + heuristic[neighbor], g_n + cost, neighbor))

# Read directed weighted graph and heuristic from CSV
df_edges = pd.read_csv('graph_edges.csv', header=None)  # should have columns: source, target, weight
df_heuristic = pd.read_csv('heuristic.csv', header=None)  # should have columns: node, heuristic_value

graph = {}
heuristic = {}

for row in df_edges.values:
    u, v, weight = row
    weight = float(weight)
    graph.setdefault(u, []).append((v, weight))

for row in df_heuristic.values:
    node, h_value = row
    heuristic[node] = float(h_value)

# Perform A* Search
start_node = input("Enter start node: ")
goal_node = input("Enter goal node: ")
a_star_directed_weighted_csv(graph, start_node, goal_node, heuristic)



# ===============================================
# EXPERIMENT 9: A* ALGORITHM - DIRECTED WEIGHTED FROM USER (KEYWORD: ASTAR_DIR_W_USER)
# ===============================================
# Theory:
# Same as Experiment 8, but with manual input from user rather than CSV files.
#
# Algorithm:
# Same A* algorithm but with different input method.

import heapq

def a_star_directed_weighted_user(graph, start, goal, heuristic):
    priority_queue = [(0 + heuristic[start], 0, start)]  # (f(n), g(n), node)
    visited = set()
    while priority_queue:
        f_n, g_n, node = heapq.heappop(priority_queue)
        if node == goal:
            print(f"Found goal: {goal}")
            return
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            for neighbor, cost in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (g_n + cost + heuristic[neighbor], g_n + cost, neighbor))

# Input directed weighted graph and heuristic values from user
graph = {}
heuristic = {}
num_edges = int(input("Enter number of edges: "))
for _ in range(num_edges):
    u, v, cost = input("Enter edge (u v cost): ").split()
    cost = float(cost)
    graph.setdefault(u, []).append((v, cost))
    # No reverse edge since it's directed

nodes = set()
for node in graph:
```

```
        nodes.add(node)
        for neighbor, _ in graph[node]:
            nodes.add(neighbor)

    for node in nodes:
        heuristic[node] = float(input(f"Enter heuristic value for {node}: "))

    # Perform A* Search
    start_node = input("Enter start node: ")
    goal_node = input("Enter goal node: ")
    a_star_directed_weighted_user(graph, start_node, goal_node, heuristic)



    # ===============================================
    # EXPERIMENT 10: A* ALGORITHM - UNDIRECTED WEIGHTED FROM CSV (KEYWORD: ASTAR_UNDIR_W_CSV)
    # ===============================================
    # Theory:
    # A* algorithm for undirected weighted graphs, reading data from CSV files.
    #
    # Algorithm:
    # Same as Experiment 8, but treating the graph as undirected.

    import heapq
    import pandas as pd

    def a_star_undirected_weighted_csv(graph, start, goal, heuristic):
        priority_queue = [(0 + heuristic[start], 0, start)]  # (f(n), g(n), node)
        visited = set()
        while priority_queue:
            f_n, g_n, node = heapq.heappop(priority_queue)
            if node == goal:
                print(f"Found goal: {goal}")
                return
            if node not in visited:
                visited.add(node)
                print(node, end=" ")
                for neighbor, cost in graph[node]:
                    if neighbor not in visited:
                        heapq.heappush(priority_queue, (g_n + cost + heuristic[neighbor], g_n + cost, neighbor))

    # Read undirected weighted graph and heuristic from CSV
    df_edges = pd.read_csv('graph_edges.csv', header=None)  # columns: source, target, weight
    df_heuristic = pd.read_csv('heuristic.csv', header=None)  # columns: node, heuristic_value

    graph = {}
    heuristic = {}

    for row in df_edges.values:
        u, v, weight = row
        weight = float(weight)
        graph.setdefault(u, []).append((v, weight))
        graph.setdefault(v, []).append((u, weight))  # Add reverse edge for undirected

    for row in df_heuristic.values:
        node, h_value = row
        heuristic[node] = float(h_value)

    # Perform A* Search
    start_node = input("Enter start node: ")
    goal_node = input("Enter goal node: ")
    a_star_undirected_weighted_csv(graph, start_node, goal_node, heuristic)



    # ===============================================
    # EXPERIMENT 11: A* ALGORITHM - UNDIRECTED WEIGHTED FROM USER (KEYWORD: ASTAR_UNDIR_W_USER)
    # ===============================================
    # Theory:
    # A* algorithm for undirected weighted graphs with manual input from user.
    #
    # Algorithm:
    # Same as Experiment 9, but treating the graph as undirected.

    import heapq

    def a_star_undirected_weighted_user(graph, start, goal, heuristic):
        priority_queue = [(0 + heuristic[start], 0, start)]  # (f(n), g(n), node)
        visited = set()
        while priority_queue:
            f_n, g_n, node = heapq.heappop(priority_queue)
            if node == goal:
                print(f"Found goal: {goal}")
```

```
                return
            if node not in visited:
                visited.add(node)
                print(node, end=" ")
                for neighbor, cost in graph[node]:
                    if neighbor not in visited:
                        heapq.heappush(priority_queue, (g_n + cost + heuristic[neighbor], g_n + cost, neighbor))

    # Input undirected weighted graph and heuristic values from user
    graph = {}
    heuristic = {}
    num_edges = int(input("Enter number of edges: "))
    for _ in range(num_edges):
        u, v, cost = input("Enter edge (u v cost): ").split()
        cost = float(cost)
        graph.setdefault(u, []).append((v, cost))
        graph.setdefault(v, []).append((u, cost))  # Add reverse edge for undirected

    nodes = set()
    for node in graph:
        nodes.add(node)
        for neighbor, _ in graph[node]:
            nodes.add(neighbor)

    for node in nodes:
        heuristic[node] = float(input(f"Enter heuristic value for {node}: "))

    # Perform A* Search
    start_node = input("Enter start node: ")
    goal_node = input("Enter goal node: ")
    a_star_undirected_weighted_user(graph, start_node, goal_node, heuristic)



    # ===============================================
    # EXPERIMENT 12: FUZZY SET OPERATIONS (KEYWORD: FUZZY_BASIC)
    # ===============================================
    # Theory:
    # Fuzzy sets are sets whose elements have degrees of membership between 0 and 1.
    # Basic operations on fuzzy sets include:
    # - Union: max(μA(x), μB(x))
    # - Intersection: min(μA(x), μB(x))
    # - Complement: 1 - μA(x)
    #
    # Algorithm:
    # 1. Define fuzzy sets as dictionaries with elements as keys and membership degrees as values
    # 2. Implement union by taking the maximum membership degree for each element
    # 3. Implement intersection by taking the minimum membership degree for each element
    # 4. Implement complement by subtracting each membership degree from 1

    # Define three fuzzy sets
    A = {"a": 0.7, "b": 0.4, "c": 0.9}
    B = {"a": 0.5, "b": 0.6, "c": 0.3}
    C = {"a": 0.8, "b": 0.2, "c": 0.5}

    # Union
    union_AB = {key: max(A.get(key, 0), B.get(key, 0)) for key in set(A).union(B)}
    print("Union of A and B:", union_AB)

    # Intersection
    intersection_AB = {key: min(A.get(key, 0), B.get(key, 0)) for key in set(A).intersection(B)}
    print("Intersection of A and B:", intersection_AB)

    # Complement
    complement_A = {key: 1 - value for key, value in A.items()}
    print("Complement of A:", complement_A)


    # ===============================================
    # EXPERIMENT 13: FUZZY SET - DE MORGAN'S LAW (UNION) (KEYWORD: FUZZY_DEMORGAN1)
    # ===============================================
    # Theory:
    # De Morgan's Law for fuzzy sets states that:
    # Complement of Union: ¬(A ∪ B) = ¬A ∩ ¬B
    #
    # Algorithm:
    # 1. Compute complement of union: 1 - max(μA(x), μB(x))
    # 2. Compute intersection of complements: min(1 - μA(x), 1 - μB(x))
    # 3. Verify that both results are equal

    # Define two fuzzy sets
    A = {"a": 0.7, "b": 0.4, "c": 0.9}
    B = {"a": 0.5, "b": 0.6, "c": 0.3}
```

```
# Complement of Union
complement_union = {key: 1 - max(A.get(key, 0), B.get(key, 0)) for key in set(A).union(B)}

# Intersection of Complements
intersection_complements = {
    key: min(1 - A.get(key, 0), 1 - B.get(key, 0)) for key in set(A).union(B)
}

# Verify De Morgan's Law
print("Complement of Union:", complement_union)
print("Intersection of Complements:", intersection_complements)
print("De Morgan's Law Verified:", complement_union == intersection_complements)



# ===============================================
# EXPERIMENT 14: FUZZY SET - DE MORGAN'S LAW (INTERSECTION) (KEYWORD: FUZZY_DEMORGAN2)
# ===============================================
# Theory:
# De Morgan's Law for fuzzy sets also states that:
# Complement of Intersection: ¬(A ∩ B) = ¬A ∪ ¬B
#
# Algorithm:
# 1. Compute complement of intersection: 1 - min(μA(x), μB(x))
# 2. Compute union of complements: max(1 - μA(x), 1 - μB(x))
# 3. Verify that both results are equal

# Define two fuzzy sets
A = {"a": 0.7, "b": 0.4, "c": 0.9}
B = {"a": 0.5, "b": 0.6, "c": 0.3}

# Complement of Intersection
complement_intersection = {key: 1 - min(A.get(key, 0), B.get(key, 0)) for key in set(A).intersection(B)}

# Union of Complements
union_complements = {
    key: max(1 - A.get(key, 0), 1 - B.get(key, 0)) for key in set(A).union(B)
}

# Verify De Morgan's Law
print("Complement of Intersection:", complement_intersection)
print("Union of Complements:", union_complements)
print("De Morgan's Law Verified:", complement_intersection == union_complements)



# ===============================================
# EXPERIMENT 15: MINIMAX - NIM GAME (WIN OR DRAW) (KEYWORD: MINIMAX_WIN)
# ===============================================
# Theory:
# Minimax is a decision rule for minimizing the possible loss in a worst-case scenario.
# In the Nim game, players take turns removing 1-3 stones from a pile, and whoever
# takes the last stone wins. The minimax algorithm helps the computer make optimal moves.
#
# Algorithm:
# 1. Define a minimax function that evaluates game states recursively:
#    a. If terminal state (no stones left), return score (+1 for win, -1 for loss)
#    b. For maximizing player: choose the maximum score of all possible moves
#    c. For minimizing player: choose the minimum score of all possible moves
# 2. For each possible move, calculate the score using minimax
# 3. Choose the move with the highest score

def minimax_nim(stones, is_maximizing):
    # Base case: if no stones left, determine the winner
    if stones == 0:
        return -1 if is_maximizing else 1  # Loss for maximizer, win for minimizer

    # Try all possible moves (take 1, 2, or 3 stones)
    if is_maximizing:
        best_score = -float('inf')
        for move in range(1, min(4, stones + 1)):  # Take 1, 2, or 3 stones
            score = minimax_nim(stones - move, False)
            best_score = max(best_score, score)
        return best_score
    else:
        best_score = float('inf')
        for move in range(1, min(4, stones + 1)):
            score = minimax_nim(stones - move, True)
            best_score = min(best_score, score)
        return best_score

def find_best_move_nim(stones):
```

```
        best_move = -1
        best_score = -float('inf')
        for move in range(1, min(4, stones + 1)):  # Try taking 1, 2, or 3 stones
            score = minimax_nim(stones - move, False)  # Opponent's turn
            if score > best_score:
                best_score = score
                best_move = move
        return best_move

    # Play the game
    stones = 10  # Initial number of stones
    print(f"Game starts with {stones} stones.")

    while stones > 0:
        # Computer's turn
        move = find_best_move_nim(stones)
        print(f"Computer takes {move} stones.")
        stones -= move
        if stones <= 0:
            print("Computer wins!")
            break

        # Player's turn
        player_move = int(input(f"{stones} stones left. How many do you take (1-3)? "))
        while player_move < 1 or player_move > 3 or player_move > stones:
            print("Invalid move. Try again.")
            player_move = int(input(f"{stones} stones left. How many do you take (1-3)? "))
        stones -= player_move
        if stones <= 0:
            print("You win!")
            break



    # ===============================================
    # EXPERIMENT 16: MINIMAX - NIM GAME (LOSE OR DRAW) (KEYWORD: MINIMAX_LOSE)
    # ===============================================
    # Theory:
    # Minimax is a decision rule for minimizing the possible loss in a worst-case scenario.
    # In the Nim game, players take turns removing 1-3 stones from a pile, and whoever
    # takes the last stone wins. This implementation modifies the minimax algorithm to ensure
    # the computer either loses or draws - never wins.
    #
    # Algorithm:
    # 1. Define a minimax function that evaluates game states recursively:
    #    a. If terminal state (no stones left), return score (-1 for loss, +1 for win)
    #    b. For maximizing player: choose the maximum score of all possible moves
    #    c. For minimizing player: choose the minimum score of all possible moves
    # 2. For each possible move, calculate the score using minimax
    # 3. Choose the move with the LOWEST score to make the computer lose if possible
    #    (or draw if losing is not possible)

    def minimax_nim(stones, is_maximizing):
        # Base case: if no stones left, determine the winner
        if stones == 0:
            return -1 if is_maximizing else 1  # Loss for maximizer, win for minimizer

        # Try all possible moves (take 1, 2, or 3 stones)
        if is_maximizing:
            best_score = -float('inf')
            for move in range(1, min(4, stones + 1)):  # Take 1, 2, or 3 stones
                score = minimax_nim(stones - move, False)
                best_score = max(best_score, score)
            return best_score
        else:
            best_score = float('inf')
            for move in range(1, min(4, stones + 1)):
                score = minimax_nim(stones - move, True)
                best_score = min(best_score, score)
            return best_score

    def find_worst_move_nim(stones):
        # We're looking for a move that leads to a loss (or draw if no loss is possible)
        best_move = -1
        best_score = float('inf')  # We want the LOWEST score, not highest

        for move in range(1, min(4, stones + 1)):  # Try taking 1, 2, or 3 stones
            score = minimax_nim(stones - move, False)  # Opponent's turn
            if score < best_score:
                best_score = score
                best_move = move

        # If we can't find a losing move, just make any valid move
```

```
        if best_move == -1 and stones > 0:
            best_move = 1

        return best_move

    # Play the game
    stones = 10  # Initial number of stones
    print(f"Game starts with {stones} stones.")
    print("In this game, the player who takes the last stone wins!")

    while stones > 0:
        # Computer's turn
        move = find_worst_move_nim(stones)
        print(f"Computer takes {move} stones.")
        stones -= move
        print(f"{stones} stones remaining.")

        if stones <= 0:
            print("Computer took the last stone. Computer loses!")
            break

        # Player's turn
        player_move = int(input(f"How many stones do you take (1-3)? "))
        while player_move < 1 or player_move > 3 or player_move > stones:
            print("Invalid move. Try again.")
            player_move = int(input(f"How many stones do you take (1-3)? "))

        stones -= player_move
        print(f"{stones} stones remaining.")

        if stones <= 0:
            print("You took the last stone. You lose!")
            break

    # Note: In this game, taking the last stone means you LOSE
    # The computer will always try to make moves that lead to its own loss if possible


    # ===============================================
    # EXPERIMENT 17: MLP - N BINARY INPUTS, TWO HIDDEN LAYERS, ONE OUTPUT (RANDOM WEIGHTS) (KEYWORD: MLP_RANDOM)
    # ===============================================
    # Theory:
    # A Multi-Layer Perceptron (MLP) is a class of feedforward artificial neural network.
    # This experiment implements an MLP with N binary inputs, two hidden layers, and one output.
    # The weights and biases are randomly initialized and not trained, demonstrating the initial
    # state of a neural network before training.
    #
    # Algorithm:
    # 1. Initialize random weights and biases for all layers
    # 2. Define the network architecture:
    #    a. Input layer with N binary inputs
    #    b. Two hidden layers with specified number of neurons
    #    c. One output layer with sigmoid activation
    # 3. Implement forward propagation:
    #    a. Compute weighted sum for each layer
    #    b. Apply activation function (sigmoid)
    # 4. Display the random weights and biases
    # 5. Test the network with all possible binary inputs

    import numpy as np

    class MLP_Random:
        def __init__(self, n_inputs, hidden1_size, hidden2_size):
            self.n_inputs = n_inputs
            self.hidden1_size = hidden1_size
            self.hidden2_size = hidden2_size

            # Initialize random weights and biases
            self.weights1 = np.random.rand(n_inputs, hidden1_size)
            self.bias1 = np.random.rand(1, hidden1_size)

            self.weights2 = np.random.rand(hidden1_size, hidden2_size)
            self.bias2 = np.random.rand(1, hidden2_size)

            self.weights3 = np.random.rand(hidden2_size, 1)
            self.bias3 = np.random.rand(1, 1)

        def sigmoid(self, x):
            return 1 / (1 + np.exp(-x))

        def forward(self, X):
            # First hidden layer
            self.z1 = np.dot(X, self.weights1) + self.bias1
```

```
        self.a1 = self.sigmoid(self.z1)

        # Second hidden layer
        self.z2 = np.dot(self.a1, self.weights2) + self.bias2
        self.a2 = self.sigmoid(self.z2)

        # Output layer
        self.z3 = np.dot(self.a2, self.weights3) + self.bias3
        self.a3 = self.sigmoid(self.z3)

        return self.a3

def experiment17():
    n_inputs = int(input("Enter number of binary inputs (N): "))
    hidden1_size = n_inputs + 2
    hidden2_size = n_inputs

    mlp = MLP_Random(n_inputs, hidden1_size, hidden2_size)

    # Generate all possible binary input combinations
    X = np.array([list(map(int, format(i, f'0{n_inputs}b'))) for i in range(2**n_inputs)])

    # Forward pass
    outputs = mlp.forward(X)
    predicted = (outputs >= 0.5).astype(int)

    print("\nFinal weights and biases:")
    print("Layer 1 Weights:\n", mlp.weights1)
    print("Layer 1 Bias:\n", mlp.bias1)
    print("\nLayer 2 Weights:\n", mlp.weights2)
    print("Layer 2 Bias:\n", mlp.bias2)
    print("\nOutput Layer Weights:\n", mlp.weights3)
    print("Output Layer Bias:\n", mlp.bias3)

    print("\nTesting the network:")
    for i, inputs in enumerate(X):
        print(f"Input: {inputs}, Output: {outputs[i][0]:.4f}, Predicted: {predicted[i][0]}")

experiment17()


# ===============================================
# EXPERIMENT 18: MLP - 4 BINARY INPUTS, ONE HIDDEN LAYER, TWO OUTPUTS (KEYWORD: MLP_4IN_2OUT)
# ===============================================
# Theory:
# This MLP variation has a fixed architecture with 4 binary inputs, one hidden layer,
# and two binary outputs. The network demonstrates how multiple outputs can be learned
# simultaneously, with each output representing a different binary classification task.
#
# Algorithm:
# 1. Initialize random weights and biases for:
#    a. Input to hidden layer connections
#    b. Hidden to output layer connections
# 2. Implement forward propagation:
#    a. Compute hidden layer activations using sigmoid
#    b. Compute output layer activations using sigmoid
# 3. Display the network architecture and parameters
# 4. Test with all possible 4-bit input combinations
import numpy as np

class MLP_4Inputs_2Outputs:
    def __init__(self, hidden_size):
        self.n_inputs = 4
        self.hidden_size = hidden_size
        self.n_outputs = 2

        # Initialize random weights and biases
        self.weights1 = np.random.rand(self.n_inputs, hidden_size)
        self.bias1 = np.random.rand(1, hidden_size)

        self.weights2 = np.random.rand(hidden_size, self.n_outputs)
        self.bias2 = np.random.rand(1, self.n_outputs)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, X):
        # Hidden layer
        self.z1 = np.dot(X, self.weights1) + self.bias1
        self.a1 = self.sigmoid(self.z1)

        # Output layer
        self.z2 = np.dot(self.a1, self.weights2) + self.bias2
```

```
        self.a2 = self.sigmoid(self.z2)

        return self.a2

def experiment18():
    hidden_size = 6  # Can be adjusted

    mlp = MLP_4Inputs_2Outputs(hidden_size)

    # Generate all possible binary input combinations for 4 inputs
    X = np.array([list(map(int, format(i, '04b'))) for i in range(16)])

    # Forward pass
    outputs = mlp.forward(X)
    predicted = (outputs >= 0.5).astype(int)

    print("\nFinal weights and biases:")
    print("Hidden Layer Weights:\n", mlp.weights1)
    print("Hidden Layer Bias:\n", mlp.bias1)
    print("\nOutput Layer Weights:\n", mlp.weights2)
    print("Output Layer Bias:\n", mlp.bias2)

    print("\nTesting the network:")
    for i, inputs in enumerate(X):
        print(f"Input: {inputs}, Outputs: {outputs[i]}, Predicted: {predicted[i]}")

experiment18()


# ===============================================
# EXPERIMENT 19: MULTI-LAYER PERCEPTRON - N BINARY INPUTS, TWO HIDDEN LAYERS, ONE OUTPUT (SIGMOID)
# ===============================================
# Theory:
# This experiment implements a Multi-Layer Perceptron with N binary inputs, two hidden layers,
# and one output. Backpropagation is used to train the network with the Sigmoid function
# as the activation function.
#
# Algorithm (Backpropagation):
# 1. Initialize random weights and biases
# 2. Forward pass: Compute output of the network
# 3. Compute error at the output layer
# 4. Backward pass: Propagate error backward through the network
# 5. Update weights and biases using the computed gradients
# 6. Repeat until convergence or max epochs reached

import numpy as np

class MLP_Backprop_Sigmoid:
    def __init__(self, n_inputs, hidden1_size, hidden2_size):
        # Initialize network architecture
        self.n_inputs = n_inputs
        self.hidden1_size = hidden1_size
        self.hidden2_size = hidden2_size

        # Initialize weights and biases with random values
        self.weights1 = np.random.randn(n_inputs, hidden1_size) * 0.01
        self.bias1 = np.zeros((1, hidden1_size))

        self.weights2 = np.random.randn(hidden1_size, hidden2_size) * 0.01
        self.bias2 = np.zeros((1, hidden2_size))

        self.weights3 = np.random.randn(hidden2_size, 1) * 0.01
        self.bias3 = np.zeros((1, 1))

    def sigmoid(self, x):
        """Sigmoid activation function"""
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        """Derivative of sigmoid function"""
        return x * (1 - x)

    def forward(self, X):
        """Forward pass through the network"""
        # First hidden layer
        self.z1 = np.dot(X, self.weights1) + self.bias1
        self.a1 = self.sigmoid(self.z1)

        # Second hidden layer
        self.z2 = np.dot(self.a1, self.weights2) + self.bias2
        self.a2 = self.sigmoid(self.z2)

        # Output layer
```

```
        self.z3 = np.dot(self.a2, self.weights3) + self.bias3
        self.a3 = self.sigmoid(self.z3)

        return self.a3

    def backward(self, X, y, output, learning_rate):
        """Backward pass to update weights and biases"""
        m = X.shape[0]  # Number of training examples

        # Output layer error
        dz3 = output - y
        dw3 = np.dot(self.a2.T, dz3) / m
        db3 = np.sum(dz3, axis=0, keepdims=True) / m

        # Second hidden layer error
        dz2 = np.dot(dz3, self.weights3.T) * self.sigmoid_derivative(self.a2)
        dw2 = np.dot(self.a1.T, dz2) / m
        db2 = np.sum(dz2, axis=0, keepdims=True) / m

        # First hidden layer error
        dz1 = np.dot(dz2, self.weights2.T) * self.sigmoid_derivative(self.a1)
        dw1 = np.dot(X.T, dz1) / m
        db1 = np.sum(dz1, axis=0, keepdims=True) / m

        # Update weights and biases
        self.weights3 -= learning_rate * dw3
        self.bias3 -= learning_rate * db3

        self.weights2 -= learning_rate * dw2
        self.bias2 -= learning_rate * db2

        self.weights1 -= learning_rate * dw1
        self.bias1 -= learning_rate * db1

    def train(self, X, y, learning_rate=0.1, epochs=1000):
        """Train the network using backpropagation"""
        X = np.array(X)
        y = np.array(y).reshape(-1, 1)

        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Backward pass and update weights
            self.backward(X, y, output, learning_rate)

            # Calculate and print error every 100 epochs
            if epoch % 100 == 0:
                error = np.mean(np.abs(output - y))
                print(f"Epoch {epoch}, Error: {error}")

        return epochs

def experiment19():
    # Get number of inputs from user
    n_inputs = int(input("Enter number of binary inputs (N): "))

    # Configure network architecture
    hidden1_size = n_inputs + 2  # Simple heuristic for hidden layer size
    hidden2_size = n_inputs      # Second hidden layer slightly smaller

    # Create MLP
    mlp = MLP_Backprop_Sigmoid(n_inputs, hidden1_size, hidden2_size)

    # Generate all possible binary input combinations
    X = np.array([list(map(int, format(i, f'0{n_inputs}b'))) for i in range(2**n_inputs)])

    # For demo purposes, use XOR for 2 inputs, or more complex function for more inputs
    if n_inputs == 2:
        # XOR function
        y = np.array([int(sum(x) == 1) for x in X])
    else:
        # For more inputs, use a function that's 1 if more than half inputs are 1
        y = np.array([int(sum(x) > n_inputs/2) for x in X])

    # Train the MLP
    epochs = mlp.train(X, y)

    # Display results
    print(f"\nTraining completed in {epochs} epochs")
    print("\nFinal weights and biases:")
    print("Layer 1 Weights:")
```

```
    print(mlp.weights1)
    print("Layer 1 Bias:")
    print(mlp.bias1)
    print("\nLayer 2 Weights:")
    print(mlp.weights2)
    print("Layer 2 Bias:")
    print(mlp.bias2)
    print("\nOutput Layer Weights:")
    print(mlp.weights3)
    print("Output Layer Bias:")
    print(mlp.bias3)

    # Test the network with all possible inputs
    print("\nTesting the network:")
    outputs = mlp.forward(X)
    predicted = (outputs >= 0.5).astype(int)

    for i, inputs in enumerate(X):
        print(f"Input: {inputs}, Raw Output: {outputs[i][0]:.4f}, Predicted: {predicted[i][0]}, Expected: {y[i]}")

# Run the experiment
experiment19()


# ===============================================
# EXPERIMENT 20: MULTI-LAYER PERCEPTRON - N BINARY INPUTS, TWO HIDDEN LAYERS, ONE OUTPUT (RELU)
# ===============================================
# Theory:
# This experiment implements a Multi-Layer Perceptron with N binary inputs, two hidden layers,
# and one output. Backpropagation is used to train the network with the ReLU (Rectified Linear Unit)
# function as the activation function.
#
# Algorithm (Backpropagation with ReLU):
# 1. Initialize random weights and biases
# 2. Forward pass: Compute output of the network using ReLU activation for hidden layers
# 3. Compute error at the output layer
# 4. Backward pass: Propagate error backward through the network
# 5. Update weights and biases using the computed gradients
# 6. Repeat until convergence or max epochs reached

import numpy as np

class MLP_Backprop_ReLU:
    def __init__(self, n_inputs, hidden1_size, hidden2_size):
        # Initialize network architecture
        self.n_inputs = n_inputs
        self.hidden1_size = hidden1_size
        self.hidden2_size = hidden2_size

        # Initialize weights and biases with random values
        # He initialization for ReLU
        self.weights1 = np.random.randn(n_inputs, hidden1_size) * np.sqrt(2.0/n_inputs)
        self.bias1 = np.zeros((1, hidden1_size))

        self.weights2 = np.random.randn(hidden1_size, hidden2_size) * np.sqrt(2.0/hidden1_size)
        self.bias2 = np.zeros((1, hidden2_size))

        self.weights3 = np.random.randn(hidden2_size, 1) * np.sqrt(2.0/hidden2_size)
        self.bias3 = np.zeros((1, 1))

    def relu(self, x):
        """ReLU activation function"""
        return np.maximum(0, x)

    def relu_derivative(self, x):
        """Derivative of ReLU function"""
        return np.where(x > 0, 1, 0)

    def sigmoid(self, x):
        """Sigmoid activation function for output layer"""
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        """Derivative of sigmoid function"""
        return x * (1 - x)

    def forward(self, X):
        """Forward pass through the network"""
        # First hidden layer with ReLU
        self.z1 = np.dot(X, self.weights1) + self.bias1
        self.a1 = self.relu(self.z1)

        # Second hidden layer with ReLU
```

```
        self.z2 = np.dot(self.a1, self.weights2) + self.bias2
        self.a2 = self.relu(self.z2)

        # Output layer with sigmoid for binary classification
        self.z3 = np.dot(self.a2, self.weights3) + self.bias3
        self.a3 = self.sigmoid(self.z3)

        return self.a3

    def backward(self, X, y, output, learning_rate):
        """Backward pass to update weights and biases"""
        m = X.shape[0]  # Number of training examples

        # Output layer error (using sigmoid derivative)
        dz3 = output - y
        dw3 = np.dot(self.a2.T, dz3) / m
        db3 = np.sum(dz3, axis=0, keepdims=True) / m

        # Second hidden layer error (using ReLU derivative)
        dz2 = np.dot(dz3, self.weights3.T) * self.relu_derivative(self.z2)
        dw2 = np.dot(self.a1.T, dz2) / m
        db2 = np.sum(dz2, axis=0, keepdims=True) / m

        # First hidden layer error (using ReLU derivative)
        dz1 = np.dot(dz2, self.weights2.T) * self.relu_derivative(self.z1)
        dw1 = np.dot(X.T, dz1) / m
        db1 = np.sum(dz1, axis=0, keepdims=True) / m

        # Update weights and biases
        self.weights3 -= learning_rate * dw3
        self.bias3 -= learning_rate * db3

        self.weights2 -= learning_rate * dw2
        self.bias2 -= learning_rate * db2

        self.weights1 -= learning_rate * dw1
        self.bias1 -= learning_rate * db1

    def train(self, X, y, learning_rate=0.01, epochs=2000):
        """Train the network using backpropagation"""
        X = np.array(X)
        y = np.array(y).reshape(-1, 1)

        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Backward pass and update weights
            self.backward(X, y, output, learning_rate)

            # Calculate and print error every 100 epochs
            if epoch % 100 == 0:
                error = np.mean(np.abs(output - y))
                print(f"Epoch {epoch}, Error: {error}")

        return epochs

def experiment20():
    # Get number of inputs from user
    n_inputs = int(input("Enter number of binary inputs (N): "))

    # Configure network architecture
    hidden1_size = n_inputs * 2  # ReLU typically needs more neurons
    hidden2_size = n_inputs      # Second hidden layer

    # Create MLP with ReLU
    mlp = MLP_Backprop_ReLU(n_inputs, hidden1_size, hidden2_size)

    # Generate all possible binary input combinations
    X = np.array([list(map(int, format(i, f'0{n_inputs}b'))) for i in range(2**n_inputs)])

    # For demo purposes, use a function that's 1 if odd number of 1's (parity function)
    y = np.array([int(sum(x) % 2 == 1) for x in X])

    # Train the MLP
    epochs = mlp.train(X, y)

    # Display results
    print(f"\nTraining completed in {epochs} epochs")
    print("\nFinal weights and biases:")
    print("Layer 1 Weights:")
    print(mlp.weights1)
```

```
        print("Layer 1 Bias:")
        print(mlp.bias1)
        print("\nLayer 2 Weights:")
        print(mlp.weights2)
        print("Layer 2 Bias:")
        print(mlp.bias2)
        print("\nOutput Layer Weights:")
        print(mlp.weights3)
        print("Output Layer Bias:")
        print(mlp.bias3)

        # Test the network with all possible inputs
        print("\nTesting the network:")
        outputs = mlp.forward(X)
        predicted = (outputs >= 0.5).astype(int)

        for i, inputs in enumerate(X):
            print(f"Input: {inputs}, Raw Output: {outputs[i][0]:.4f}, Predicted: {predicted[i][0]}, Expected: {y[i]}")

    # Run the experiment
    experiment20()


    # ===============================================
    # EXPERIMENT 21: MLP - N BINARY INPUTS, TWO HIDDEN LAYERS, ONE OUTPUT (TANH) (KEYWORD: MLP_TANH)
    # ===============================================
    # Theory:
    # This MLP uses hyperbolic tangent (tanh) activation functions in the hidden layers,
    # which outputs values between -1 and 1. Tanh can help with faster convergence in some
    # cases compared to sigmoid. The output layer still uses sigmoid for binary classification.
    #
    # Algorithm:
    # 1. Initialize weights and biases with appropriate scaling for tanh
    # 2. Implement forward propagation:
    #    a. First hidden layer with tanh activation
    #    b. Second hidden layer with tanh activation
    #    c. Output layer with sigmoid activation
    # 3. Implement backpropagation:
    #    a. Compute gradients using tanh derivatives for hidden layers
    #    b. Update weights using gradient descent
    # 4. Train on all possible binary input combinations
    # 5. Display final weights and test performance
    import numpy as np

    class MLP_Backprop_Tanh:
        def __init__(self, n_inputs, hidden1_size, hidden2_size):
            self.n_inputs = n_inputs
            self.hidden1_size = hidden1_size
            self.hidden2_size = hidden2_size

            # Initialize weights and biases
            self.weights1 = np.random.randn(n_inputs, hidden1_size) * 0.01
            self.bias1 = np.zeros((1, hidden1_size))

            self.weights2 = np.random.randn(hidden1_size, hidden2_size) * 0.01
            self.bias2 = np.zeros((1, hidden2_size))

            self.weights3 = np.random.randn(hidden2_size, 1) * 0.01
            self.bias3 = np.zeros((1, 1))

        def tanh(self, x):
            return np.tanh(x)

        def tanh_derivative(self, x):
            return 1 - np.tanh(x)**2

        def sigmoid(self, x):
            return 1 / (1 + np.exp(-x))

        def forward(self, X):
            # First hidden layer with tanh
            self.z1 = np.dot(X, self.weights1) + self.bias1
            self.a1 = self.tanh(self.z1)

            # Second hidden layer with tanh
            self.z2 = np.dot(self.a1, self.weights2) + self.bias2
            self.a2 = self.tanh(self.z2)

            # Output layer with sigmoid
            self.z3 = np.dot(self.a2, self.weights3) + self.bias3
            self.a3 = self.sigmoid(self.z3)

            return self.a3
```

```
    def backward(self, X, y, output, learning_rate):
        m = X.shape[0]

        # Output layer error
        dz3 = output - y
        dw3 = np.dot(self.a2.T, dz3) / m
        db3 = np.sum(dz3, axis=0, keepdims=True) / m

        # Second hidden layer error
        dz2 = np.dot(dz3, self.weights3.T) * self.tanh_derivative(self.z2)
        dw2 = np.dot(self.a1.T, dz2) / m
        db2 = np.sum(dz2, axis=0, keepdims=True) / m

        # First hidden layer error
        dz1 = np.dot(dz2, self.weights2.T) * self.tanh_derivative(self.z1)
        dw1 = np.dot(X.T, dz1) / m
        db1 = np.sum(dz1, axis=0, keepdims=True) / m

        # Update weights and biases
        self.weights3 -= learning_rate * dw3
        self.bias3 -= learning_rate * db3

        self.weights2 -= learning_rate * dw2
        self.bias2 -= learning_rate * db2

        self.weights1 -= learning_rate * dw1
        self.bias1 -= learning_rate * db1

    def train(self, X, y, learning_rate=0.1, epochs=1000):
        X = np.array(X)
        y = np.array(y).reshape(-1, 1)

        for epoch in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output, learning_rate)

            if epoch % 100 == 0:
                error = np.mean(np.abs(output - y))
                print(f"Epoch {epoch}, Error: {error}")

        return epochs

def experiment21():
    n_inputs = int(input("Enter number of binary inputs (N): "))
    hidden1_size = n_inputs + 2
    hidden2_size = n_inputs

    mlp = MLP_Backprop_Tanh(n_inputs, hidden1_size, hidden2_size)

    # Generate all possible binary input combinations
    X = np.array([list(map(int, format(i, f'0{n_inputs}b'))) for i in range(2**n_inputs)])

    # Use XOR-like function for 2 inputs, parity function for more
    if n_inputs == 2:
        y = np.array([int(sum(x) == 1) for x in X])
    else:
        y = np.array([int(sum(x) % 2 == 1) for x in X])

    # Train the network
    epochs = mlp.train(X, y)

    print(f"\nTraining completed in {epochs} epochs")
    print("\nFinal weights and biases:")
    print("Layer 1 Weights:\n", mlp.weights1)
    print("Layer 1 Bias:\n", mlp.bias1)
    print("\nLayer 2 Weights:\n", mlp.weights2)
    print("Layer 2 Bias:\n", mlp.bias2)
    print("\nOutput Layer Weights:\n", mlp.weights3)
    print("Output Layer Bias:\n", mlp.bias3)

    print("\nTesting the network:")
    outputs = mlp.forward(X)
    predicted = (outputs >= 0.5).astype(int)

    for i, inputs in enumerate(X):
        print(f"Input: {inputs}, Output: {outputs[i][0]:.4f}, Predicted: {predicted[i][0]}, Expected: {y[i]}")

experiment21()
```

```
# ===============================================
# EXPERIMENT 22: TEXT PROCESSING PIPELINE (KEYWORD: TEXT_PROCESSING)
# ===============================================
# Theory:
# Text preprocessing is crucial for NLP tasks. This pipeline demonstrates fundamental
# text cleaning and normalization steps that convert raw text into a more analyzable form.
#
# Algorithm:
# 1. Text cleaning:
#    a. Remove punctuation and special characters using regex
#    b. Remove numbers and extra whitespace
#    c. Remove non-ASCII characters
# 2. Case normalization:
#    a. Convert all text to lowercase
# 3. Tokenization:
#    a. Split text into individual words/tokens
# 4. Stopword removal:
#    a. Filter out common words that carry little meaning
# 5. Spelling correction:
#    a. Identify and correct misspelled words
import re
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from spellchecker import SpellChecker

def experiment22():
    # Read text file
    with open('sample_text.txt', 'r', encoding='utf-8') as file:
        text = file.read()

    print("Original text:")
    print(text[:500], "...")  # Print first 500 characters

    # a. Text cleaning
    text = re.sub(r'\[.*?\]', '', text)  # Remove text in brackets
    text = re.sub(r'[%s]' % re.escape(string.punctuation), '', text)  # Remove punctuation
    text = re.sub(r'\w*\d\w*', '', text)  # Remove words with numbers
    text = re.sub(r'\s+', ' ', text)  # Remove extra whitespace
    text = re.sub(r'[^\x00-\x7F]+', '', text)  # Remove non-ASCII

    print("\nAfter cleaning:")
    print(text[:500], "...")

    # b. Convert to lowercase
    text = text.lower()
    print("\nAfter lowercase conversion:")
    print(text[:500], "...")

    # c. Tokenization
    tokens = word_tokenize(text)
    print("\nTokens (first 50):")
    print(tokens[:50])

    # d. Remove stopwords
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word not in stop_words]
    print("\nAfter stopword removal (first 50):")
    print(filtered_tokens[:50])

    # e. Correct misspelled words
    spell = SpellChecker()
    corrected_tokens = []
    for word in filtered_tokens:
        corrected = spell.correction(word)
        if corrected is not None and corrected != word:
            corrected_tokens.append(f"{word}({corrected})")
        else:
            corrected_tokens.append(word)

    print("\nWith spelling corrections:")
    print(corrected_tokens[:50])

# Note: You'll need to install nltk and spellchecker packages
# and download nltk data (stopwords, punkt) before running
experiment22()


# ===============================================
# EXPERIMENT 23: TEXT PROCESSING WITH STEMMING/LEMMATIZATION (KEYWORD: TEXT_STEM_LEMMA)
# ===============================================
# Theory:
# Stemming and lemmatization are text normalization techniques that reduce words to their
```

```
# base or root forms. Stemming uses heuristic rules, while lemmatization uses vocabulary
# and morphological analysis for more accurate results.
#
# Algorithm:
# 1. Perform basic text cleaning (punctuation, numbers, etc.)
# 2. Case normalization (lowercase conversion)
# 3. Apply stemming (Porter Stemmer algorithm)
# 4. Apply lemmatization (WordNet lemmatizer)
# 5. Generate n-grams (3 consecutive words) from lemmatized tokens
import re
import string
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.corpus import wordnet

def experiment23():
    # Read text file
    with open('sample_text.txt', 'r', encoding='utf-8') as file:
        text = file.read()

    # a. Text cleaning
    text = re.sub(r'\[.*?\]', '', text)
    text = re.sub(r'[%s]' % re.escape(string.punctuation), '', text)
    text = re.sub(r'\w*\d\w*', '', text)
    text = re.sub(r'\s+', ' ', text)
    text = re.sub(r'[^\x00-\x7F]+', '', text)

    # b. Convert to lowercase
    text = text.lower()

    # c. Stemming and Lemmatization
    tokens = word_tokenize(text)

    # Stemming
    ps = PorterStemmer()
    stemmed_tokens = [ps.stem(word) for word in tokens]
    print("Stemmed tokens (first 50):")
    print(stemmed_tokens[:50])

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    lemmatized_tokens = [lemmatizer.lemmatize(word) for word in tokens]
    print("\nLemmatized tokens (first 50):")
    print(lemmatized_tokens[:50])

    # d. Create list of 3 consecutive words after lemmatization
    trigrams = []
    for i in range(len(lemmatized_tokens) - 2):
        trigrams.append(f"{lemmatized_tokens[i]} {lemmatized_tokens[i+1]} {lemmatized_tokens[i+2]}")

    print("\nTrigrams (first 20):")
    print(trigrams[:20])

# Note: Requires nltk and wordnet data
experiment23()


# ===============================================
# EXPERIMENT 24: ONE-HOT ENCODING FOR TECHNICAL TEXTS (KEYWORD: ONE_HOT_ENCODING)
# ===============================================
# Theory:
# One-hot encoding represents text as binary vectors where each dimension corresponds to
# a word in the vocabulary. A document is represented by a vector with 1s for present words
# and 0s for absent words.
#
# Algorithm:
# 1. Read multiple technical text documents
# 2. Perform basic text cleaning
# 3. Create vocabulary of unique words across all documents
# 4. For each document:
#    a. Create binary vector with length equal to vocabulary size
#    b. Set vector elements to 1 for words present in document
# 5. Display vocabulary and encoded vectors
from sklearn.feature_extraction.text import CountVectorizer
import glob

def experiment24():
    # Read 3 technical text files
    files = glob.glob('technical_*.txt')
    documents = []

    for file in files[:3]:  # Process first 3 files
        with open(file, 'r', encoding='utf-8') as f:
```

```
            text = f.read()
            # Basic cleaning
            text = re.sub(r'[^\w\s]', '', text.lower())
            documents.append(text)

    # Create one-hot encoding
    vectorizer = CountVectorizer(binary=True)
    X = vectorizer.fit_transform(documents)

    print("Vocabulary size:", len(vectorizer.vocabulary_))
    print("\nFeature names (first 50):")
    print(vectorizer.get_feature_names_out()[:50])

    print("\nOne-hot encoded matrix:")
    print(X.toarray())

experiment24()


# ===============================================
# EXPERIMENT 25: BAG OF WORDS FOR MOVIE REVIEWS (KEYWORD: BAG_OF_WORDS)
# ===============================================
# Theory:
# Bag of Words (BoW) represents text as word frequency vectors, ignoring word order but
# maintaining multiplicity. It's a simple but effective text representation for many NLP tasks.
#
# Algorithm:
# 1. Read multiple movie review documents
# 2. Perform text cleaning and tokenization
# 3. Create vocabulary of unique words
# 4. For each document:
#    a. Count occurrences of each vocabulary word
#    b. Create frequency vector
# 5. Display vocabulary and BoW vectors
from sklearn.feature_extraction.text import CountVectorizer
import glob

def experiment25():
    # Read 3 movie review files
    files = glob.glob('review_*.txt')
    documents = []

    for file in files[:3]:
        with open(file, 'r', encoding='utf-8') as f:
            text = f.read()
            # Basic cleaning
            text = re.sub(r'[^\w\s]', '', text.lower())
            documents.append(text)

    # Create bag of words
    vectorizer = CountVectorizer()
    X = vectorizer.fit_transform(documents)

    print("Vocabulary size:", len(vectorizer.vocabulary_))
    print("\nFeature names (first 50):")
    print(vectorizer.get_feature_names_out()[:50])

    print("\nBag of words matrix:")
    print(X.toarray())

experiment25()


# ===============================================
# EXPERIMENT 26: TF-IDF FOR TOURIST PLACES (KEYWORD: TFIDF)
# ===============================================
# Theory:
# TF-IDF (Term Frequency-Inverse Document Frequency) measures word importance by considering:
# - Term Frequency (TF): how often a word appears in a document
# - Inverse Document Frequency (IDF): how rare a word is across all documents
# This helps highlight distinctive words in each document.
#
# Algorithm:
# 1. Read multiple documents about tourist places
# 2. Perform text preprocessing
# 3. Calculate TF for each word in each document
# 4. Calculate IDF for each word across all documents
# 5. Compute TF-IDF scores (TF * IDF)
# 6. Display vocabulary and TF-IDF vectors
from sklearn.feature_extraction.text import TfidfVectorizer
import glob

def experiment26():
```

```
    # Read 3 tourist place files
    files = glob.glob('tourist_*.txt')
    documents = []

    for file in files[:3]:
        with open(file, 'r', encoding='utf-8') as f:
            text = f.read()
            # Basic cleaning
            text = re.sub(r'[^\w\s]', '', text.lower())
            documents.append(text)

    # Create TF-IDF vectors
    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(documents)

    print("Vocabulary size:", len(vectorizer.vocabulary_))
    print("\nFeature names (first 50):")
    print(vectorizer.get_feature_names_out()[:50])

    print("\nTF-IDF matrix:")
    print(X.toarray())

experiment26()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-ea684ac10dfd> in <cell line: 0>()
     25     print(X.toarray())
     26
---> 27 experiment26()

4 frames
/usr/local/lib/python3.11/dist-packages/sklearn/feature_extraction/text.py in _count_vocab(self, raw_documents,
fixed_vocab)
   1280             vocabulary = dict(vocabulary)
   1281             if not vocabulary:
-> 1282                 raise ValueError(
   1283                     "empty vocabulary; perhaps the documents only contain stop words"
   1284                 )

ValueError: empty vocabulary; perhaps the documents only contain stop words
```

Start coding or generate with AI.