Question 1

Write a Python program to calculate the betweenness centrality of nodes in a given network graph.

Implement two versions:

- Using NetworkX
- Without using NetworkX (Implement betweenness centrality manually using shortest paths)
- Compare the results of both implementations. (25 + 25 marks)

```
graph = {
    "Alice": ["Bob", "Charlie", "David"],
    "Bob": ["Alice", "David", "Eve", "Hank"],
    "Charlie": ["Alice", "David", "Frank", "Grace"],
    "David": ["Alice", "Bob", "Charlie", "Eve", "Frank"],
    "Eve": ["Bob", "David", "Frank", "Ivy"],
    "Frank": ["Charlie", "David", "Eve", "Grace", "Ivy", "Jack"],
    "Grace": ["Charlie", "Frank", "Hank"],
    "Hank": ["Bob", "Grace", "Ivy", "Jack"],
    "Ivy": ["Eve", "Frank", "Hank", "Jack", "Kelly"],
    "Jack": ["Frank", "Hank", "Ivy", "Kelly", "Leo"],
    "Kelly": ["Ivy", "Jack", "Leo"],
    "Leo": ["Jack", "Kelly"]
}

import networkx as nx
from collections import defaultdict
from itertools import combinations

def networkx_betweenness_centrality(graph):
    G = nx.Graph(graph)

    return dict(nx.betweenness_centrality(G))

def manual_betweenness_centrality(graph):
    def bfs_shortest_paths(start):
        distances = {node: float('inf') for node in graph}
        distances[start] = 0
        queue = [start]
        paths = {node: [] for node in graph}
        paths[start] = [[start]]

        while queue:
            current = queue.pop(0)
            for neighbor in graph[current]:
                if distances[neighbor] > distances[current] + 1:
```

```
                    distances[neighbor] = distances[current] + 1
                    paths[neighbor] = [path + [neighbor] for path in
paths[current]]
                    queue.append(neighbor)
                elif distances[neighbor] == distances[current] + 1:
                    paths[neighbor].extend([path + [neighbor] for path
in paths[current]])

        return paths

    betweenness = {node: 0 for node in graph}

    for start in graph:
        all_paths = bfs_shortest_paths(start)

        for end in graph:
            if start == end:
                continue

            paths_between = all_paths[end]

            for path in paths_between:
                for intermediate in path[1:-1]:
                    betweenness[intermediate] += 1 /
len(paths_between)

    total_pairs = len(graph) * (len(graph) - 1) / 2
    return {node: score / total_pairs for node, score in
betweenness.items()}

networkx_result = networkx_betweenness_centrality(graph)
print("NetworkX Betweenness Centrality:")
for node, centrality in sorted(networkx_result.items(), key=lambda x:
x[1], reverse=True):
    print(f"{node}: {centrality:.4f}")

print("\n" + "="*50 + "\n")

manual_result = manual_betweenness_centrality(graph)
print("Manual Betweenness Centrality:")
for node, centrality in sorted(manual_result.items(), key=lambda x:
x[1], reverse=True):
    print(f"{node}: {centrality:.4f}")

print("\n" + "="*50 + "\n")

print("Difference between NetworkX and Manual Implementation:")
for node in graph:
    diff = abs(networkx_result[node] - manual_result[node])
    print(f"{node}: {diff:.4f}")
```

```
NetworkX Betweenness Centrality:
Frank: 0.2588
Jack: 0.2098
Hank: 0.1324
Ivy: 0.1265
Bob: 0.0808
David: 0.0717
Charlie: 0.0567
Eve: 0.0512
Grace: 0.0242
Kelly: 0.0152
Alice: 0.0091
Leo: 0.0000

==================================================

Manual Betweenness Centrality:
Frank: 0.4313
Jack: 0.3497
Hank: 0.2207
Ivy: 0.2109
Bob: 0.1346
David: 0.1194
Charlie: 0.0944
Eve: 0.0854
Grace: 0.0404
Kelly: 0.0253
Alice: 0.0152
Leo: 0.0000

==================================================

Difference between NetworkX and Manual Implementation:
Alice: 0.0061
Bob: 0.0538
Charlie: 0.0378
David: 0.0478
Eve: 0.0341
Frank: 0.1725
Grace: 0.0162
Hank: 0.0883
Ivy: 0.0843
Jack: 0.1399
Kelly: 0.0101
Leo: 0.0000
```

Question 2

Write a Python program to apply the Girvan-Newman Algorithm to detect communities in a given network graph.

Implement two versions:

- Using NetworkX
- Without using NetworkX (Manually remove edges with the highest betweenness and detect communities)
- Compare the results of both implementations. (25 + 25 marks)

```
import networkx as nx
import copy

def networkx_girvan_newman(graph):
    G = nx.Graph()
    for node, neighbors in graph.items():
        for neighbor in neighbors:
            G.add_edge(node, neighbor)

    communities_generator = nx.community.girvan_newman(G)
    top_level_communities = next(communities_generator)

    return [list(community) for community in top_level_communities]

def calculate_edge_betweenness(graph):
    nodes = list(graph.keys())
    edge_betweenness = {}

    for start in nodes:
        distances = {node: float('inf') for node in nodes}
        distances[start] = 0
        predecessors = {node: [] for node in nodes}
        path_counts = {node: 0 for node in nodes}
        path_counts[start] = 1

        queue = [start]
        while queue:
            current = queue.pop(0)
            for neighbor in graph[current]:
                if distances[neighbor] > distances[current] + 1:
                    distances[neighbor] = distances[current] + 1
                    predecessors[neighbor] = [current]
                    path_counts[neighbor] = path_counts[current]
                    queue.append(neighbor)
                elif distances[neighbor] == distances[current] + 1:
                    predecessors[neighbor].append(current)
                    path_counts[neighbor] += path_counts[current]
```

```
        node_credits = {node: 1 for node in nodes}
        for node in sorted(nodes, key=lambda x: distances[x],
reverse=True):
            for pred in predecessors[node]:
                edge = tuple(sorted((node, pred)))
                edge_betweenness[edge] = edge_betweenness.get(edge, 0)
+ node_credits[node]
                node_credits[pred] += node_credits[node]

    return edge_betweenness

def manual_girvan_newman(graph):
    def find_communities(current_graph):
        visited = set()
        communities = []

        def dfs(node, community):
            visited.add(node)
            community.append(node)
            for neighbor in current_graph.get(node, []):
                if neighbor not in visited:
                    dfs(neighbor, community)

        for node in current_graph:
            if node not in visited:
                community = []
                dfs(node, community)
                communities.append(community)

        return communities

    working_graph = copy.deepcopy(graph)

    communities = find_communities(working_graph)

    while len(communities) == 1:
        edge_betweenness = calculate_edge_betweenness(working_graph)

        max_betweenness_edge = max(edge_betweenness,
key=edge_betweenness.get)
        node1, node2 = max_betweenness_edge

        working_graph[node1] = [n for n in working_graph[node1] if n !
= node2]
        working_graph[node2] = [n for n in working_graph[node2] if n !
= node1]

        communities = find_communities(working_graph)

    return communities
```

```python
print("NetworkX Girvan-Newman Communities:")
networkx_communities = networkx_girvan_newman(graph)
for i, community in enumerate(networkx_communities, 1):
    print(f"Community {i}: {community}")

print("\nManual Girvan-Newman Communities:")
manual_communities = manual_girvan_newman(graph)
for i, community in enumerate(manual_communities, 1):
    print(f"Community {i}: {community}")
```

```
NetworkX Girvan-Newman Communities:
Community 1: ['Charlie', 'David', 'Grace', 'Eve', 'Alice', 'Frank',
'Bob']
Community 2: ['Leo', 'Kelly', 'Ivy', 'Hank', 'Jack']

Manual Girvan-Newman Communities:
Community 1: ['Alice', 'Bob', 'David', 'Charlie', 'Frank', 'Eve',
'Grace', 'Hank']
Community 2: ['Ivy', 'Jack', 'Kelly', 'Leo']
```