1. În cadrul unei rețele de socializare, dorim să identificăm grupuri de persoane care se cunosc între ele, direct sau indirect. Fiecare persoană este reprezentată ca un nod într-un graf neorientat, iar o muchie între două noduri indică faptul că cele două persoane se cunosc. Să se utilizeze algoritmul DFS pentru a identifica și a număra toate componentele conexe din graf. O componentă conexă este un subgraf în care orice două noduri sunt conectate prin căi, și niciun nod nu este conectat cu noduri din afara componentei.

   Date de Intrare: Un graf neorientat reprezentat ca o listă de adiacență.
   Sarcina: Implementați algoritmul DFS pentru a parcurge graficul și pentru a identifica toate componentele conexe.
   Programul trebuie să afișeze numărul total de componente conexe din graf.

Un exemplu de implementare a algoritmului DFS este :

```
# Inițializăm o variabilă globală pentru timp
time = 0
def dfs_visit(graph, u):
    global time
    time += 1  # Incrementăm timpul la începutul vizitei
    u.d = time  # Timpul de descoperire pentru u
    u.color = 'GRAY'  # Marcăm nodul u ca fiind în curs de explorare (gri)
    for v in graph[u]:
        if v.color == 'WHITE':  # Dacă v nu a fost vizitat
            v.pi = u  # Setăm predecesorul lui v la u
            dfs_visit(graph, v)  # Vizităm recursiv v
    u.color = 'BLACK'  # Marcăm nodul u ca fiind complet explorat (negru)
    time += 1  # Incrementăm timpul la finalul vizitei
    u.f = time  # Timpul de finalizare pentru u
# Clasa pentru un nod în graf
class Node:
    def __init__(self, name):
```

```
        self.name = name
        self.color = 'WHITE'  # Toate nodurile sunt inițial albe (nevizitate)
        self.d = 0  # Timpul de descoperire
        self.f = 0  # Timpul de finalizare
        self.pi = None  # Predecesorul nodului
# graf simplu pentru demonstrație
graph = {
    'u': ['v', 'x'],
    'v': ['y'],
    'x': ['v'],
    'y': ['x'],
    'w': ['y', 'z'],
    'z': ['z']
}
# Transformăm lista de adiacență într-un dicționar de noduri
graph_nodes = {k: Node(k) for k in graph.keys()}
for k, v in graph.items():
    graph_nodes[k] = [graph_nodes[vertex] for vertex in v]


# Apelăm dfs_visit pentru a vizita nodurile începând de la un nod ales
dfs_visit(graph_nodes, graph_nodes['u'])
```

2. Într-un oraș format din intersecții (noduri) și străzi (muchii), trebuie să găsiți cea mai scurtă cale de la o intersecție de start la o destinație. Scrieți un program care folosește algoritmul lui Dijkstra pentru a determina cea mai scurtă cale de la plecare la destinație. Afișați lungimea totală a căii.

    Date de Intrare:

        Numărul de Intersecții (N): Numărul total de intersecții în oraș.

        Numărul de Străzi (M): Câte străzi sunt în oraș.

Detalii Străzi: Pentru fiecare stradă, introduceți intersecția de start, de sfârșit și lungimea străzii (ex: A B 4).

Punctul de Plecare și Destinație: Specificați intersecția de plecare și cea de destinație.

Un exemplu de implementare in Python :

```
def dijkstra(graph, weights, start):
    # Inițializarea surselor unice; setarea distanțelor inițiale și a predecesorilor
    distances, predecessors = initialize_single_source(graph, start)
    # Setul S va ține evidența nodurilor vizitate
    S = set()
    # Coada de priorități Q va folosi un heap minim pentru a extrage nodul cu distanța minimă
    Q = [(0, start)]
    heapq.heapify(Q)
    while Q:
        # Operația de extragere a minimului
        current_distance, u = heapq.heappop(Q)
        # Dacă nodul nu a fost vizitat, îl procesăm
        if u not in S:
            S.add(u)
            # Pentru fiecare vecin v al lui u, efectuăm operația de relaxare
            for v in graph[u]:
                relax(u, v, weights, distances, predecessors)
                # Dacă v nu este în S, îl adăugăm în coada de priorități
                if v not in S:
                    heapq.heappush(Q, (distances[v], v))
    return distances, predecessors
def initialize_single_source(graph, start):
    # Inițializează toate distanțele ca fiind infinit și predecesorii ca fiind None
```

```
    distances = {node: float('infinity') for node in graph.keys()}
    predecessors = {node: None for node in graph.keys()}
    distances[start] = 0  # Distanța de la nodul de start la sine este 0
    return distances, predecessors
def relax(u, v, weights, distances, predecessors):
    # Dacă distanța cunoscută la v este mai mare decât distanța la u plus greutatea muchiei u-v
    if distances[v] > distances[u] + weights[(u, v)]:
        # Actualizează distanța la v
        distances[v] = distances[u] + weights[(u, v)]
        # Actualizează predecesorul lui v la u
        predecessors[v] = u
```