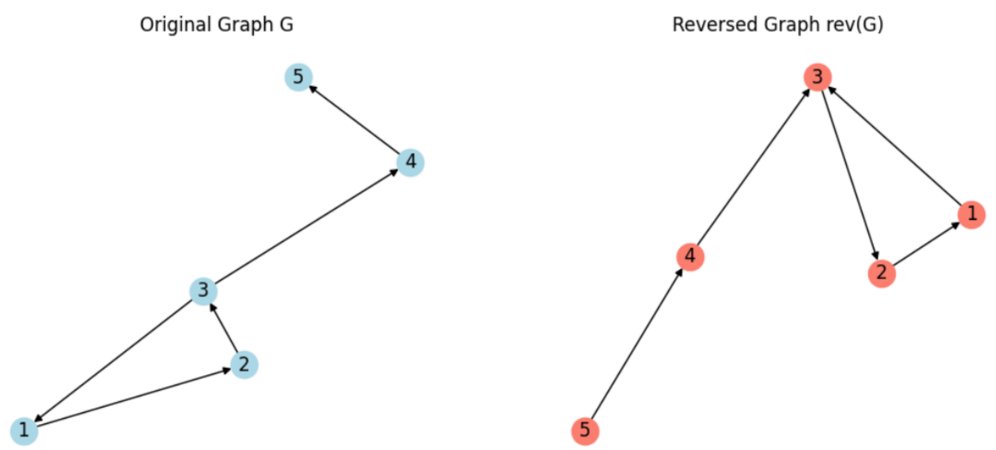
Problem1:

SCC and reversal, let G a directed graph:

1. Describe an algorithm to compute the reversal rev(G) of a directed graph G in O(V + E) time.

```
def reverse_graph(G):
    R = nx.DiGraph()
    R.add_nodes_from(G.nodes())
    for u, v in G.edges():
        R.add_edge(v, u)
    return R
```

   1. Creates a new directed graph R with all vertices from G
   2. For each edge (u, v) in G, adds edge (v, u) to R
   3. Time complexity: O(V) for vertices + O(E) for edges = O(V + E)



2. Prove that for every directed graph G, the strong component graph scc(G) 1 is acyclic.

```
#build SCC graph
def scc_graph(G):
    scc = list(nx.strongly_connected_components(G))
    index = {node: i for i, comp in enumerate(scc) for node in comp}
    SG = nx.DiGraph()
    SG.add_nodes_from(range(len(scc)))

    for u, v in G.edges():
        if index[u] != index[v]:
            SG.add_edge(index[u], index[v])
    return SG

SG = scc_graph(G)
```

1. Find all strongly connected components (SCCs) using Kosaraju/Tarjan
2. Create SCC graph where each SCC is a vertex
3. Add edge between SCCs if there's any edge between their vertices
4. Acyclicity proof: If SCC graph had a cycle, all SCCs on that cycle would be mutually reachable, contradicting maximality of SCCs

3. Prove that scc(rev(G)) = rev(scc(G)) for every directed graph G.

```
R = reverse_graph(G)
SG = scc_graph(G)

scc_of_revG = scc_graph(R)
rev_of_sccG = reverse_graph(SG)
```

1. rev(G) reverses all edges in G
2. SCCs remain the same in rev(G) (strong connectivity is preserved under edge reversal)
3. Edges between SCCs in scc(rev(G)) are exactly the reverses of edges in scc(G)
4. Therefore: scc(rev(G)) = rev(scc(G))

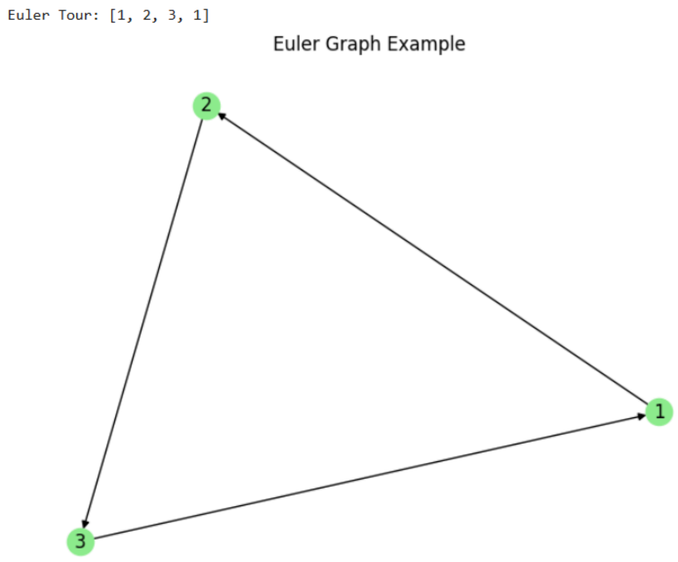
4. Fix an arbitrary directed graph G. For any vertex v of G, let S(v) denote the strong component of G that contains v. For all vertices u and v of G, prove that u can reach v in G if and only if S(u) can reach S(v) in scc(G).

```python
# S(v) is represented by index[v]
index = {node: i for i, comp in enumerate(scc) for node in comp}

# u can reach v in G if:
# Case 1: index[u] == index[v] (same SCC)
# Case 2: index[u] != index[v] and SCC index[u] can reach SCC index[v] in SG
```

1. Let S(v) be the SCC containing vertex v
2. Forward direction: If u → v path exists in G, traverse SCCs along this path to get S(u) → S(v) path in SCC(G)
3. Backward direction: If S(u) → S(v) path exists in SCC(G), use edges between SCCs to construct u → v path in G
4. Same SCC case: trivial reachability via SCC's internal cycles

Problem2:

```python
def euler_tour(G):
    if not nx.is_strongly_connected(G):
        raise ValueError("Graph is not strongly connected")

    for v in G.nodes():
        if G.in_degree(v) != G.out_degree(v):
            raise ValueError("In-degree != out-degree → no Euler t

    G_copy = G.copy()
    start = list(G.nodes())[0]
    stack = [start]
    circuit = []

    while stack:
        v = stack[-1]
        if G_copy.out_degree(v) > 0:
            u = next(iter(G_copy[v]))
            stack.append(u)
            G_copy.remove_edge(v, u)
        else:
            circuit.append(stack.pop())
    return circuit[::-1]

#Euler graph
E = nx.DiGraph()
E.add_edges_from([(1,2),(2,3),(3,1)])

tour = euler_tour(E)
print("Euler Tour:", tour)

nx.draw(E, with_labels=True, arrows=True, node_color="lightgreen")
plt.title("Euler Graph Example")
plt.show()
```



Problem3:

We are given:

A → B

A → C

B → C

B → D

C → E

D → E

D → F

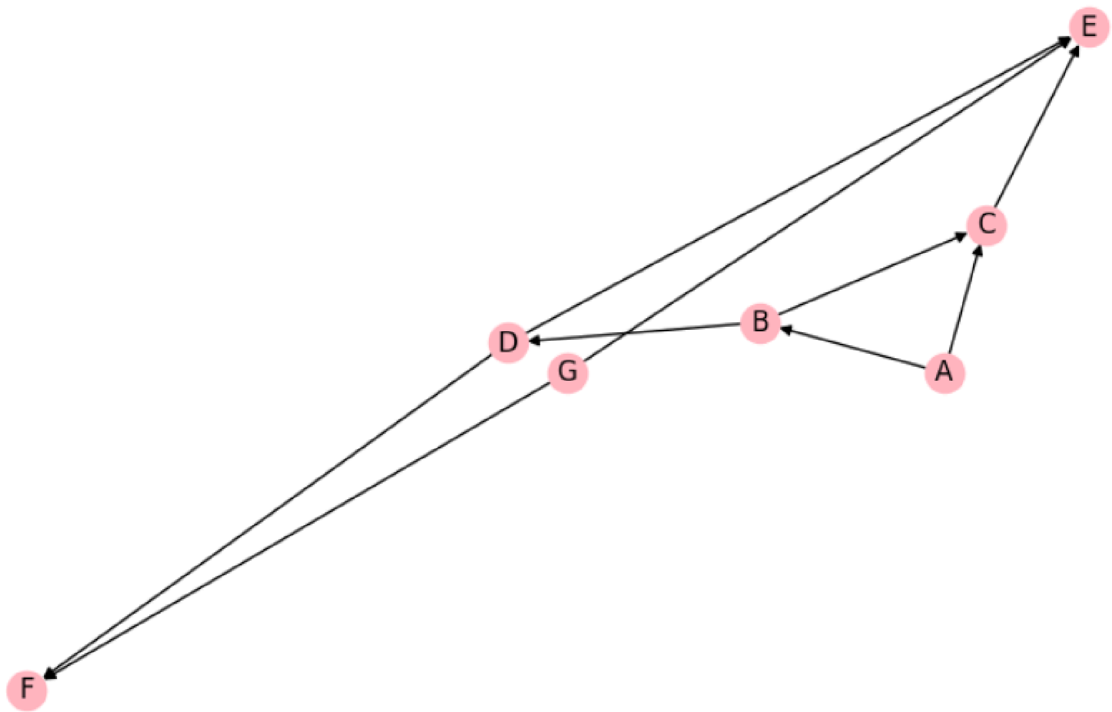
G → F

G → E

Topological Sort**: A linear ordering of vertices such that all edges u → v follow u before v. We compute:

- Start from A
- Use DFS-based or Kahn's Algorithm
- Multiple valid orderings exist

```
edges = [
    ("A","B"), ("A","C"),
    ("B","C"), ("B","D"),
    ("C","E"),
    ("D","E"), ("D","F"),
    ("G","F"), ("G","E")
]

G3 = nx.DiGraph()
G3.add_edges_from(edges)

plt.figure(figsize=(8,5))
nx.draw(G3, with_labels=True, node_color="lightpink", arrows=True)
plt.title("Course Prerequisite Graph")
plt.show()

# Topological sort
order = list(nx.topological_sort(G3))
print("One possible topological order:", order)
```




```
ne possible topological order: ['A', 'G', 'B', 'C', 'D', 'E', 'F']
```