R-6.1 .

Let's break down the requirements and reasoning for the graph GGG:

1. **Simple Undirected Graph**: No loops or multiple edges between the same pair of vertices.
2. **Vertices**: 12
3. **Edges**: 18
4. **Connected Components**: 3

Drawing the Graph

Given these constraints, we need to construct a graph that adheres to the specifications. Here's a step-by-step approach:

1. **Step 1: Distribute Vertices Among Components**: Let's distribute the 12 vertices among the 3 components. For simplicity, we can assume an even distribution (4 vertices per component), though other distributions are possible.
2. **Step 2: Add Edges to Each Component**: For each component to be connected:
    - A component with 4 vertices (let's call it C1C_1C1) needs a minimum of 3 edges to be connected (forming a tree).
    - If we add more edges to C1C_1C1, we can make it denser but still connected (up to (42)=6\binom{4}{2} = 6(24)=6 edges if fully connected).

Let's consider:

- Component 1: 4 vertices, 5 edges (one edge more than a tree, making it a small cycle).
- Component 2: 4 vertices, 6 edges (fully connected).
- Component 3: 4 vertices, 7 edges (fully connected plus one extra edge).

This gives us a total of 5+6+7=185 + 6 + 7 = 185+6+7=18 edges.

Why Impossible with 66 Edges

If GGG had 66 edges with 12 vertices and 3 connected components:

1. **Maximum Number of Edges in a Complete Graph**: The maximum number of edges in a simple undirected graph with 12 vertices (complete graph K12K_{12}K12) is given by:

(122)=12×112=66\binom{12}{2} = \frac{12 \times 11}{2} = 66(212)=212×11=66

This would imply that all vertices are interconnected in a single component, leaving no room for multiple connected components.

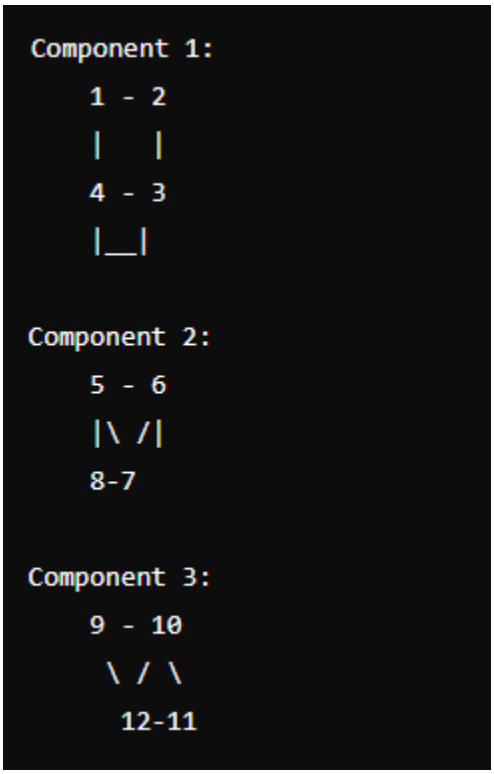
2. **Implication of 66 Edges**:
    - With 66 edges, the graph is fully connected, meaning there is only one connected component.

- Having 66 edges in a graph with 3 connected components would require each component to be fully connected and then some, which is impossible since all possible edges are used up in connecting every vertex.

Thus, having 66 edges would result in a single connected component rather than 3, making it impossible to satisfy the requirement of having 3 connected components.

Visual Representation

Here's a possible representation of the graph with the given constraints (vertices labeled from 1 to 12):



R-6.4.

To find a sequence of courses that allows Bob to satisfy all the prerequisites, we can represent the courses and their prerequisites as a directed graph where each course is a node, and an edge from node A to node B means that course A is a prerequisite for course B. We then need to find a topological order of this graph.

Here is the list of courses and their prerequisites:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15

- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

First, we list the courses:

- LA15
- LA16
- LA22
- LA31
- LA32
- LA126
- LA127
- LA141
- LA169

Next, we construct the directed graph of the prerequisites:

- LA15 → LA16
- LA15 → LA31
- LA16 → LA32
- LA16 → LA127
- LA16 → LA141
- LA31 → LA32
- LA32 → LA126
- LA32 → LA169
- LA22 → LA126
- LA22 → LA141

To find a topological order, we perform a topological sort on the graph. Here is the step-by-step process:

1. Identify nodes with no incoming edges (no prerequisites): LA15, LA22
2. Remove these nodes and their outgoing edges from the graph and add them to the sorted list.

3. Repeat the process with the remaining nodes.

Starting with nodes with no incoming edges: LA15, LA22

1. LA15 and LA22 have no prerequisites, so we start with them.
2. Remove LA15 and LA22 from the graph. The updated list of nodes and edges is:
    - LA16: LA15 (removed)
    - LA31: LA15 (removed)
    - LA32: LA16, LA31
    - LA126: LA22 (removed), LA32
    - LA127: LA16
    - LA141: LA22 (removed), LA16
    - LA169: LA32

Next nodes with no incoming edges: LA16, LA31

1. Add LA16 and LA31 to the list.
2. Remove LA16 and LA31 from the graph. The updated list of nodes and edges is:
    - LA32: LA16 (removed), LA31 (removed)
    - LA126: LA32
    - LA127: LA16 (removed)
    - LA141: LA16 (removed)
    - LA169: LA32

Next nodes with no incoming edges: LA32

1. Add LA32 to the list.
2. Remove LA32 from the graph. The updated list of nodes and edges is:
    - LA126: LA32 (removed)
    - LA127
    - LA141
    - LA169: LA32 (removed)

Next nodes with no incoming edges: LA127, LA141, LA169

1. Add LA127, LA141, and LA169 to the list.
2. Remove LA127, LA141, and LA169 from the graph. The updated list of nodes and edges is:

- LA126

Next node with no incoming edges: LA126

1. Add LA126 to the list.
2. Remove LA126 from the graph. The updated list of nodes and edges is empty.

The final topological order (sequence of courses) that satisfies all the prerequisites is:

1. LA15
2. LA22
3. LA16
4. LA31
5. LA32
6. LA127
7. LA141
8. LA169
9. LA126

So, Bob should take the courses in the following sequence to satisfy all prerequisites:

LA15, LA22, LA16, LA31, LA32, LA127, LA141, LA169, LA126.

R-6.7.

To determine whether to use an adjacency list or an adjacency matrix, we need to consider the characteristics of each structure and the specific requirements given in each case.

a. The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

Use Adjacency List

Justification:

- **Space Complexity**:
  - **Adjacency List**: O(V+E)O(V + E)O(V+E), where VVV is the number of vertices and EEE is the number of edges.
  - **Adjacency Matrix**: O(V2)O(V^2)O(V2).

For 10,000 vertices (V=10,000V = 10,000V=10,000) and 20,000 edges (E=20,000E = 20,000E=20,000):

- Adjacency List: O(10,000+20,000)=O(30,000)O(10,000 + 20,000) = O(30,000)O(10,000+20,000)=O(30,000).
- Adjacency Matrix: O(10,0002)=O(100,000,000)O(10,000^2) = O(100,000,000)O(10,0002)=O(100,000,000).

- The adjacency list is much more space-efficient for this sparse graph (since EEE is much smaller than V2V^2V2).

b. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

Use Adjacency List

Justification:

- **Space Complexity**:
  - **Adjacency List**: O(V+E)O(V + E)O(V+E).
  - **Adjacency Matrix**: O(V2)O(V^2)O(V2).

For 10,000 vertices and 20,000,000 edges:

- Adjacency List: O(10,000+20,000,000)=O(20,010,000)O(10,000 + 20,000,000) = O(20,010,000)O(10,000+20,000,000)=O(20,010,000).
- Adjacency Matrix: O(10,0002)=O(100,000,000)O(10,000^2) = O(100,000,000)O(10,0002)=O(100,000,000).

- Even though the graph is denser, the adjacency list is still more space-efficient than the adjacency matrix.

c. You need to answer the query areAdjacent as fast as possible, no matter how much space you use.

Use Adjacency Matrix

Justification:

- **Query Time Complexity**:
  - **Adjacency List**: O(V)O(V)O(V) in the worst case, if you need to scan through all adjacent vertices.
  - **Adjacency Matrix**: O(1)O(1)O(1), since you can directly access any element in the matrix.

- If the primary concern is the speed of the areAdjacent query and space is not a constraint, the adjacency matrix is preferable because it allows for constant-time O(1)O(1)O(1) adjacency checks.