1) **Write a Gremlin command** that creates the above graph

```
g=TinkerGraph.open();
CS101=g.addVertex(id,"CS101")
CS201=g.addVertex(id,"CS201")
CS220=g.addVertex(id,"CS220")
CS334=g.addVertex(id,"CS334")
CS420=g.addVertex(id,"CS420")
CS681=g.addVertex(id,"CS681")
CS400=g.addVertex(id,"CS400")
CS526=g.addVertex(id,"CS526")
CS201.addEdge("prereq",CS101)
CS220.addEdge("prereq",CS201)
CS334.addEdge("prereq",CS201)
CS420.addEdge("prereq",CS220)
CS681.addEdge("prereq",CS334)
CS400.addEdge("prereq",CS334)
CS526.addEdge("prereq",CS400)
CS420.addEdge("coreqs",CS220)
CS526.addEdge("coreqs",CS400)
```

Output:
graphtraversalsource[tinkergraph[vertices:8 edges:9], standard]

Explanation:
- The first line of command creates a tinker graph and assigns it to a graph variable called g
- Next 8 lines will create vertex and assign it to the respective variables who's id's are given by id as a key/value pair
- Next 7 lines will create edges between the vertices of the graph specified with edge label as the first argument and vertex as the second argument. For example: addEdge is creating an edge between CS201 to CS101 with the label prereq.
- The last 2 line is creating another edge between nodes CS420 and CS220, and CS526 and CS400 labeled by coreqs.
- The output for the above query can be determined by the command g and generated output is shown above

2)**Write a query** that will output JUST the doubly-connected nodes

```
g.traversal().V().as('from').out('prereq').as('to').in('coreqs').select('from','to')
```

OR

```
g.traversal().V().as('a').out('prereq').as('b').in('coreqs').select('a','b')
```

Output:
[from:v[CS420],to:v[CS220]]
[from:v[CS526],to:v[CS400]]

Output: for the second/alternative query
[a:v[CS420],b:v[CS220]]
[a:v[CS526],b:v[CS400]]

Explanation:
Note: I have used from and to as the label for the node.
This could be changed to a and b to get the output as mentioned in the specifications.
For each vertices of the graph compute the following specified by V():

- assign name using 'as' command to the previous step (name is "from")
- out command computes all the out adjacent vertices to the vertex from the previous step with the edge labeled prereq, i.e V().as ('from') step
- "as" command names the previous step to 'to'
- "in" command will further get the adjacent vertices to the vertex whose edge is labeled as coreqs.
- Select command will select all such vertices and display it in from and to format

3) **Write a query** that will output all the ancestors (for us, these would be prereqs) of a given vertex.

Specific Approach: Hardcoding the vertex using V('CS526') command.

```
g.traversal().V('CS526').repeat(out().dedup()).emit()
```

OR

General approach: Using "has" command to determine the specific node given the id parameter/ property in the has clause.

```
g.traversal().V().has(id,'CS526').repeat(out().dedup()).emit()
```

Output:

```
v[CS400]
v[CS334]
v[CS201]
v[CS101]
```

Explanation:
For the entire graph, starting from the vertex with id as CS526 repeat the procedure of finding the adjacent vertices for the traversal and dedup will filter of all the duplicating vertices from the result. Finally, emit will emit the respective vertex at each iteration of the command repeat.

4) **Write a query** that will output the max depth starting from a given node (provides a count (including itself) of all the connected nodes till the deepest leaf). This would give us a total count of the longest sequence of courses that can be taken, after having completed a prereq course

Specific approach: Hardcoded root vertex

```
g.traversal().V('CS101').repeat(__.in()).emit(__.not(inE())).path().count(local).max()
```

OR

General approach: using has function, which retrieves the vertex or node based on the conditions specified in the has clause. Here the id is used as a parameter to get the respective node say CS101 in this case. This node can be changed to the specific node values.

```
g.traversal().V().has(id,'CS101').repeat(__.in()).emit(__.not(inE())).path().count(local).max()
```

Output:

```
5
```

Explanation:
Traversing through the graph starting from the vertex with id as CS101, repeatedly traversing (spawned anonymously using __) through all the adjacent vertex with emitting each vertex and also removing few vertex that does not return/emit any object (not() command); and the path command finds the history of the vertices those are traversed from the traversal __. The count command finds the total number of traversal in the streams of traversal; local is used to count the local object in the stream and finally the max command will return the max value from the stream of traversal.

Bonus :

g=TinkerGraph.open().traversal()

g.addV("CS101").property(id,"CS101").as("CS101").addV("CS201").property(id,"CS201").as("CS201").addV("CS220").property(id,"CS220").as("CS220").addV("CS420").property(id,"CS420").as("CS420").addV("CS334").property(id,"CS334").as("CS334").addV("CS681").property(id,"CS681").as("CS681").addV("CS400").property(id,"CS400").as("CS400").addV("CS526").property(id,"CS526").as("CS526").addE("prereq").from("CS201").to("CS101").addE("prereq").from("CS220").to("CS201").addE("prereq").from("CS420").to("CS220").addE("prereq").from("CS334").to("CS201").addE("prereq").from("CS681").to("CS334").addE("prereq").from("CS400").to("CS334").addE("prereq").from("CS526").to("CS400").addE("coreqs").from("CS420").to("CS220").addE("coreqs").from("CS526").to("CS400").iterate()

Explanation:
The above query for generating a graph using a 2 line of command as show above

- First command will create the tinker graph with empty vertices and edges and assign it to the graph variable g
- The second command will create a graph traversal with addV(), addE() and property() commands. Initially all the vertices are created using addV() command with label assigned to the vertex inside the parenthesis. The vertex is also assigned the property determined by key value pair with key being id and value being the course name. Finally the previous step are assigned name of their respective label using as command.
- After the vertices are created, edges are created between the vertices using addE command that assigned the label prereq between the vertices given by from and to commands. From command specifies from which vertex the edge start and to command where the other end pint of the vertex is at. Finally, iterate command is used to iterated through the traversal without outputting the the edges on the last command. Next() command can be used as an alternative to iterate() command, but it is not used in this query because the output should be just the creation of the graph rather than the display of each edges or last edge only.

2)
g.V().as('from').out('prereq').as('to').in('coreqs').select('from','to')

OR

g.V().as('a').out('prereq').as('b').in('coreqs').select('a','b')

Same as the above second query; but here since the graph generation query is generated such that ‘g’ is assigned the traversal variable, we can directly use g that traverse the graph rather than using g.traversal() command.

3)
g.V().has(id,'CS526').repeat(out().dedup()).emit()

Same as the above third query; but here since the graph generation query is generated such that ‘g’ is assigned the traversal variable, we can directly use g that traverse the graph rather than using g.traversal() command.

4)
g.V().has(id,'CS101').repeat(__.in()).emit(__.not(inE())).path().count(local).max()

Same as the above fourth query; but here since the graph generation query is generated such that ‘g’ is assigned the traversal variable, we can directly use g that traverse the graph rather than using g.traversal() command.