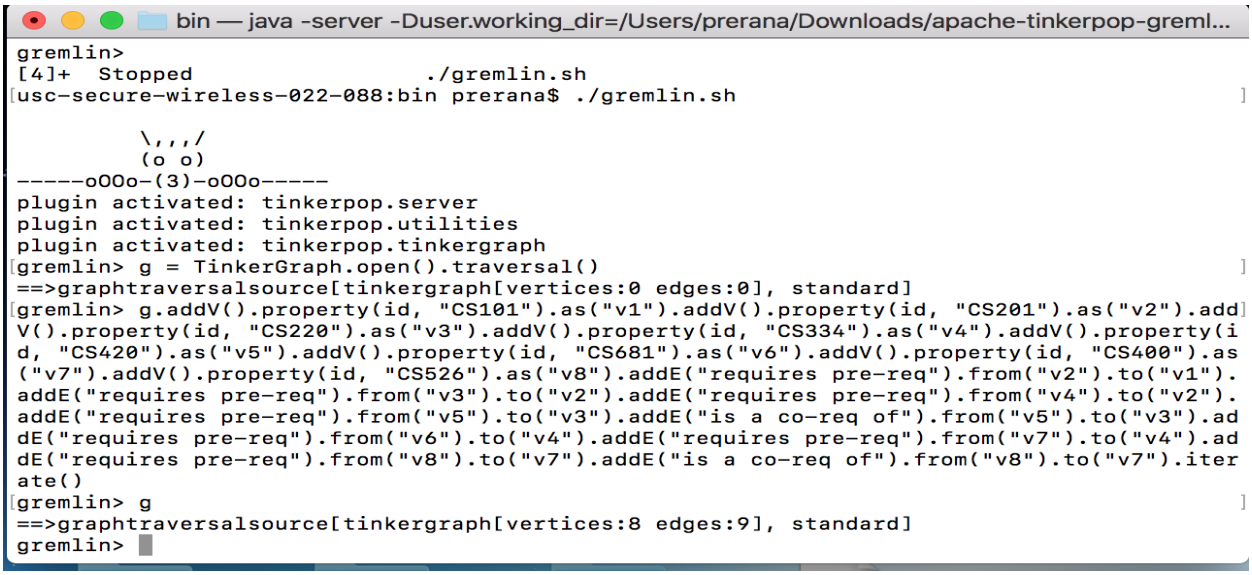
TinkerPop Gremlin:

Q1. **Write a Gremlin command** that creates the above graph [hint - you will also need a 'traversal' for it]. The command could be a multi-statement one, or a single line one (with function chaining).

- g = TinkerGraph.open().traversal()
g.addV().property(id, "CS101").as("v1").addV().property(id, "CS201").as("v2").addV().property(id, "CS220").as("v3").addV().property(id, "CS334").as("v4").addV().property(id, "CS420").as("v5").addV().property(id, "CS681").as("v6").addV().property(id, "CS400").as("v7").addV().property(id, "CS526").as("v8").addE("requires pre-req").from("v2").to("v1").addE("requires pre-req").from("v3").to("v2").addE("requires pre-req").from("v4").to("v2").addE("requires pre-req").from("v5").to("v3").addE("is a co-req of").from("v5").to("v3").addE("requires pre-req").from("v6").to("v4").addE("requires pre-req").from("v7").to("v4").addE("requires pre-req").from("v8").to("v7").addE("is a co-req of").from("v8").to("v7").iterate()

Explanation: Here I have used function chaining to create a graph g with 8 vertices and 9 edges. I am creating a graph instance and opening it, which is a reference to my graph data structure and a TraversalSource g which provides more info about my graph to gremlin using TinkerGraph.open().traversal() command. Next, I am adding vertices to g using addV() function, defining a property called id and calling those vertices as v1,v2,…v8 using as() function. Then I am adding edges using addE(), from() and to() functions. Edge name is passed as a parameter to addE(). Iterate() is nothing but Pipe.iterate method which calls Pipe.next or next() method for all objects in the pipe and next() gets the next object in the pipe or the next n objects.

Output:



Q2. **Write a query** that will output JUST the doubly-connected nodes.

- g.V().as("b").in("is a co-req of").as("a").select("a","b")

or

g.V().as("a").out("is a co-req of").as("b").select("a","b")

Explanation: We can use any one of the above queries to do this operation. Since doubly connected nodes have an extra edge labeled “is a co-req of”, we can just traverse the graph using in() or out() function looking for that extra edge labeled “is a co-req of” and outputting those vertices which have this edge. Out() gets the vertices which have an outgoing edge to them(where edge ends) and in() is for vertices from where an edge starts.

Output:

```
gremlin> g.V().as("b").in("is a co-req of").as("a").select("a","b")
==>[a:v[CS526],b:v[CS400]]
==>[a:v[CS420],b:v[CS220]]
gremlin> g.V().as("a").out("is a co-req of").as("b").select("a","b")
==>[a:v[CS420],b:v[CS220]]
==>[a:v[CS526],b:v[CS400]]
gremlin>
```

Q3. **Write a query** that will output all the ancestors (for us, these would be prereqs) of a given vertex.

- g.V().hasId('CS526').repeat(out().dedup()).emit()

or

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

or

g.V().hasId("CS526").as("ex").repeat(out()).until(hasId("CS101")).path().dedup().unfold().where(neq("ex"))

Explanation: We can use either of the queries above to do this. In first and second queries, I am just traversing the nodes which have an out edge entering them and avoiding duplicates using dedup() and emitting those nodes using emit() which gives all the ancestors. The terminating condition here is till the last node along this path is reached which is CS101 in every case. In third query (or part) I am traversing from the vertex/node CS526 and calling it ex using as(). Repeating out() traversal until I find CS101 which is the root node and printing the path using path(). dedup() is used to avoid duplicate nodes encountered along the path. Unfold() is used to print each node on a separate line in output. Where() and neq("ex") (which means not equal to), prints all the ancestors except the node CS526 itself because want only the ancestor nodes. All these queries give same answer.

Output:

```
gremlin> g.V().hasId('CS526').repeat(out().dedup()).emit()
==>v[CS400]
==>v[CS334]
==>v[CS201]
==>v[CS101]
gremlin> g.V('CS526').repeat(out().dedup()).emit()
==>v[CS400]
==>v[CS334]
==>v[CS201]
==>v[CS101]
gremlin> g.V().hasId("CS526").as("ex").repeat(out()).until(hasId("CS101")).path().dedup().
unfold().where(neq("ex"))
==>v[CS400]
==>v[CS334]
==>v[CS201]
==>v[CS101]
gremlin>
```

Q4. **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.

- g.V().hasId('CS101').emit().repeat(__.in()).path().dedup().tail().unfold().count()

Explanation: We start traversing from vertex CS101 and emitting all the paths till we reach leaf node by using emit().repeat (__.in()).path() and tail(n) function is used to emit the last n-objects and tail() emits only the last object from the output which in this case is the longest path from CS101. We remove duplicate nodes using dedup() and count all the vertices along this path using count() function and display it as output.

Output:

```
gremlin>  g.V().hasId('CS101').emit().repeat(__.in()).path().dedup().tail().unfold().count
()
==>5
gremlin>  g.V().hasId('CS526').emit().repeat(__.in()).path().dedup().tail().unfold().count
()
==>1
gremlin>
```