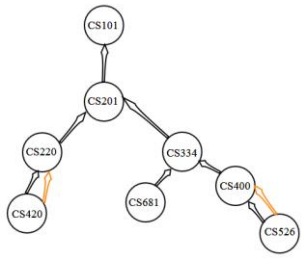
HOMEWORK-4

ANSWER-1

Here:
V(8) = CS526 V(7) = CS400 V(6) = CS681 V(5) = CS420
V(4) = CS334 V(3) = CS220 V(2) = CS201 V(1) = CS101



g=TinkerGraph.open().traversal()

g.addV("CS101").property(T.id,1).as("v1").addV("CS201").property(T.id,2).as("v2").addV("CS220").property(T.id,3).as("v3").addV("CS334").property(T.id,4).as("v4").addV("CS420").property(T.id,5).as("v5").addV("CS681").property(T.id,6).as("v6").addV("CS400").property(T.id,7).as("v7").addV("CS526").property(T.id,8).as("v8").addE("prerequisite").from("v2").to("v1").property(T.id,9).addE("prerequisite").from("v3").to("v2").property(T.id,10).addE("prerequisite").from("v4").to("v2").property(T.id,11).addE("prerequisite").from("v5").to("v3").property(T.id,12).addE("prerequisite").from("v6").to("v4").property(T.id,13).addE("prerequisite").from("v7").to("v4").property(T.id,14).addE("prerequisite").from("v8").to("v7").property(T.id,15).addE("corequisite").from("v5").to("v3").property(T.id,16).addE("corequisite").from("v8").to("v7").property(T.id,17).next()

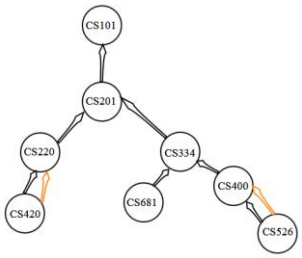
```
gremlin> g=TinkerGraph.open().traversal()
==>graphtraversalsource[tinkergraph[vertices:0 edges:0], standard]
gremlin> g.addV("CS101").property(T.id,1).as("v1").addV("CS201").property(T.id,2).as("v2").addV("CS220").property(T.id,3).as("v3").addV("CS334").property(T.id,4).as("v4").addV("CS420").property(T.id,5).as("v5").addV("CS681").property(T.id,6).as("v6").addV("CS400").property(T.id,7).as("v7").addV("CS526").property(T.id,8).as("v8").addE("prerequisite").from("v2").to("v1").property(T.id,9).addE("prerequisite").from("v3").to("v2").property(T.id,10).addE("prerequisite").from("v4").to("v2").property(T.id,11).addE("prerequisite").from("v5").to("v3").property(T.id,12).addE("prerequisite").from("v6").to("v4").property(T.id,13).addE("prerequisite").from("v7").to("v4").property(T.id,14).addE("prerequisite").from("v8").to("v7").property(T.id,15).addE("corequisite").from("v5").to("v3").property(T.id,16).addE("corequisite").from("v8").to("v7").property(T.id,17).next()
==>e[17][8-corequisite->7]
gremlin> g
==>graphtraversalsource[tinkergraph[vertices:8 edges:9], standard]
```

Explanation:

First I have opened a graph named "***g***" and made it ready for traversal. After this step, I added vertices and edges as per the graph shown in question.
For adding vertices, I added the name of node as ***addV("name")*** and a property named *id* for its unique identification. Using forward chaining, I added all the vertices and edges in one command. Then I added the edges giving its name as ***"prerequisite"*** or ***"corequisite"***, its path ***from("node a") to("node b")***, and its property ***id*** for unique identification.
Here ***as("xyz")*** is used for giving an alias named *xyz* for future reference of that particular edge or vertex.
To check if our graph is well formed, we can simply see its number of edges and vertices by typing our name of graph "***g***".

ANSWER-2

Here:
V(8) = CS526 V(7) = CS400 V(6) = CS681 V(5) = CS420
V(4) = CS334 V(3) = CS220 V(2) = CS201 V(1) = CS101



g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('prerequisite').inV().as("b").select("a","b")

OR

g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('corequisite').inV().as("b").select("a","b")

```
gremlin> g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('prerequisite').inV().as("b").select("a","b")
==>[a:v[5],b:v[3]]
==>[a:v[8],b:v[7]]
gremlin> g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('corequisite').inV().as("b").select("a","b")
==>[a:v[5],b:v[3]]
==>[a:v[8],b:v[7]]
```

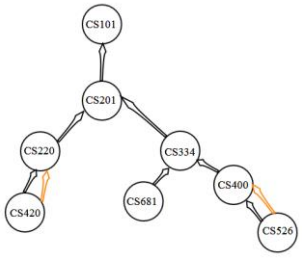
Explanation:

In this query we have to find the doubly-connected nodes.

Here I have given an alias name “a” to the vertices that fulfills the following where condition
Where condition : vertices from whom prerequisite as well as corequisite edges are coming out
Finally we find the vertex on the other end of the edge, whose one end satisfies the above where condition, i.e. this step is to have both the ends of the edges (prerequisite and corequisite)
To find the other end of the edges, we can use both of the following:
outE('prerequisite').inV()
OR
outE('corequisite').inV()
From a particular vertex, the other end vertex of its edges can be found using the outgoing edges of “prerequisite” or “corequisite” from that particular vertex.
This step is referred with its alias named “b”.
Select statement concatenated at the end is to print using both the aliases.

ANSWER-3

Here:
V(8) = CS526 V(7) = CS400 V(6) = CS681 V(5) = CS420
V(4) = CS334 V(3) = CS220 V(2) = CS201 V(1) = CS101



g.V(8).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")

Other hardcoded nodes:

g.V(7).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
g.V(6).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
g.V(5).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
g.V(4).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
g.V(3).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
g.V(2).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
g.V(1).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")

Since V(1)=CS101 has no path of prerequisites going out of it, no output will be displayed.

```
gremlin> g.V(8).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[7]
==>v[4]
==>v[2]
==>v[1]
gremlin> g.V(7).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[4]
==>v[2]
==>v[1]
gremlin> g.V(6).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[4]
==>v[2]
==>v[1]
gremlin> g.V(5).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[3]
==>v[2]
==>v[1]
gremlin> g.V(4).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[2]
==>v[1]
gremlin> g.V(3).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[2]
==>v[1]
gremlin> g.V(2).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[1]
gremlin> g.V(1).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
gremlin>
```

Explanation:

In this query we have to find the ancestors of a particular vertex.

Firstly I selected any vertex whose ancestors are to be found.
Then repeat (a looping step) the following step: from the selected vertex the outgoing edge "***prerequisite***" is selected, ***dedup()*** step is to avoid repetition of elements, ***inV()*** step is to get the vertex at the other end of edge (outgoing edge "prerequisite" of previously selected vertex). This vertex obtained at inV() is now selected for the next step in the loop and again the same steps are followed. Also this vertex is given an alias name "***a***" for future use.
The repeat loop is updated using ***emit()*** for forward traversal. Emit() determines if the current object in the loop is released or not. ***path()*** is to find the path i.e. the nodes that are visited during the entire loop fulfilling all the conditions presented in the till its termination condition. ***select*** is to print our final answer that is stored in the alias "a".

ANSWER-4

g.V(1).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)

Other hardcoded nodes:

g.V(2).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
g.V(3).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
g.V(4).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
g.V(5).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
g.V(6).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
g.V(7).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
g.V(8).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)

```
gremlin> g.V(1).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>5
gremlin> g.V(2).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>4
gremlin> g.V(3).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>2
gremlin> g.V(4).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>3
gremlin> g.V(5).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>1
gremlin> g.V(6).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>1
gremlin> g.V(7).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>2
gremlin> g.V(8).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>1
```

Explanation:

In this query we have to count maximum depth starting from a given node.

First select a node in g.V(*nodeX*). For reverse traversal, we use emit() first and then start our loop i.e. repeat().
In repeat(), first dedup() is used remove repetetion of objects having incoming edges as “prerequisite”. Path() finds all possible paths in ascending order.
Eg:

```
gremlin> g.V(1).emit().repeat(dedup().in('prerequisite')).path()
==>[v[1]]
==>[v[1],v[2]]
==>[v[1],v[2],v[3]]
==>[v[1],v[2],v[4]]
==>[v[1],v[2],v[3],v[5]]
==>[v[1],v[2],v[4],v[6]]
==>[v[1],v[2],v[4],v[7]]
==>[v[1],v[2],v[4],v[7],v[8]]
```

Tail() finds the last of which is the longest one:

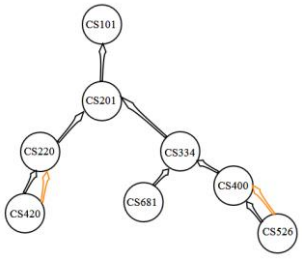
```
gremlin> g.V(1).emit().repeat(dedup().in('prerequisite')).path().tail()
==>[v[1],v[2],v[4],v[7],v[8]]
```

Finally count(local) is used to count the number of elements in out final answer, i.e. here it finds the number of elements in the tail() set of nodes (having maximum number of nodes).

BONUS QUESTION

ANSWER-1

Here:
V(8) = CS526 V(7) = CS400 V(6) = CS681 V(5) = CS420
V(4) = CS334 V(3) = CS220 V(2) = CS201 V(1) = CS101



g=TinkerGraph.open().traversal()

g.addV("CS101").property(T.id,1).as("v1").addV("CS201").property(T.id,2).as("v2").addV("CS220").property(T.id,3).as("v3").addV("CS334").property(T.id,4).as("v4").addV("CS420").property(T.id,5).as("v5").addV("CS681").property(T.id,6).as("v6").addV("CS400").property(T.id,7).as("v7").addV("CS526").property(T.id,8).as("v8").addE("prerequisite").from("v2").to("v1").property(T.id,9).addE("prerequisite").from("v3").to("v2").property(T.id,10).addE("prerequisite").from("v4").to("v2").property(T.id,11).addE("prerequisite").from("v5").to("v3").property(T.id,12).addE("prerequisite").from("v6").to("v4").property(T.id,13).addE("prerequisite").from("v7").to("v4").property(T.id,14).addE("prerequisite").from("v8").to("v7").property(T.id,15).addE("corequisite").from("v5").to("v3").property(T.id,16).addE("corequisite").from("v8").to("v7").property(T.id,17).next()

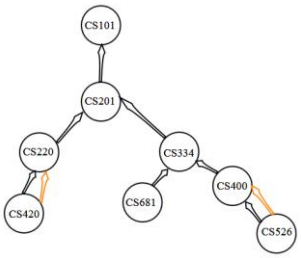
```
gremlin> g=TinkerGraph.open().traversal()
==>graphtraversalsource[tinkergraph[vertices:0 edges:0], standard]
gremlin> g.addV("CS101").property(T.id,1).as("v1").addV("CS201").property(T.id,2).as("v2").addV("CS220").property(T.id,3).as("v3").addV("CS334").property(T.id,4).as("v4").addV("CS420").property(T.id,5).as
("v5").addV("CS681").property(T.id,6).as("v6").addV("CS400").property(T.id,7).as("v7").addV("CS526").property(T.id,8).as("v8").addE("prerequisite").from("v2").to("v1").property(T.id,9).addE("prerequisite
").from("v3").to("v2").property(T.id,10).addE("prerequisite").from("v4").to("v2").property(T.id,11).addE("prerequisite").from("v5").to("v3").property(T.id,12).addE("prerequisite").from("v6").to("v4").prope
rty(T.id,13).addE("prerequisite").from("v7").to("v4").property(T.id,14).addE("prerequisite").from("v8").to("v7").property(T.id,15).addE("corequisite").from("v5").to("v3").property(T.id,16).addE("corequisi
te").from("v8").to("v7").property(T.id,17).next()
==>e[17][8-corequisite->7]
gremlin> g
==>graphtraversalsource[tinkergraph[vertices:8 edges:9], standard]
```

Explanation:

First I have opened a graph named "***g***" and made it ready for traversal. After this step, I added vertices and edges as per the graph shown in question.
For adding vertices, I added the name of node as ***addV("name")*** and a property named *id* for its unique identification. Using forward chaining, I added all the vertices and edges in one command.
Then I added the edges giving its name as ***"prerequisite"*** or ***"corequisite"***, its path ***from("node a") to("node b")***, and its property ***id*** for unique identification.
Here ***as("xyz")*** is used for giving an alias named *xyz* for future reference of that particular edge or vertex.
To check if our graph is well formed, we can simply see its number of edges and vertices by typing our name of graph "***g***".

ANSWER-2

Here:
V(8) = CS526 V(7) = CS400 V(6) = CS681 V(5) = CS420
V(4) = CS334 V(3) = CS220 V(2) = CS201 V(1) = CS101



g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('prerequisite').inV().as("b").select("a","b")

OR

g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('corequisite').inV().as("b").select("a","b")

```
gremlin> g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('prerequisite').inV().as("b").select("a","b")
==>[a:v[5],b:v[3]]
==>[a:v[8],b:v[7]]
gremlin> g.V().as("a").where(out('prerequisite').and().out('corequisite')).outE('corequisite').inV().as("b").select("a","b")
==>[a:v[5],b:v[3]]
==>[a:v[8],b:v[7]]
```

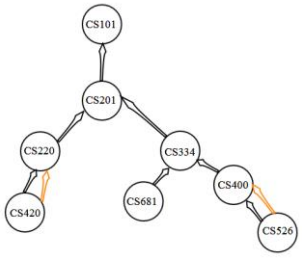
Explanation:

In this query we have to find the doubly-connected nodes.

Here I have given an alias name “a” to the vertices that fulfills the following where condition
Where condition : vertices from whom prerequisite as well as corequisite edges are coming out
Finally we find the vertex on the other end of the edge, whose one end satisfies the above where condition, i.e. this step is to have both the ends of the edges (prerequisite and corequisite)
To find the other end of the edges, we can use both of the following:
outE('prerequisite').inV()
OR
outE('corequisite').inV()
From a particular vertex, the other end vertex of its edges can be found using the outgoing edges of “prerequisite” or “corequisite” from that particular vertex.
This step is referred with its alias named “b”.
Select statement concatenated at the end is to print using both the aliases.

ANSWER-3

Here:
V(8) = CS526 V(7) = CS400 V(6) = CS681 V(5) = CS420
V(4) = CS334 V(3) = CS220 V(2) = CS201 V(1) = CS101



g.V(8).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")

Since V(1)=CS101 has no path of prerequisites going out of it, no output will be displayed.

```
gremlin> g.V(8).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[7]
==>v[4]
==>v[2]
==>v[1]
gremlin> g.V(7).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[4]
==>v[2]
==>v[1]
gremlin> g.V(6).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[4]
==>v[2]
==>v[1]
gremlin> g.V(5).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[3]
==>v[2]
==>v[1]
gremlin> g.V(4).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[2]
==>v[1]
gremlin> g.V(3).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[2]
==>v[1]
gremlin> g.V(2).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
==>v[1]
gremlin> g.V(1).repeat(outE('prerequisite').dedup().inV().as("a")).emit().path().select("a")
gremlin>
```

Explanation:

In this query we have to find the ancestors of a particular vertex.

Firstly I selected any vertex whose ancestors are to be found.
Then repeat (a looping step) the following step: from the selected vertex the outgoing edge "***prerequisite***" is selected, ***dedup()*** step is to avoid repetition of elements, ***inV()*** step is to get the vertex at the other end of edge (outgoing edge "prerequisite" of previously selected vertex). This vertex obtained at inV() is now selected for the next step in the loop and again the same steps are followed. Also this vertex is given an alias name "***a***" for future use.
The repeat loop is updated using ***emit()*** for forward traversal. Emit() determines if the current object in the loop is released or not. ***path()*** is to find the path i.e. the nodes that are visited during the entire loop fulfilling all the conditions presented in the till its termination condition. ***select*** is to print our final answer that is stored in the alias "a".

ANSWER-4

g.V(1).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)

```
gremlin> g.V(1).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>5
gremlin> g.V(2).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>4
gremlin> g.V(3).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>2
gremlin> g.V(4).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>3
gremlin> g.V(5).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>1
gremlin> g.V(6).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>1
gremlin> g.V(7).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>2
gremlin> g.V(8).emit().repeat(dedup().in('prerequisite')).path().tail().count(local)
==>1
```

Explanation:

In this query we have to count maximum depth starting from a given node.

First select a node in g.V(*nodeX*). For reverse traversal, we use emit() first and then start our loop i.e. repeat().
In repeat(), first dedup() is used remove repetetion of objects having incoming edges as "prerequisite". Path() finds all possible paths in ascending order.
Eg:

```
gremlin> g.V(1).emit().repeat(dedup().in('prerequisite')).path()
==>[v[1]]
==>[v[1],v[2]]
==>[v[1],v[2],v[3]]
==>[v[1],v[2],v[4]]
==>[v[1],v[2],v[3],v[5]]
==>[v[1],v[2],v[4],v[6]]
==>[v[1],v[2],v[4],v[7]]
==>[v[1],v[2],v[4],v[7],v[8]]
```

Tail() finds the last of which is the longest one:

```
gremlin> g.V(1).emit().repeat(dedup().in('prerequisite')).path().tail()
==>[v[1],v[2],v[4],v[7],v[8]]
```

Finally count(local) is used to count the number of elements in out final answer, i.e. here it finds the number of elements in the tail() set of nodes (having maximum number of nodes).