Project 6

Due Date: Tuesday 6 December by 11:59 PM

General Guidelines.

The methods required are described below. Make sure your signatures match or they won't work with the autograder. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment, and you are welcome to include those in your submission.

Your solution must be coded in Java.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!
Importing the Math class is okay.

This project allows more classes to be used than previous projects, so make sure you read the instructions carefully. In particular, you are allowed to use the ArrayList and Hashmap classes. Make sure you use them efficiently! If there are other java classes you would like to use, just ask, making sure to give a brief explanation of how you plan on using it!

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will be subject to consequences according to the syllabus and university policy.

Note on grading and provided tests: There is no tester provided, but sample "graph" files are provided that you can use for your own testing. The graphs we use for grading will be similar to these, and these are small enough that you could draw them out and verify your results manually.

Project Overview.

This project is all about graphs. You will be required to implement a Graph class in a file called `Graph.java` that can handle both directed and undirected graphs and can solve various graph-related problems. Very likely, you will want to include multiple representations of the graphs so that you can easily solve each of the problems, and please be mindful of how your use structures so that your implementations are not too inefficient.

Note: If it is easier for you, you can create separate classes for an undirected graph and a directed graph, but make sure the `Graph.java` handles both and serves as the actual interface between the tester and your code. If you do decide to have separate classes, you will find that a lot of code will be repeated, which is why I decided to have them together.

API: Required Methods for Graph.java

Signature	**Description**	**Expected Runtime**
Graph(int *n*, boolean *directed*)	constructor; *n* = the number of vertices; *directed* = true if the graph is directed, false if it is undirected	
Graph(String *fn*)	constructor; *fn* = the filename containing the graph information (see below for a breakdown of the files); NOTE: Do not hard-code any path names or filenames or your code may not work with the autograder!	
int V()	return the number of vertices	O(1)
int E()	return the number of edges	O(1)
void addEdge(int *v*, int *w*)	add an edge from *v* to *w* unless (*v*, *w*) already exists in the graph	O(1)

boolean adjTo(int *v*, int *w*)	return true iff *v* is adjacent to *w*	O(1)
boolean isSimple()	return true iff the graph is simple	O(V)
HashMap<ArrayList<Integer>> components()	return a Hashmap containing lists that indicate the connected components (for an undirected graph) or strongly connected components (for a directed graph); the key for each component should be the smallest vertex in that component	O(V + E)
int numComponents()	return the number of connected (or strongly connected for digraphs) components in the graph	O(V + E)
boolean isBiconnected()	return true iff the graph (only undirected) is biconnected	O(V(V+E))
ArrayList<Integer> articulationVertices()	return a list of the articulation vertices in the graph (only undirected graphs)	O(V(V+E))
boolean isAcyclic()	return true iff the graph is acyclic	O(V+E)
ArrayList<Integer> path(int v, int w)	return a list of vertices that make up a path from *v* to *w*; return null if such a path does not exist	O(V+E)
ArrayList<Integer> cycle(int v)	return a list of vertices that make up a cycle starting and ending at *v*; return null if such a cycle does not exist; a self-loop may count as a cycle	O(V+E)
Graph spanningTree()	return a graph that is a spanning tree of this	O(V+E)

	graph (undirected graphs only)	
boolean isConnected()	return true if the graph is connected (for undirected graphs)	O(V+E)
boolean isStronglyConnected()	return true if the graph is strongly connected (for directed graphs)	O(V+E)
ArrayList<Integer> topSort()	return a list of vertices that is a topological ordering of the (directed) graph; return null if such an ordering does not exist	O(V+E)
boolean isArticulation(int v)	return true iff *v* is an articulation vertex (undirected graphs only)	O(V+E)
boolean isBridge(int v, int w)	return true iff (v, w) is a bridge (which is an edge that is like an articulation vertex–if it is removed, the graph becomes disconnected	O(V+E)
ArrayList<Integer> dfsOrder(int v)	return a list of the vertices in the order they are discovered when doing dfs starting at vertex *v*	O(V+E)
ArrayList<Integer> bfsOrder(int v)	return a list of the vertices in the order they are discovered when doing bfs starting at vertex *v*	O(V+E)
boolean equals(Graph G)	return true if the graphs are the same–meaning that they have the same vertices and edges, but the order of the edges in the representation does not matter	O(V+E)
Graph transitiveClosure()	return a graph that is the transitive closure	$O(V^3)$

	of this graph (directed graphs only)	
int degree(int v)	return the degree of the vertex (for undirected graphs)	O(1)
int indegree(int v)	return the indegree of the vertex (for directed graphs)	O(1)
int outdegree(int v)	return the outdegree of the vertex (for directed graphs)	O(1)

Examples.

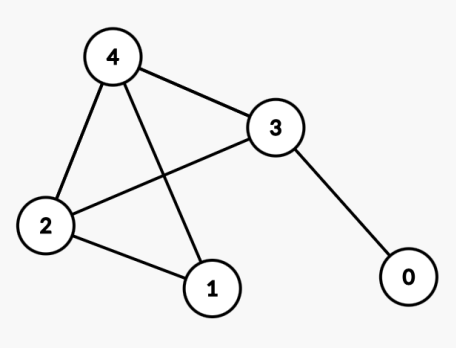
1.

graph1.txt

```
u 5
1 2
3 2
2 4
0 3
3 4
2 4
4 3
1 4
```

Explanation of the file. The first line indicates that this graph is undirected and has 5 vertices. Each subsequent line is an edge to be added. Note that you should not allow duplicate edges.

The graph drawn out.



Examples of functions that could be tested.

Call	Result	Notes
V()	5	
E()	6	
adjTo(4, 1)	true	
adjTo(3, 1)	false	
isSimple()	true	
components()	(0, {0, 1, 2, 3, 4})	0 is the key, the list containing the vertices is the value; the order of the list does not matter

numComponents()	1	
isBiconnected()	false	
articulationVertices()	{3}	the order of the list does not matter
isAcyclic()	false	
path(1, 3)	{1, 2, 3}	there may be other paths–only one needs to be returned
cycle(1)	{1, 2, 4, 1}	there may be other cycles–only one needs to be returned
cycle(0)	null	
spanningTree()	0: 3 1: 2 2: 1, 4 3: 0, 4 4: 2, 3	there may be other spanning trees–only one needs to be returned
isConnected()	true	
isArticulation(3)	true	
isArticulation(4)	false	
isBridge(1, 4)	false	
isBridge(0, 3)	true	
dfsOrder(0)	{0, 3, 2, 1, 4}	order will depend on the

		adjacency list order
bfsOrder(0)	{0, 3, 2, 4, 1}	order will depend on the adjacency list order
degree(0)	1	

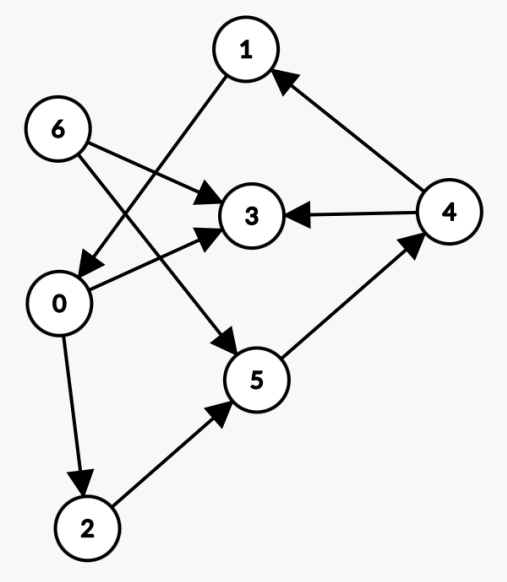
2.

graph7.txt

```
d 7
0 2
0 3
1 0
2 5
4 1
4 3
5 4
6 3
6 5
```

Explanation of the file. The first line indicates that this graph is directed and has 7 vertices. Each subsequent line is an edge to be added. Note that you should not allow duplicate edges.

The graph drawn out.



Examples of functions that could be tested.

Call	Result	Notes
V()	7	
E()	9	
adjTo(4, 1)	true	
adjTo(1, 4)	false	
isSimple()	true	
components()	(0, {0, 2, 5, 4, 1}), (3, {3}),	these are the key-value pairs

	(6, {6})	in the HashMap; the order of the lists does not matter
numComponents()	3	
isAcyclic()	false	
path(1, 3)	{1, 0, 3}	there may be other paths–only one needs to be returned
path(3, 1)	null	
cycle(1)	{1, 0, 2, 5, 4, 1}	there may be other cycles–only one needs to be returned
cycle(6)	null	
isStronglyConnected()	false	
dfsOrder(0)	{0, 2, 5, 4, 1, 3}	order will depend on the adjacency list order; note that 6 is not included because 6 is not reachable from 0
bfsOrder(0)	{0, 2, 3, 5, 4, 1}	order will depend on the adjacency list order; note that 6 is not included because 6 is not reachable from 0
indegree(0)	1	
outdegree(0)	2	
topSort()	null	

`transitiveClosure()`	`0: 0, 1, 2, 3, 4, 5`
	`1: 0, 1, 2, 3, 4, 5`
	`2: 0, 1, 2, 3, 4, 5`
	`3:`
	`4: 0, 1, 2, 3, 4, 5`
	`5: 0, 1, 2, 3, 4, 5`
	`6: 0, 1, 2, 3, 4, 5`	

The graphs above were drawn using https://csacademy.com/app/graph_editor/, which may be a useful tool for you as you use the graphs provided for testing.

Testing your code on lectura.

Once you have transferred your files,

- you can compile a java file in the terminal with the following command:
  `javac <filename>`
- you can can compile all the .java files with this command:
  `javac *.java`
- you can run a compiled file with the following command:
  `java <filename>`

Submission Procedure.

Note: These instructions assume that all your code is in one file. If you have multiple files, you may submit those as well. Do not submit any code that is only for testing. Only submit code that is required for your solution.
To submit your code, please upload the files to **lectura** and use the **turnin** command below. Once you log in to lectura and transfer your files, you can submit using the following command:

```
turnin cs345p6 Graph.java
```

Upon successful submission, you will see this message:

```
Turning in:
    Graph.java - ok
All done.
```

Note: Your code submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it runs and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues.

Grading.

Tests are not provided for this project. Instead, you are given some example graph files to work with, and it is up to you to test your code. Rest assured, however, that the graphs we use will be very similar to those provided. Also, the functions will be tested separately so that you can get some credit even if you are unable to complete every function–**as long as your code compiles and does not crash!**

Possible Deductions

Item	**Max Deduction Possible**
Bad Coding Style	5 points
Not following instructions	100% of the points you would have earned (i.e. automatic 0)
Not submitted correctly on lectura	100% of the points you would have earned (i.e. automatic 0)
Does not compile/run on lectura	100% of the points you would have earned (i.e. automatic 0)

Other notes about grading:

- If you do not follow the directions (e.g. importing classes without permission, etc.), you may not receive credit for that part of the assignment.
- Good Coding Style includes: using good indentation, using meaningful variable names, using informative comments, breaking out reusable functions instead of repeating them, etc.
- If you implement something correctly but in an inefficient way, you may not receive full credit.
- In cases where efficiency is determined by counting something like array accesses, it is considered academic dishonesty to *intentionally* try to avoid getting more access counts without actually improving the efficiency of the solution, so if you are in doubt, it is always best to ask. If you are asking, then we tend to assume that you are trying to do the project correctly.
- If you have questions about your graded project, you may contact the TAs and set up a meeting to discuss your grade with them in person. Regrades on programs that do not work will only be allowed under limited circumstances, usually with a standard 15-point deduction. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.