Assignment 4

This fourth assignment focuses on implementation and comparison of the various sorting techniques considered so far.

Updating your repository

Most of the hard work was done in the last lab. All you need to do is fetch and merge the changes I made to the upstream repository.

Before we start anything, make sure that you have NO uncommitted changes. You must either commit what you have, if it is ready to commit, or stash the changes, allowing you to retrieve them later.

1. First make sure that your local master is the active branch. It should have a little checkbox next to it.
2. We should make sure to bring in the newest version of the upstream remote. Right-click on the "upstream" remote in the "Remotes" section on the left, and choose "fetch upstream".
3. Now right-click on the upstream/master branch within the "Remotes" section to bring up its context menu. One of the options will say "Merge upstream/master onto master". Click it.
4. You should now be ready to go. Whenever you push your changes to your origin, these commits from the upstream will go along with them. If you like, you can go ahead and push now.

The assignment

Introductory stuff

TODO

The main file you will work on in this assignment is TopologicalSort.java, inside src/main/java/utils.

As in the previous assignment, you will find numerous TODO comments throughout the file. They mark the locations where you need to add something to the implementations. Some times it will be one or two lines, some times it may be more. If you use Eclipse, you can see blue rectangles on the right side of the code indicating the location of all TODOs, and you can click on them to go there. You can keep track of your progress that way, so make sure to remove the TODO from the methods you have already implemented.

A standard printout of a run of the various methods can be obtained by running the main method of the TopologicalSort class (so running the TopologicalSort class as an application). To do that, you will need to first build using the corresponding gradle task, and then run the runTopological gradle task from the run group of tasks (you may need to refresh the gradle project and possibly close and reopen the gradle tasks to make it appear).

Alternatively, you can go to your terminal (or Eclipse's console), go to the main project folder and run:

```
java -cp build/libs/algorithms-assignments.jar utils.TopologicalSort
```

If you like to write more tests, you can create test files under /src/test/java. Ask for help if you are not sure how to start those tests. You can use the :test Gradle task to run the tests.

You should make a new *commit* in GitKraken after completing each self-contained part, typically after each method you complete. Do not try to commit everything at once.

When you are done with the assignment, in order to "submit" it, simply **push** your commits via GitKraken. These will now be posted on your account in GitLab and I will be able to follow them from there.

The structure of the files

There are one file of concern in this assignment. TopologicalSort.java.

The TopologicalSort class contains a SimpleGraph instance along with its order. These are set via the constructor. The only other part is a run method, which is meant to execute a source-removal topological sort and return a List of the vertices in that order.

Here is the main workflow of the run method:

1. We keep a resultList that will contain the list of vertices in sorted order. This is what we return at the end of our method.
2. A processQueue queue contains vertices that are source vertices (no incoming edges). These vertices are the next to go in the result list.
3. An edgeCount array keeps track of the number of incoming edges for each vertex. Vertices with count 0 are source vertices.
4. We effectively process the vertices in the processQueue. For each such source vertex, we "remove" it from the graph. We don't actually remove the vertex, we simply adjust the edgeCount array to account for the fact that the edges outgoing from this vertex are removed. So if the vertex 3 is the next one to be removed, and it has edges going to vertices 5 and 6, then we adjust the edgeCount values of 5 and 6 to be one less.
5. When a vertex reaches an edgeCount of 0, it is a new source vertex of the smaller graph and we therefore add it to processQueue.
6. We repeat this process until processQueue is empty

The actual tasks

Now on to the tasks you need to complete. Make sure you have read the previous section for an understanding of the structure of the files.

All your work is in the body of the run method.

1. You first need to initialize the resultList, processQueue and edgeCount local variables. The list and queue start empty, and the edgeCount array will start with values 0, which Java does for us when we call the array constructor.
2. You then need to start the edgeCount array with the correct values. edgeCount[j] must end up containing the number of edges that are incoming to j. You have to simply go through the vertices of the graph, and for each vertex go through the outgoing edges. For each such edge, you must increment the edgeCount of the target vertex.
3. Next we need to search for any available source vertices, so we can start the process. These vertices are those with edgeCount 0. You simply go through the edgeCount array, and any vertices that have a count of 0 you add to the processQueue.
4. If no vertices were found that way, this means that graph is not a directed acyclic graph and therefore cannot be topologically-sorted. You can simply throw an error.
5. Next we have a main loop that goes on as long as the processQueue is not empty (you have to write the while loop). It consists of the following parts:
    a. First we remove the next available vertex from the queue (the while loop body should only execute if the processQueue is not empty.
    b. This vertex is the next source vertex, so we need to add it to our resultList.
    c. This vertex is to be removed from the graph. We therefore go through the outgoing edges from this vertex, and decrement the corresponding edgeCount entries.
    d. If one of those entries was already at 0 when we tried to decrement it, we have a logic error, as this should only happen if we miscounted the edges earlier.
    e. If the entry became 0 after we decremented it, then we just discovered a new source vertex, and we add it to the queue.

And we're done!