HW 6: Topological Sorting

I have written a program called `TaskSorter` that reads a `.task` file and sorts the tasks into the order the must be performed in. To support this you must create both a `Digraph` object and a `TopologicalSorter` non-instantiable class.

The `.task` file contains n lines, which is the number of tasks to do. Each line is a tab-separated list, where the first token is some new task, and all remaining tokens are the other tasks that must be done prior to it. For example, this line defines task `A`, but indicates that tasks `B` and `C` must be done first:

```
A    B    C
```

Every listed task will be the first task on exactly one line, and no task is directly dependent on itself. Tasks in the file are listed in arbitrary order.

When no tasks are dependent on each other, the program outputs a numbered list which sorts the tasks into a valid order. For example, when running the program on the provided file `acyclic.task`, one possible output is:

```
$ java TaskSorter acyclic.task
The file "acyclic.task" contains 4 tasks, with no cycles.  You must:
 1. Impignorate the bibbles
 2. Xertz the zoanthropes
 3. Quire the ratoon
 4. Snollygoster the gubbin
```

However, some `.task` files contain mutual dependencies: cycles. If several tasks are dependent on each other (either directly or indirectly), then they must be done at the same time. In this case, some numbered items will be comma-separated lists. For example:

```
$ java TaskSorter cyclic.task
The file "cyclic.task" contains 6 tasks, some of which are mutually
dependent. You must:
 1. Schnock the gardyloo
 2. Dumfoozle the bumfuzzle, Absquatulate the gorns, Frick the fracks
 3. Wobble the horta, Borborate the taradiddle
```

Your `Digraph` object should implement the following API:

- `Digraph()` (constructor) (O(1)). Creates an empty digraph.
- `void addVertex(String name)` (amortized O(1)) Inserts a vertex into the digraph. Throws an `IllegalArgumentException` if a vertex by that name already exists.
- `void addEdge(String vertex1, String vertex2)` (amortized O(1)) Adds a new one-way edge from the first vertex to the second. It should throw an `IllegalArgument`

`Exception` if the vertices don't exist.

- `boolean deleteEdge(String vertex1, String vertex2)` (amortized O(1)) Removes an edge from the first vertex to the second. (It will not affect an edge in the opposite direction.) Returns `true` if successful, or `false` if there was no edge. Throws an `IllegalArgumentException` if the vertices don't exist.
- `int size()` (O(1)) Return the number of vertices in the digraph.
- `boolean isValidVertex(String vertex)` (O(1)) Returns `true` if the vertex exists, or `false` otherwise.
- `String[] getAdjacencyList(String vertex)` (O(d) where d is the vertex's outward degree) Gets a list of all other vertices that this vertex has an edge to. It may return an array of length 0 if the vertex has no outward edges.
- `String[] getVertices()` (O(n)) Returns an array of all the vertices.
- `Digraph makeReverseGraph()` (O(n)) Returns a new `Digraph` with the same vertices but edges in the opposite directions.

Depending on your implementation, you may find it useful to make other public functions to communicate with your `TopologicalSorter`. Methods with linear running times may assume that a vertex's degree is proportional to n.

Your `TopologicalSorter` non-instantiable class should have these public functions:

- `boolean isDirectedCycle(Digraph digraph)` (O(n)) Returns `true` if there is at least one cycle in the digraph.
- `String[] sortTopologically(Digraph digraph)` (O(n)) Returns a sorted array of all the vertices, such that each vertex is listed before all others to which it has an edge.
- `String[][] findStrongComponents(Digraph digraph)` (O(n)) Returns a 2D sorted array of all the vertices. Every element of the array is an array of vertices in one strongly-connected component. The ordering within each component is arbitrary, but the components must be in a valid topological ordering for the kernel DAG.

Again, you may assume that on average each vertex's degree is proportional to n.

The usual caveats hold: coding style and comments are important, and if you aren't using Java you should change the API to match your language of choice.