COP 3503 - Programming Assignment #3
Topological Sorting
Assigned: Feb 11, 2014 (Tuesday)
Due: Feb 25, 2014 (Tuesday) at 11:55 PM WebCourses time

Objective
Implement topological sorting using depth-first search and decrease-by-one and conquer techniques.

Problem: Topological Sorting
Given a directed graph, find an ordering of its vertices such that *u* can only appear in the ordering after all vertices *v* have already appeared such that (*v*,*u*) is an edge in the graph. The graph must contain no cycles to have a topological sort, i.e., it must be a directed acyclic graph (DAG).

Algorithm 1 – Topological Sorting by Depth-first search

Implement the topological sorting algorithm described in section 4.2 of the textbook using depth-first search. See the section "TS(0, DFS) Algorithm Walk-through" for the steps your algorithm should follow for the first graph of the example input.

Algorithm 2 – Topological Sorting by decrease-by-one and conquer

Implement the topological sorting algorithm described in the section "Decrease-by-one and conquer topological sorting algorithm." This strategy is discussed in section 4.2 of the textbook.

Input and Output

Input

Input is from **graphs.txt**. The first line is the number of graphs, *k*. The rest of the file contains the *k* graphs. The first line of a graph is its number of vertices, *n*. The next *n* lines, each containing *n* integers, specify its adjacency matrix.

Output

Output is to **System.out**. Each graph *j* in the input (where graphs are labeled *0* through *k-1*) produces two lines of output in one of the two following forms. The first form is used if the graph does not have a topological sort:

TS(j, DFS): NO TOPOLOGICAL SORT
TS(i, DBO): NO TOPOLOGICAL SORT

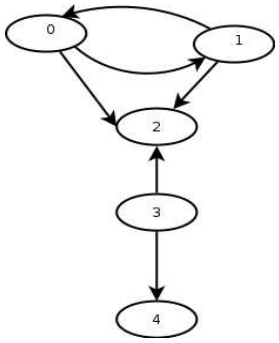
The second form is used if the graph does have a topological sort:

TS(j, DFS): *topological ordering of vertices in the graph as produced by the DFS algorithm*
TS(i, DBO): *topological ordering of vertices in the graph as produced by the DBO algorithm*

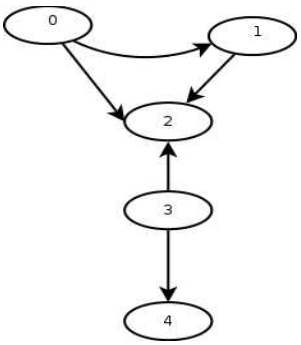
Sample input

```
2
5
0 1 1 0 0
1 0 1 0 0
0 0 0 0 0
0 0 1 0 1
0 0 0 0 0
5
0 1 1 0 0
0 0 1 0 0
0 0 0 0 0
0 0 1 0 1
0 0 0 0 0
```

Graph 0



Graph 1



Sample output

```
TS(0, DFS): NO TOPOLOGICAL SORT
TS(0, DBO): NO TOPOLOGICAL SORT
TS(1, DFS): 3 4 0 1 2
TS(1, DBO): 0 3 1 4 2
```

Restrictions on Source Code

- Submit a file named **TopoSort.java** which defines a *public class TopoSort* which defines a *public static void main(String[])* method. This method Reads the input file "graphs.txt" and produces the desired output to System.out. You may submit other Java files as well.
- Do not put any of your your Java files in a package. Omit the package statement to leave them in the default package.
- Your program must compile using Java 7.0 or later. It's okay to develop your program using the IDE of your choice, although Eclipse is recommended. Your program should include a header comment with the following information: your name, course number, section number, assignment title, and date.

Deliverables

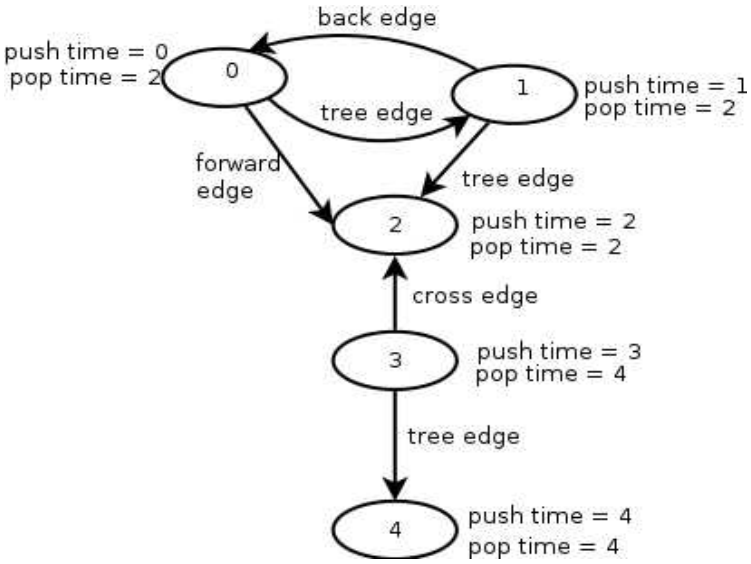
You must submit TopSort.java and any additional Java files to WebCourses by 11:55 PM on Sunday, February 25, 2014. You must send your source files as an attachment using the "Add Attachments" button. Assignments that are typed into the submission box will not be accepted. Assignments that are 1 day late are deducted 25% of the points received. Assignments more than 1 day late are not accepted. Programs that do not compile will receive no credit.

TS(0, DFS) Algorithm Walk-through

DFS uses **time** counter initialized to 0.

Every time a new vertex is encountered (and consequently pushed) it is assigned a **push time ← time**. After pushing, **time** is incremented. When a vertex is popped, it is assigned a **pop time ← time**.

If we are at vertex u and we push vertex v, then (u,v) is a tree edge. Note that in order to push v, v must not already have a push time (i.e., v must have never entered the stack before).



Push 0 (time = 0)
Push 1 (time = 1)

- Check (1,0). vertex 0 is still on the stack (no pop time), so (1,0) is a back edge. This edge creates a cycle. The algorithm may stop here since no topological sort is possible, but it will continue for demonstration purposes.

Push 2 (time = 2)
Pop 2 (time = 2)
Pop 1 (time = 2)

- Check (0,2). Pop time of vertex 2 > push time of vertex 0, so (0, 2) is a forward edge.

Pop 0 (time = 2)

Push 3 (time = 3)

- Check (3,2). Pop time of vertex 2 < Push time of vertex 3, so (3, 2) is a cross edge.

Push 4 (time = 4)
Pop 4 (time = 4)
Pop 3 (time = 4)

This graph does not have a topological sort because it has a back edge. However, if it had no back edge, then the topological sort would be the reverse order in which vertices were popped, i.e., 3, 4, 0, 1, 2.

For this assignment, it is not necessary to explicitly identify cross edges or forward edges, only back edges. Another way to identify a back edge (u,v) is to check if v is currently on the stack. It is not necessary to implement the time system as shown here, but the pushing and popping order of your algorithm must be the same as shown here.

Decrease-by-one and conquer topological sorting algorithm

Let Q be a queue.

For each vertex (in increasing order), add it to Q if it has no incoming edges.

Let T be an empty array that will contain the vertices of the topological ordering.

While Q is not empty:

v ← Dequeue vertex from Q.

Add V to the topological ordering T.

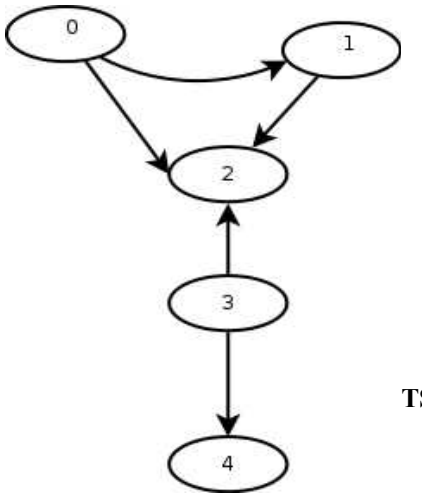
Delete the outgoing edges of v.

Enqueue any adjacent vertices of v (in increasing order) without incoming edges.

If every vertex of the graph was not dequeued, then stop, since the graph contains a cycle.

Otherwise, print the topological ordering T.

(It is not necessary to explicitly delete edges from the graph. An alternative is to keep a count of incoming edges associated with each vertex and decrement this counter whenever an incoming edge is "deleted".)



TS(1, DBO): 0 3 1 4 2