I collaborated with: W3029343

Problem
In class we outlined a *topological sorting* algorithm that would produce, for any DAG $G = (V, E)$, a total ordering of the vertices that was consistent with all of the edge directions; that is, an ordering $v_1, \ldots, v_n$ such that every edge $e = \{v_i, v_j\}$ satisfies $i < j$. This question asks you to consider some of the more subtle aspects of that algorithm. So, before trying to solve this problem, please read Section 3.6 of your text carefully. Go on. Do it now. I'll wait....
Welcome back. Recall that the algorithm, essentially, works as described in the pseudo-code on the next page. What makes it efficient is how Steps 3 and 5 are implemented (and integrated with one another). Done properly, the algorithm requires only $O(n + m)$ space and time, in part because each vertex and edge are inspected only a constant number of times.

Algorithm 1 Topological Sort

Require: A DAG $G = (V, E)$
Ensure: An ordered list $L = v_1, \ldots, v_n$ such that every edge $e = \{v_i, v_j\}$ satisfies $i < j$.

1: Create an empty list L
2: **while** $V \neq \emptyset$ **do**
3: Select a $v \in V$ with $indeg(v) = 0$
4: Add v to the end of L
5: Delete v from V and all edges incident with (containing) v from E
6: **end while**
7: **return** L

a) Is it necessary to delete v and its edges from G? Precisely, suppose we have sufficient space to store G along with only an additional $O(n)$ of space available (n is the number of vertices), but not enough space to copy G. Suppose further that we do not want to delete G in the process of creating L. Is it possible, with only minor modifications to the algorithm, to successfully compute L in $O(n + m)$ time and space?

b) Could we have selected which vertex to remove next by looking for those with out-degree 0 instead of in-degree 0 (obviously we would then be adding new vertices to the beginning of L and not the end!)? I'd like you to consider this question. Precisely,

i) Suppose we change line 3 of the algorithm so that vertices of out-degree 0 (instead of in-degree 0) are selected, and we change line 4 so that vertices are added to the beginning of L. How would this impact the running time of the algorithm? Note: You don't have enough room to copy G or make the graph G_{rev}.

Solution
a) It is not necessary to delete v and its edges from G. We can simply mark v as visited and reduce all of its neighbors' indegree by one. Eventually, another vertice will have indegree of zero and the algorithm works with this modification.

b) Finding the vertex with in-degree 0 is a constant operation since it will have no edges pointing to it in our adjacency list. We can keep track of this vertex in a variable when the adjacency list is created. Now, finding the vetice with out-degree 0 will require us to search our adjacency list which takes $O(n + m)$ since it is set up in the previously stated way. We would have to do this operation $O(n + m)$ times because that is the runtime of our original algorithm. Thus, the runtime of this modified algorithm is $O(n + m) * O(n + m) = O(n + m)^2$. However, we can flip our adjacency list before we enter our while loop so that finding the vertice with out-degree zero is constant. This will make our runtime $O(n + m + n + m)$.