a) For a DAG, it means that it's directed acyclic graph. This means there's no cycle within the graph. Use prove by contradiction, and firstly, suppose that there exists a non-empty DAG that does not have a sink. So, this means that for each vertex u ∈ G, u has at least one edge that points to another vertex v. As a result, $|E| \geq |V|$ and since each vertex has an edge pointing out, then there must exist a cycle within the graph G. And this contradicts with the property of DAG, it's acyclic and should not have any form of cycle within. Hence, every non-empty DAG has a sink.

b) This algorithm feels just like a reversed operation of BFS by going from the leaf to the root of a tree. Since every non-empty DAG has a sink, then line 1 of the algorithm is guaranteed to run in the beginning. For a sink v ∈ V, there exists at least one vertex u ∈ V such that the edge (u, v) connects u and v. With the execution of Line(2), it's guaranteed that all sink vertices will be appended to the topologically sorted list of vertices. So now need to prove all vertices that are not sink vertex in the beginning, will become a sink vertex as the algorithm runs. When a vertex v is removed in Line(3), so will the edge that connects it with its parent u. If the vertex u only has edge that points to v, then u will now be a sink edge after the removal of the edge. And if the vertex u has other edges pointing out to other vertices, then that means there are other vertices that are sink vertex, and that vertex will be the descendant of the vertex u, as if all descendant vertices of u are non-sink vertices, then there will be a cycle which makes the graph not acyclic. So by picking those vertices as sink, remove them and edges incident of them, eventually vertex u will be a sink node, and it will be removed after all of its descendants have been removed from the graph. And this algorithm would run until there is no sink vertex, and all of the vertices will be inserted into the topologically sorted list in the reversed order. And by printing out the list in reverse, we would get a topologically sorted vertices pointing from left to right.

c) In the original algorithm, picking a sink and adding it to the topologically sorted list of vertices takes linear time. But removing all edges incident on the sink does not take linear time, and at Line(4) we need to repeat the algorithm, hence there must be a large loop which has a time efficiency of $\Theta(|V|)$ that contains all of Line(1) to (3). As it's impossible to remove the outer-most loop, then need to figure out a way to work with the algorithm on the inside. So a good way may be to generate a linked list l while first traversing the graph. For the linked list l, we would record for each entry/vertex of it, there is a list containing all other vertices that points to that vertex. For example, if vertex 4 and 5 point to vertex 3, then for $l[3]$, it would be [4, 5]. Let n represent $(|V| + |E|)$, and this would take time $\Theta(n)$. While Line(1) and Line(2) of the original algorithm takes $O(|V|)$ and $O(1)$ time, Line(3) would take at most $O(n)$ time by using the linked list l. So these would take $\Theta(n)$ time.