In this question we will try out some of the graph algorithms we have seen in class. Draw the annotated graphs that result from applying each of the following algorithms. You do not need to show the running states of the algorithms (i.e. the stacks, queues, etc) just the final result. Whenever you have a choice of exploring two vertices, visit them in alphabetic order (i.e. \( a \) before \( c \), or \( u \) before \( v \)). This also means that each of the algorithms should begin at \( a \).

(a) (10 pts) **Test-Bipartiteness**\((G_1)\) to obtain a BFS tree similar to the one shown in class. Colour the vertices depending on whether they are at an even or odd level (black and white is fine) and draw the non-tree edges as dashed lines. What is the bipartite partitioning?

**ANSWER:** The partitioning is \( X = \{a, e, f, j\} \) and \( Y = \{b, c, d, g, h, i\} \).

(b) (10 pts) **Top-Sort**\((G_2)\) to obtain the DFS tree. Number the vertices and draw the non-tree edges as dashed lines. What is the topological ordering?

**ANSWER:** The topological ordering is \( h, a, d, c, j, b, f, g, e, i \).
2. (10 pts) Give an $O(n + m)$ time algorithm to decide whether an undirected graph $G$ contains a cycle of odd length, and, if so, outputs one. Then, argue that your algorithm (i) terminates, (ii) is correct, and (iii) has the claimed running time.

(Hint: What algorithm from class determines whether an undirected graph contains a cycle of odd length? Modify the algorithm to output the cycle.)

**ANSWER:**

**Find-Odd-Cycle($G$)**

1. $\triangleright S$ is a Stack
2. Run BFS on $G$ to label all vertices with their distance $d(v)$ from some vertex $s$ and their parent in the BFS tree
3. for every edge $(v, w)$ of $G$
   4. do if $d(v) = d(w)$
      5. then while $v \neq w$
         6. do Output $v$
         7. Push($S$, $w$)
         8. $v \leftarrow v$’s parent
         9. $w \leftarrow w$’s parent
      10. Output $v$
6. while $S$ is not empty
7. do $w \leftarrow \text{Pop}(S)$
8. Output $w$
9. exit
10. Report that $G$ contains no odd cycles

(i) The algorithm terminates because it runs BFS and then looks at each edge at most once in line 3 and then each vertex at most once when outputting the cycle.

(ii) As we proved in class, if there is an odd cycle then there are two vertices $v$ and $w$ that are adjacent and at the same level of the BFS tree. Each vertex has one parent in the BFS tree, so $v$ and $w$ have a least common ancestor $l$, and the set of vertices $v \to l \to w$ form an odd cycle. If there is no odd cycle then the algorithm reaches the end because $\text{Report-Bipartiteness}(G)$ does and then reports that $G$ contains no odd cycles.

(iii) BFS takes $O(n + m)$ time. The algorithm considers every edge at most once in line 3 and every vertex at most once in lines 5—10. Thus, the algorithm runs in linear time.
3. (10 pts) The reverse of a directed graph $G = (V, E)$ is another directed graph $G^R = (V, E^R)$ on the same vertex set, but with all edges reversed; that is, $E^R = \{(v, u) : (u, v) \in E\}$.

Give a linear-time algorithm for computing the reverse of a graph in adjacency list format. Then, argue that your algorithm (i) terminates, (ii) is correct, and (iii) has the claimed running time.

**ANSWER:**

Let $G^R$ be the reverse of $G$

1. **Reverse($G$)**
   
   1. for each edge $(v, u) \in E$
   2. do reverse $(v, u)$ to get $(u, v)$

   (i) The algorithm terminates because it considers each edge once.

   (ii) The algorithm is correct because it reverses each edge. Thus, we have the new graph $G^R = (V, E^R)$ where $V$ is the original vertex set and $E^R$ is the set of reversed edges.

   (iii) The algorithm runs in time $O(m)$ because it does a constant amount of work for each of the $O(m)$ edges. We simply look at each element of the edge list in our adjacency list representation and swap the vertices.

4. Suppose that you are trapped with nothing but chalk in an enchanted labyrinth. The labyrinth consists of rooms connected by doors. One room contains the exit.

   (a) (1 pt) How can you use the chalk to implement DFS from your current position?

   **ANSWER:** We can use the chalk to mark the rooms (vertices) and doors (edges). We can apply each operation of recursive DFS by walking through any unmarked door and marking the door with the direction that we entered. If the room is explored we mark the door as a non-tree door and backtrack. If the room is unexplored, but not the exit, we mark the room as explored and continue our search. When there are no unexplored doors in a room we backtrack. When we reach the exit we are done.

   (b) (2 pts) If you could not mark the doors of the labyrinth, could you still implement DFS by only marking rooms?

   **ANSWER:** Yes. Any door that leads to a marked room is either a tree edge or a non-tree edge. We can thus tell if a door is unexplored by just marking rooms.

   (c) (2 pts) If you could not mark the rooms of the labyrinth, could you implement DFS by only marking doors?

   **ANSWER:** Yes. Any room that has a marked door (besides the one we just used) is an explored room and any room that does not have a marked door is an unexplored room. We can thus tell if a room is unexplored without marking it.
(d) (5 pts) Finally, suppose that doors between rooms can appear and disappear. However, the doors do not change arbitrarily—there is always a path of unexplored rooms from your current position in the labyrinth to the exit (and you can see the mark of a room before passing through a door). Argue that DFS will still reach the exit.

**ANSWER:** DFS will still reach the exit because there is always a path of unexplored rooms to the exit and we never enter an explored room. Thus, DFS will explore each room at most once and must eventually reach the exit.

(e) (BONUS: 5 pts) How long does DFS take in 4(d)?)

**ANSWER:** DFS takes $O(n^2)$ time in 4(d). Even if the graph has only $O(n)$ edges, there could be $\Theta(n)$ edges to consider from each visited room for a total of $O(n^2)$ time.