1. BFS: d, b, i, n, m (other vertices not reached)
   DFS: d, b, n, m, i (other vertices not reached)

2.
   ```
   int[][] convertToMatrix(LinkedList[] list, int maxVertex) {
           int[][] matrix = new int[maxVertex][maxVertex];
           for(int i = 1; i <= list.length; i++){
                   Node curr = list[i].head;
                   while(curr!=null){
                   matrix[i][curr.data] = 1;
                   matrix[curr.data][i] = 1;
                   curr=curr.next;
                   }
           }
           return matrix;
   }
   ```

   Time complexity is O(# of edges) because this code visits every element in the adjacency list and therefore sees every edge twice.

3. The DFS function would set all vertices in the graph to be unvisited, then for every vertex in the graph, if it has not been visited, it will call graph_DFS. That function will first print the vertex and set it to be visited. Then, it will traverse through that vertex's row in the matrix, recursively calling itself if the current element is set to 1 and the adjacent node with its value equal to the column number has not been visited, the parameter for the call being that adjacent node.
   The time coplexity should be equal to the largest value in the graph^2 $*$ the number of vertices in the graph, making it $O(matrix.size + |V|)$.
   We use the adjacency list because the version using it has a time cost smaller than the version with the matrix.

4. The code does not work because of the nature of BFS vs. the nature of DFS. The reason that after DFS is completed, the end times are topological is because DFS traverses as far as it can for each step, without using a frontier system. BFS pauses at every step and looks at other adjacent nodes first because of its use of a frontier, thus making the end times for the nodes non-topological (If (u, v) exists in E, then u.end_time is not necessarily larger than v.end_time).