1. Complexity:
    a. Double for loop over all contents of an array, totally summing to V(V-1)/2. This is $O(V^2)$.
    b. Perform V increments of the first for loop, and the total internal loop is the sum of all degrees, which is $2E$. Totally, $O(V + 2E)$.
    c. Same condition as 1a; $O(V^2)$.
    d. V increments of the first loop, but now sum of all outdegrees is only E since there is no duplication in the adjacency list. $O(V + E)$.
2. Assuming that the adjacency list is stored in increasing order, DFS could look like:
    a.

i	0	1	2	3	4	5
marked	T	T	T	T	T	T
edgeTo	-	4	0	2	2	1

    b. BFS, assuming increasing order:

i	0	1	2	3	4	5
marked	T	T	T	T	T	T
edgeTo	-	4	0	2	0	0

3. Assuming increasing order:
    a.

i	0	1	2	3	4	5
marked	T	F	T	T	T	T
edgeTo	-	-	0	2	2	4

    b. And BFS:

i	0	1	2	3	4	5
marked	T	F	T	T	T	T
edgeTo	-	-	0	2	0	0

4. Iterative DFS uses a stack data structure to replicate the recursion call stack.

```
DepthFirstPaths(Graph G, int startingVertex)
   a. Mark startingVertex as visited
   b. Push startingVertex onto Stack
   c. While Stack contains vertices:
        i.  Pop currentVertex from Stack
       ii.  Mark currentVertex as visited
      iii.  For all adjacentVertex adjacent to currentVertex
               1. If adjacentVertex is not visited
                     a. Push onto stack
                     b. edgeTo for adjacentVertex = currentVertex
```

This while loop will execute as long as vertices are contained on it. At a minimum, this contributes an $O(V)$ term to the complexity. For every vertex, the for loop will check adjacent vertices using edges, even if it doesn't pass the conditional. This would be 2E. Totally, $O(V + 2E)$. (assuming undirected graph, directed graph is V+E).

Recursive BFS will still require the use of a queue to terminate execution, so it isn't a fully recursive solution as one would like. The way this will work is that the while loop will be replaced with recursive calls, effectively the inverse of what occurred during iterative DFS.

```
d. BreadthFirstPaths(Graph G, int startingVertex)
     i. Push startingVertex onto Queue;
    ii. Mark startingVertex as visited;
   iii. Bfs(G);
e. Bfs(Graph G)
     i. Dequeue currentVertex from Queue;
    ii. For every adjacentVertex adjacent to currentVertex
           1. If adjacentVertex has not been visited
                a. Mark adjacentVertex as visited;
                b. Push adjacentVertex onto vertex;
                c. edgeTo for adjacentVertex = currentVertex;
   iii. Bfs(G);
```

BFS will execute on every reachable vertex from the starting position for an undirected graph, since they will all be enqueued and checked. This will cost at minimum $O(V)$. Then, there will be a check on every edge in the adjacency list for the currentVertex, costing at most 2E. This would be $O(V + 2E)$. (assuming undirected graph; directed graph is V+E).

5. We'll use the concept of red/white coloring to implement a BFS cycle detection algorithm. We'll update the color of the graph based on the distance from the source node. This Boolean flag will be 'RED' == true or false. If we're coloring neighbors of a given BFS call, and we try to assign RED/WHITE to a node that is already WHITE/RED, then there is an odd cycle and the graph is bipartite. During execution, keep track of a queue that contains the red nodes, just like the slides.

```
a. BipartiteFirstPaths(Graph G, int startingVertex)
     i. Push startingVertex onto BFSQueue;
    ii. Push startingVertex onto BipartiteQueue;
   iii. Mark startingVertex as visited;
    iv. Mark startingVertex COLOR as RED;
     v. currentColor == WHITE;
    vi. While(BFSQueue is not empty)
           1. Dequeue currentVertex from BFSQueue;
           2. For every adjacentVertex adjacent to
              currentVertex
                 a. If adjacentVertex is not visited
                      i. Mark adjacentVertex as visited
                     ii. Set edgeTo of adjacentMatrix to
                         currentVertex
                    iii. Mark currentVertex COLOR as
                         currentColor;
                     iv. If(currentColor == RED)
                            1. Push currentVertex onto
                               BipartiteQueue;
                 b. Else if adjacentVertex COLOR !=
                    currentColor
```

```
                    i. Graph is not bipartite. Terminate.
            3. currentColor = !currentColor;
```

Executing the trace, we'll return the contents of the BipartiteQueue, the color true/false array, and the edgeTo matrices.

i	0	1	2	3	4	5	6
edgeTo	-	0	0	1	2	0	0
Color	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
Visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

bipartiteQueue	0	3	4

6. For cycle detection, we'll use a DFS call. Cycles are easily found when we're executing a DFS trace and find the same vertex is visited twice (since this is an undirected graph, we'll make sure to explicitly skip the call to the currentVertex). Note: you could use a stack in the beginning without backtracking, but this was the first way that came to my head.

```
a. Public DepthFirstCycle(Graph G, int startingVertex)
     i. Dfs(G,s)
    ii. CycleNotFound = true;
b. DFS(Graph G, int currentVertex)
     i. Mark currentVertex as visited
    ii. For every adjacentVertex adjacent to currentVertex &&
        adjacentVertex != currentVertex
           1. If adjacentVertex is not visited
                 a. edgeTo adjacentVertex = currentVertex
                 b. dfs(G,adjacentVertex)
           2. else if adjacentVertex is visited &&
              CycleNotFound
                 a. cycleNotFound = false;
                 b. iterateVertex = currentVertex;
                 c. cycleStack.push(adjacentVertex);
                 d. while edgeTo[iterateVertex] ! =
                    adjacentVertex
                      i. cycleStack.push(iterateVertex);
                     ii. iterateVertex = edgeTo[iterateVertex];
                 e. cycleStack.push(iterateVertex)
```

Trace of these variables on the given graph would look like this:

i	0	1	2	3	4	5	6
edgeTo	0	0	3	1	2	4	4
visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

cycleStack	0	2	3	1	0
cycleNotFound	FALSE				
adjacentVertex	0				
currentVertex	2				
iterateVertex	0				
edgeTo[iterateVertex]	0				

7. The optimal solution for this problem is Tarjan's Algorithm for Strongly Connected Components, which works this way:
    a. Execute DFS in a loop over every component of the graph.
    b. When visiting nodes, keep track of the minimumVertex used to reach that vertex. When initially running, the minimumVertex = the current vertex value.
    c. When visiting vertices, add them to a stack of visited vertices.
    d. During function returns for DFS, if the returning vertex is on the visiting stack then set the minimumVertex for that node equal to the smaller value between the returning vertex and the current vertex.
    e. Once all adjacent vertices are visited, if a cycle is detected, then remove vertices from visit stack until we reach the current vertex.

Detailed pseudocode for this problem is available below:

https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

This is one algorithm for detecting SCCs of a graph in linear time. However, other implementations you might have tried could be a variant on Kosaraju's Algorithm for SCCs:

https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm