1a.

The code shown on slide 17 goes through every vertex and prints every vertex that is adjacent to it. An adjacency matrix representation is array that is size VxV where is V the number of vertices. Each row shows what vertices are adjacent to the index of that vertex. If a vertex is adjacent to another vertex that correspond matrix position has the value 1 otherwise the value is equal. It takes O(V) time to check the what vertices are adjacent to the current vertex since each row has V columns. Since there are a total of V vertices in the graph the total run time of the code will be $O(V) * O(V) = O(V^2)$

1b.

The adjacency list representation of a graph consists of V linked lists where each linked list contains the elements adjacent to the currently indexed vertex. This means to find every vertex adjacent to one currently being selected takes degree(v) time where degree(v) is the number vertices adjacent to v. According to the handshaking lemma. The sum of the degree of every vertex is equal to 2E where is the total number of edges. Since the code has to visit every vertex as well which takes O(V) time the total run time is equal to $O(V) + O(2E) = O(V+2E)$.

1c.

A directed graph does not change the run time of code for an adjacency matrix representation because the array is still VxV and to find the nodes adjacent to a vertex still takes v time. This means that the run time of the code for an undirected graph is still equal to $O(V^2)$.

1d.

For an adjacency list representation of a directed graph. The size of each linked list is equal to the outdegree of each vertex. The outdegree is the number of tail ends adjacent to a vertex. According to the degree sum formula the sum of every vertex's outdegree is equal to the sum of every vertex's indegree which is equal to O(E). This means that the total run time of this code for an adjacency list representation of a directed graph is O(V+E).

2a.

0	2	3	4	1	5

2b.

0	2	4	5	3	1

3a.

0	2	3	4	5	

3b.

0	2	4	5	3	

4a.

1. Construct a stack that will be used for the DSF
2. Create array of size V that states whether each vertex has been visited
3. Set every index in the array as false.
4. Push the starting vertex to examine at the top of stack
5. While the stack is not empty
    a. Set the vertex to look at equal to the top of stack
    b. Pop the top item from the stack
    c. If the current vertex has not been visited
        i. Print the current vertex
        ii. Set that that vertex as visited using array mentioned in step 2
    d. For beginning to end of current vertex's adjacency list
        i. If the vertex has not been visited
            1. Push that current vertex in the adjacency list to the top of the stack

This algorithm requires that every node has to be printed which takes O(V) time. Additionally, the algorithm also must look at the adjacent list of the vertex that is currently being examined. In the worst-case situation, the adjacency list of every vertex has to be examined. From the previous sections we know that the time it takes to see every adjacent vertex of every vertex for an adjacent list representation is equal to O(E) for a directed graph and O(2E) for an undirected graph. This means that the total big O rum time for this algorithm is O(V+E) or O(V+2E) depending on if the graph is directed or not.

4b.

Assumption: Queue is passed to function that contains first vertex to look at. Before the function is called a vertex should be set to not discovered using these steps

1. Create array of size V that states whether each vertex has been visited
2. Set every index in the array as false (not visited).

Main recursive function.

1. If the queue is empty
    a. Return and end the function
2. Set current vertex being examined as the front of the queue
3. Remove the top item from the queue
4. Print that current vertex
5. For beginning to end of current vertex's adjacency list
    a. If the current vertex has not been visited
        i. Set the current vertex as visited

        ii. Push the current vertex on to the queue
    6. Call function again (start at step 1) and pass the current queue, graph, and what has been discovered

The run time for this algorithm is also equal to O(V+E) or O(V+2E) for the same reason as the one mentioned in the previous graph. Every vertex has to be visited which takes O(V) and it takes O(E) for a directed graph and O(2E) for an undirected graph to see the adjacency list for every vertex when the graph is represented using an adjacency list.

5.

1. Create a color array of size equal to the total number of vertices to indicate the color of each vertex
    a. Assume that a vertex can either be not colored, set to color 1 or set to color 2
2. For beginning of color array to end
    a. Indicate that every vertex has not been colored
3. Create a queue to store the vertices
4. Set the first node to look at as one color using the array mentioned in step 1
5. Push the current vertex on the queue
6. While queue is not empty
    a. Set the vertex to look at equal to the top of queue
    b. Pop from the queue
    c. if the current vertex adjacency list is empty
        i. state that the graph is not bipartite and end function
    d. For beginning to end of current vertex's adjacency list
        i. If the adjacent vertex currently being investigated has not been colored
            1. Set color of the current adjacent vertex equal to the opposite color of the current vertex being investigated. For example, if the current vertex being investigated was color 1, the color of the adjacent vertex being looked at should be color 2
            2. Push the adjacent vertex onto the queue
        ii. Else if the adjacent vertex currently being investigated has the same color as the vertex that is being investigated
            1. state that the graph is not bipartite and end function
7. state that the graph is not bipartite and end function

For this Queue trace I am assuming that node 0 is the starting point and the adjacent list for each vertex are in ascending order. The algorithm still functions properly if they are not in ascending order.

0
Step 1

Step 2

1
Step 3

1
2
Step 4

1
2
Step 5

1
2
5
Step 6

1
2
5
6
Step 7

2
5
6
Step 8

2
5
6
3
Step 9

5
6
3
Step 10

5
6
3
4
Step 11

6
3
4
Step 12

3
4
Step 13

4
Step 14

Step 15

The queue is empty which means the graph is bipartite

6.

1. Create array of size equal to the total number of vertices that states whether each vertex has been visited
2. Set every index in the array as false (not visited).
3. For zero to total number of vertices minus one
    a. If the current node being looked at has not been visited
        i. Call the recursion function for cycle detection and pass the array mentioned in part 1, the number -1, and the current vertex being investigated
        ii. If the recursion function returns the number 1
            1. Return 1 and end function

4. Return 0 and end function

Algorithm for recursion function mentioned:

1. Set the current vertex as visited
2. For beginning to end of current vertex's adjacency list
    a. If the current adjacent vertex has not been visited
        i. Call the recursion function for cycle detection and pass the visited array, the current vertex being investigated, and the current adjacent vertex being investigated (equal to index of for loop mentioned in step 2 of this algorithm)
        ii. If the recursion function returns the number 1
            1. Return 1 and end function
    b. Else if the current adjacent matrix is not the parent (variable passed from recursion call)
        i. Return 1 and end function
3. Return 0 and end function

For this Stack trace I am assuming that node 0 is the starting point and the adjacent list for each vertex are in ascending order. The algorithm still functions properly if they are not in ascending order.

					5
				4	4
			2	2	2
		3	3	3	3
	1	1	1	1	1
0	0	0	0	0	0
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6

At this point a cycle has been detected since one of the adjacent vertices of 5 is 0 which is not its parent. A cycle could have also been detected after step 4 if the first vertex in 2's adjacent list is 0 since 0 was already visited at this point and not its parent.

7.

Main function

1. Create array of size equal to the total number of vertices that states whether each vertex has been visited
2. Set every index in the array as false (not visited).
3. Create a stack
4. For zero to total number of vertices minus one
    a. If the current node being looked at has not been visited

        i. Call a DFS function that pushes a vertex onto a stack once it has no more unvisited adjacent nodes to finish. The stack, array of visited vertices, and the current index of the for loop should be passed
5. Create a transpose of the graph (flip direction of edges)
6. Repeat Step 2
7. While the stack is not empty
    a. Set current vertex to look at as top of the stack
    b. Remove vertex from top of stack
    c. If the current vertex has not been visited
        i. Calls DFS function on transpose graph that prints the vertex result as it does the search. The array of visited vertices, and the current vertex should be passed to the function
        ii. Create an indentation or new line to separate fundamental sets

Function mentioned in 4ai

1. Set current vertex as visited
2. For beginning to end of current vertex's adjacency list
    a. If the current adjacent vertex has not been visited
        i. Recursively call this function and pass the current index, the stack and the array of visited of vertices
3. Push current vertex to the stack

Creating transpose of a graph

1. Create a graph of size equal to the total number of vertices
2. For zero to total number of vertices minus one
    a. For beginning to end of current vertex's adjacency list. The vertex is indicated by the index of the for loop
        i. Add vertex referenced by the index of step 2a. to the adjacency list of the vertex referenced by the index of step 2
3. Return the graph

Function mentioned in 7ci

1. Set current vertex as visited
2. Print current vertex
3. For beginning to end of current vertex's adjacency list
    a. If the current adjacent vertex has not been visited
        i. Recursively call this function and pass the current index, the stack and the array of visited of vertices

The total run time of this algorithm can be calculated by determining the time of each step of the main function. The time it takes to complete Step 1-2 is equal to O(V) since you have to set the value of V elements in an array where V is the total number of vertices. Creating a stack takes

O(1) time since it is initially empty. Step 4 performs a DFS that ensures every vertex is visited and pushes a vertex to stack. Based on the information described in Problem 4a, the total run time of DFS is O(V+E). The total time to push an item on to a stack is O(1), which means the total run time for step 4 is O(V+E). Step 5 creates a graph that is transpose of an original by visiting every vertex and then visiting ever vertex's adjacency list. This means that the total run time is equal to O(V+E). Step 6 is the same as step 2 which also takes O(V). Step 7 also performs a DFS that ensures every vertex is visited. This means that the run time of step 7 is O(V+E). The total run time is O(V)+O(1)+O(V+E)+ O(V+E)+O(V)+O(V+E) = O(V+E)

For this Stack trace I am assuming that node 0 is the starting point and the adjacent list for each vertex are in ascending order. The algorithm still functions properly if they are not in ascending order.

					2
				4	4
			5	5	5
		7	7	7	7
	6	6	6	6	6
3	3	3	3	3	3
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6

	0				
1	1	1			
2	2	2	2		
4	4	4	4	4	
5	5	5	5	5	5
7	7	7	7	7	7
6	6	6	6	6	6
3	3	3	3	3	3
Step 7	Step 8	Step 9	Step 10	Step 11	Step 12

7			
6	6		
3	3	3	
Step 13	Step 14	Step 15	Step 16

After step 11 the first fundamental set has been found and is 0-2-1

After step 14 the second fundamental set has been found and is 4-5-7

After step 16 the final set has been found and is 6-3.

After step 16 the stack is finished which means the algorithm has finished.