1. Undirected Graphs (symmetric) and Directed Graphs (need not be symmetric)

Undirected Graph – when one can traverse either directed between 2 nodes
Directed Graph – when once can traverse only in the specified direction between two nodes

```
for (int v = 0; v < G.V(); v++)
  for (int w : G.adj(v))
    StdOut.println(v + "-" + w);
```

a. Undirected Graphs Adjacency Matrix Representation

Prove exact time complexity in terms of V (# of vertices) and E (# of edges) for Graph G:

Exact Time complexity: $O(V^2)$

Proof: It consumes more space and even if graph contains less number of edges, it takes more space, adding vertex is $O(V^2)$ time and removing an edge takes O(1) time. The representation regarding the code is easier to implement but it takes more space.

b. Undirected Graphs Adjacency List Representation

Prove exact time complexity in terms of V (# of vertices) and E (# of edges) for Graph G:

Exact Time Complexity: O(V + 2E)

Proof: Similar time complexity as Direct Graphs Adjacency List, but the 2E is for the going to either edge since graph is not direct and head only in direction.

c. Directed Graphs Adjacency Matrix Representation

Prove exact time complexity in terms of V (# of vertices) and E (# of edges) for Graph G:

Exact Time Complexity: $O(V^2)$

Proof: Add vertex is easier, in worst case more space $O(V^2)$ can be consumed due to C(V,2) being number of edges in graph, downside of this is that queries regarding whether an edge from one vertex to another are not efficient and can be done O(V), go through each vertex again instead of edges

d. Directed Graphs Adjacency List Representation

Prove exact time complexity in terms of V (# of vertices) and E (# of edges) for Graph G:

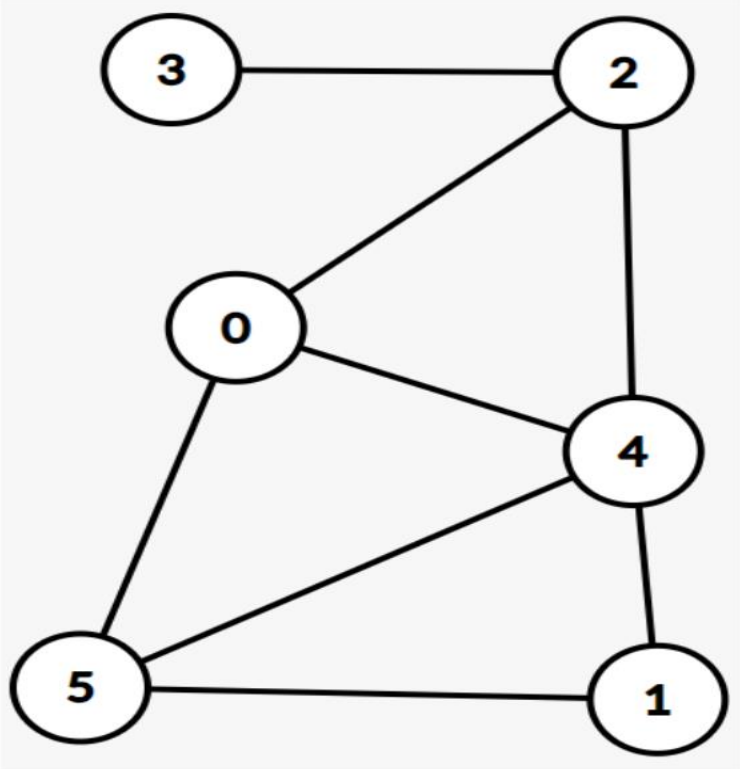
Exact Time Complexity: O(V+E)

It consumes more space and even if graph contains less number of edges, it takes more space, adding vertex is $O(V^2)$ time and removing an edge takes O(1) time. The representation regarding the code is easier to implement but it takes more space.

*side-note → list is better since one does not have to go through edge that are not there

2. Undirected

a. DFS trace



0 as a source vertex

Vertex	0	1	2	3	4	5
edgeTo[]	-	5	4	2	1	0
Marked[]	T	T	T	T	T	T

b.

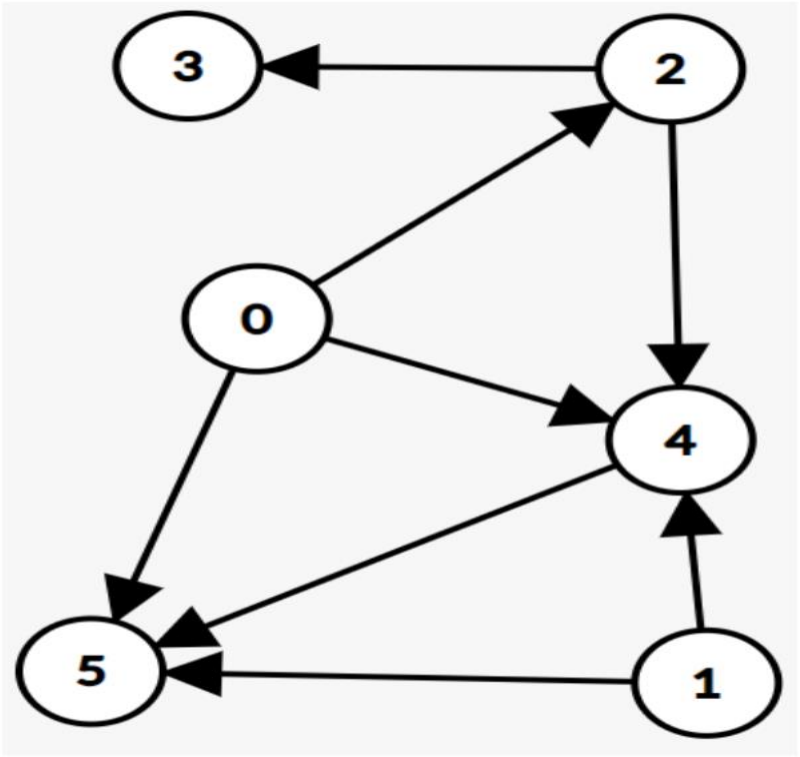
BFS trace

0 as a source vertex

Vertex	0	1	2	3	4	5
edgeTo[]	-	5	0	2	0	0
distanceTo[]	0	2	1	2	1	1

3. Directed

a. DFS trace



Vertex	0	1	2	3	4	5
edgeTo[]	-	-	0	2	0	0
Marked[]	T	F	T	T	T	T

b. BFS trace

Vertex	0	1	2	3	4	5
edgeTo[]	-	-	0	2	0	0
distanceTo[]	0	-	1	2	1	1

4. a. Iterative DFS

Big-Oh time complexity (in terms of V and E) = O(2E) = O(E)

Whichever terms is bigger will dominate the time complexity, a dense graph dominated by the number of Edges is O(E) and sparsely connect graph with some vertices disconnected, then graph will be dominated by the number of Vertices so O(V).

Proof: The time complexity is found by adding up the degree and this done by O(degree(G)). Degree(G) is the sum of all degrees of the vertex in the graph, which means for each vertex, it will get iterated over every vertex connected to it, and in the end every vertex will get accessed twice, so at most the loop can run 2•E, meaning that 2E is the sum of all degrees in the graph. O(2E) is O(E) knowing that we do not need the coefficient.

Pseudocode :

1. create stack to do iterative DFS
2. push source node into Stack
3. while stack is not empty
a. pop a vertex from stack
b. if vertex already discovered, then ignore it
c. if popped vertex v not discovered yet, then print it and process its undiscovered adjacent nodes into stack
d. do for every edge

Alternative Pseudocode:

1. Create a stack, S
2. Push the source "s" into the stack – S.push(s)
3. While stack has items -- !isEmpty()
   a. Pop off the most recent one – S.pop(v)
   b. If it is not visited, mark it as so
   c. For every vertex connected to the current vertex
      i. Grab current vertex – getVertex()
      ii. If not visited, push it into the stack – S.push(v)

b. Recursive BFS

Big-Oh time complexity (in terms of V and E) = O(V+E)

Proof: which ever terms is bigger will dominate the time complexity, a dense graph dominated by the number of Edges is O(E) and sparsely connect graph with some vertices disconnected, then graph will be dominated by the number of Vertices so O(V).

Pseudocode:

1. Create a queue used to do BFS
2. Mark source vertex as discovered
3. Push source vertex into the queue
4. While queue is not empty
   a. Pop front node from queue and print it
   b. Do for every edge
   c. Mark it discovered and push it into queue

Alternative Pseudocode:

1. Create a queue
2. Push the starting point onto the queue as mark it as visited
3. Remove the least recently added vertex for as long as the queue is not empty
4. Add the vertex's unmarked neighbors to the queue and mark them

Proof: have to store all of the vertexes in the queue which would result in a time complexity of V, relationship between the vertices and the edges

Another Pseudocode:

1. Create a queue Q
2. Choose a source vertex s, mark it, and enqueue – Q.enqueue(s);
3. While Q is notEmpty();
   a. Remove the least recently added vertex v
   b. If v has unmarked neighboring vertices:
      i. Visit neighboring vertices
      ii. Enqueue the unmarked neighboring vertices – Q.enqueue(w).
      iii. Mark the neighboring vertices.

5. Graph-processing Challenge 2 stack track of algorithm

1) Run DFS on the graph
a) Color one node red and it's adjacent node white

i) Continue this with a loop for the number of vertices in the graph

2) Create a function that will check to see if any two adjacent nodes have the same color in the entire graph.

a) This can be done by having a double for loop that will run for the number of vertices
   i) If any two adjacent nodes have the same color, then it is an odd length cycle and thus can't be bipartite
   ii) If any two adjacent nodes do not have the same color then it is bipartite

One Version

Vertex	0	1	2	3	4	5	6
edgeTo[]	-	0	3	1	2	4	-
Color	Red	White	White	Red	Red	White	-

Another Version

Vertex	0	1	2	3	4	5	6
edgeTo[]	-	0	3	1	2	4	4
Color	Red	White	White	Red	Red	White	White

Alternative way

1) Run DFS on the graph

a. As you run DFS run a Boolean function called "color" that does the following:
i. Create a counter variable i
ii. If i modulus 2==1
    1. Color the vertex white
iii. If I modulus 2 == 0
    1. Color the vertex red

2) Create a Boolean function called check that checks whether 2 connected/neighboring have the same color.

a. If two vertices of the same color share an edge/connection:
    i. Print: "Graph is not-bipartite"

b. Else
    i. Print: "Graph is bipartite"

Stack trace of algorithm in Tabular Form:

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

All points will have multiple connections so we have to choose the points to color where they will not have an adjacent/connected point with the same associated color. We can select 0,3 and 4 to be red as they are not adjacent but make up the points that connect to all others.
This is for Queue Trace assuming node 0 is starting point and adjacent list for each vertex are in ascending order.

0		1	1	1	1	1	2	2	5	5	6	3	4	
			2	2	2	2	5	5	6	6	3	4		
					5	5	6	6	3	3	4			
						6		3		4				
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12	Step 13	Step 14	Step 15

6. Graph-processing Challenge 3 stack track of algorithm

Pseudocode:

To check if graph has cycle we use DFS and recursion and to do this we make a boolean function call isCyclic:

1. Set a mark on v as visited (current node)
2. A while loop that will continue until next is null, this will be it is adjacent vertices
    a. If the vertices adjacent to this is not visited
        i. Make a recursion for that adjacent
        ii. Return true
    b. Else if the adjacent is visited and it is not the parent of the current vertex
        i. Return true
3. After loop
    a. Return false

Another pseudocode:

Use recursive function and parent to detect cycle from vertex v

1. Create a stack S
2. Push the source vertex s onto stack s
3. Move onto the next vertex in the array and mark it

a. Pop off the most recent entry in the list

b. If for any vertex v, that has neighbor vertex w:

i. If w is marked && w has a lower index in the stack

1. The graph is cyclic

ii. If w is marked && w has a higher index in the stack

1. The graph is acyclic

This case cycle is [0-5-4-6-0]

Vertex	0	1	2	3	4	5	6
edgeTo[]	6	-	-	-	5	0	4
Marked[]	T	F	F	F	T	T	T

Stack trace of algorithm in Tabular Form:

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

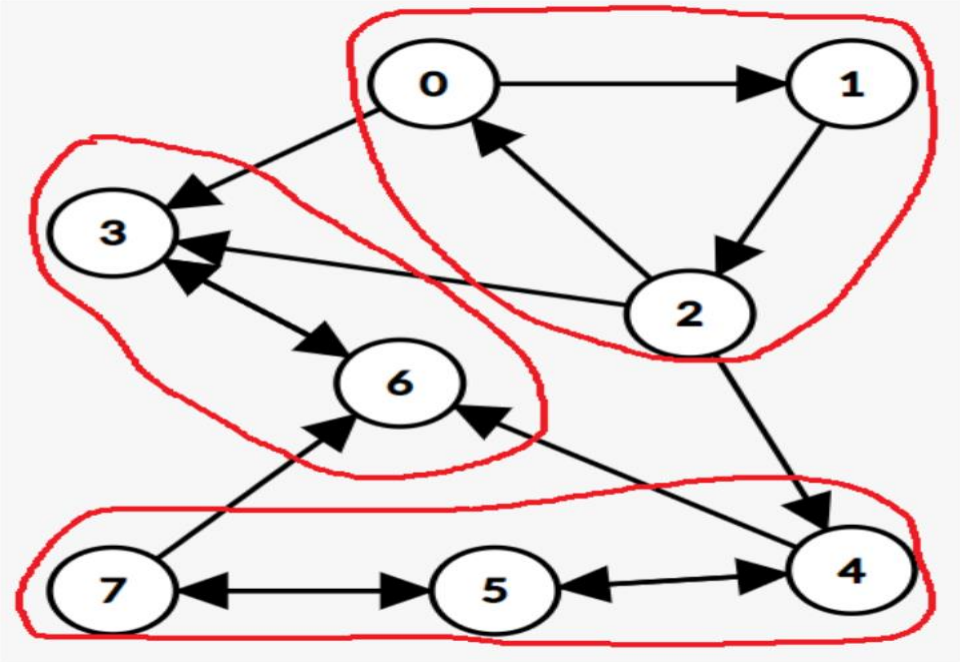
This is for Stack Trace assuming node 0 is starting point and adjacent list for each vertex are in ascending order.

					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

In order to maintain a cycle that returns to the original we can have a cycle such as 0-5-4-6-0 because it starts and ends in 0 while being capable of following the given graph's restrictions.

7.



Quick Outline (linear time algorithm):

To find fundamental sets →

1. Write a function "linear search" with "list" and "searchterm" as the parameters
2. For index from 0 → length (list)
3. If list[index] == searchterm then
4. Return index
5. ENDIF
6. End loop
7. Return -1

Pseudocode for linear time algorithm:

1. First, create a class graph with int V and a pointer to an array containing adjacency list
2. Create function to detect a cycle
    a. If the vertex equals false or not visited

        i. Mark current node as visited and make it part of recursion stack as well
        ii. Use a for loop to recur all vertices that are adjacent to this node currently
    b. Else remove the vertex from recursion stack
3. Another function that returns true if graph confirms a cycle
    a. First mark all vertices as not visited and not part of recursion attack
    b. Call previous function to detect cycles and return true if cycle
    c. Else return false

Has same time complexity of DFS

Tabular Form:

3 separated circled zones are given

1st one:

RedCircle (0,1,2)

Vertex	0	1	2
Marked[]	T	T	T
edgeTo[]	2	0	1

2nd one:

RedCircle (3,6)

Vertex	3	6
Marked[]	T	T
edgeTo[]	6	3

3rd one:

RedCircle (7,5,4)

Vertext	4	5	7
Marked[]	T	T	T
edgeTo[]	5	4	1

This is not linear case (showing a different way)

1) Can use DFS to find fundamental sets
    a. Create stack and push the starting vertex onto the stack
    b. Pop off the most recent vertex added to stack and mark it
    c. For every vertex connected to the current one we push it to the stack

2) Check to see if the adjacent vertex that has been visited equals the starting vertex
    a. If it does equal the starting vertex then partition the loop of vertices that lead to the starting vertex again
        i. Choose new starting vertex on the graph and then restart the process all over again.
    b. If it does not equal the starting vertex then move on the next vertex and keep looking

3) Continue this until all of the vertices on the graph have been visited

One Version

Vertex	0	1	2	3	4	5	6	7
edgeTo[]	2	0	1	6	7	4	3	5
Marked[]	T	T	T	T	T	T	T	T

Another Version

Vertex	0	1	2	3	4	5	6	7
edgeTo[]	2	0	1	6	5	7	3	5
Marked[]	T	T	T	T	T	T	T	T