Andy Guo
172004093

Assignment 4

1)

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

a. The time complexity is O(v^2) for Undirected adjacency matrix. To get this because the first and second for loop both goes from 0 to v. This means throughout the adjacency matrix the very row will get read every value of v. This would result in a run time of v where v is the time it takes to go through the for loop. Now looking at the edges and how its stored, we know it must go through v times as well since for adjacency matrices we might have “redundant information”, example, we need to keep track of connections that don’t exist. And since there is two for loops we do v * v, since they both traverse through the loop, giving our answer v^2.

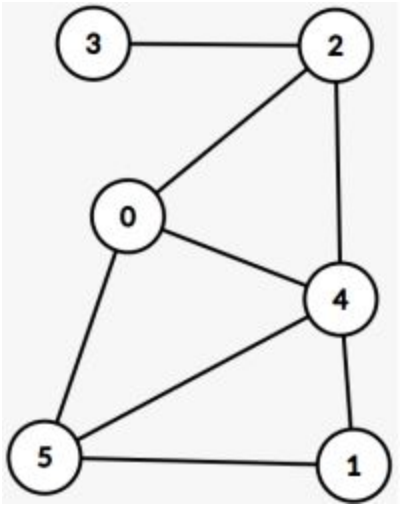
b. The time complexity is O(V+2E) for Undirected adjacency list. We know this because we always go through the loop once which results in a time completion of V. Now looking at the edges, the inner loop will be calculated once for every edge. This will give us a O(deg(v)) where deg(v) is the degree of the current node. In other words the big O is (1 + deg(1) + deg(2) + deg(3)... + deg(v)), we have a 1 in there because deg(v) might be 0. Now summing it all up, we get a runtime of O(V * 1 + deg(v1) + deg(v2) + ...). Essentially we do V (outer) + 2E (inner).

c. The time complexity is O(v^2) for Directed adjacency matrix. Directed graphs is very similar to undirected graphs. Even though we have a particular direction this time we still must check for all connection for each vertices. It will again take V times to go through every thing and another V times for all its adjacents. This will be v * v thus giving us V^2 as the time complexity.

d. The time complexity is O(V + E) for directed adjacency list. Similar to part b, we must add up all the deg(v) just like how we calculated before. This ends with the same time complexity as the O(V + E).

2) Undirected

a. DFS trace



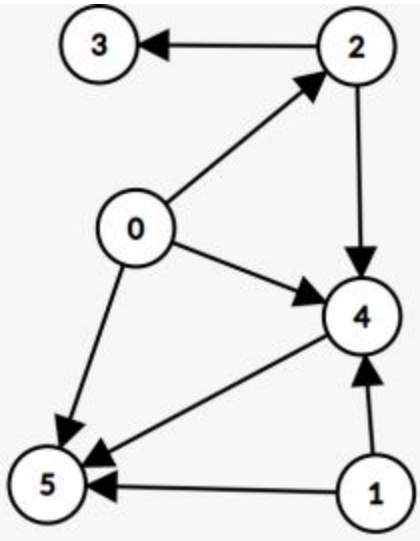
V	0	1	2	3	4	5
edgeTo[]	-	5	4	2	1	0
marked[]	T	T	T	T	T	T

b. BFS trace

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

3) Directed

a. DFS



V	0	1	2	3	4	5
edgeTo[]	-	-	0	2	0	0
marked[]	T	F	T	T	T	T

b. BFS

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

4) Big-Oh time complexity & Pseudocode

a. Iterative DFS

i. The time complexity is found by adding up the degree, we do this by O(degree(G)). In this problem the degree(G) is the sum of all the degree of the vertex in the graph. This means for each vertex, it will get iterated over every vertex connected to it. In the end every vertex will get accessed twice. This means at most the loop can run 2*E. This means that 2E is sum of all degrees in the graph. Knowing that we don't need the coefficient, we know that O(2E) is O(E).

ii. To do this code:

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 isn’t 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
    i. The time complexity for this is O(V + E)
    ii. To do this code:
        1. Create a queue
        2. Push the starting point into the queue, mark it as visited
    iii. Remove the oldest one added to the vertex, if the queue is not empty
    iv. Add the vertex of the unmarked neighbors to the queue and mark them too

5) Challenge problem 2 of undirected graphs

We can checking using either DFS or BFS. Using DFS we will need to we are going to make a boolean function called isBipartite:

1. First make a array that will store all the colors; everything in this array will have NO COLOR (using a for loop) and set the color of the source RED
2. While its neighbors is less than the total number of vertices
    a. We will make sure that there is no color in the neighbor and that the color is different from the current.
        i. If there color of the neighbor and current is the same return false.
    b. We will check to make sure that the color neighbor has no color
        i. Assign the neighbors with colors of RED or WHITE accordingly
        ii. Push the color into the stack
        iii. Set elements to the neighbor and set neighbor to 1
    c. Increment the neighbor
3. Outside the while loop we will pop the stack
4. Return true, since the stack has no self loop and the color is different from the adjacent return true.

V	0	1	2	3	4	5	6
edgeTo[]	-	0	3	1	2	4	4
color	red	white	white	red	red	white	white

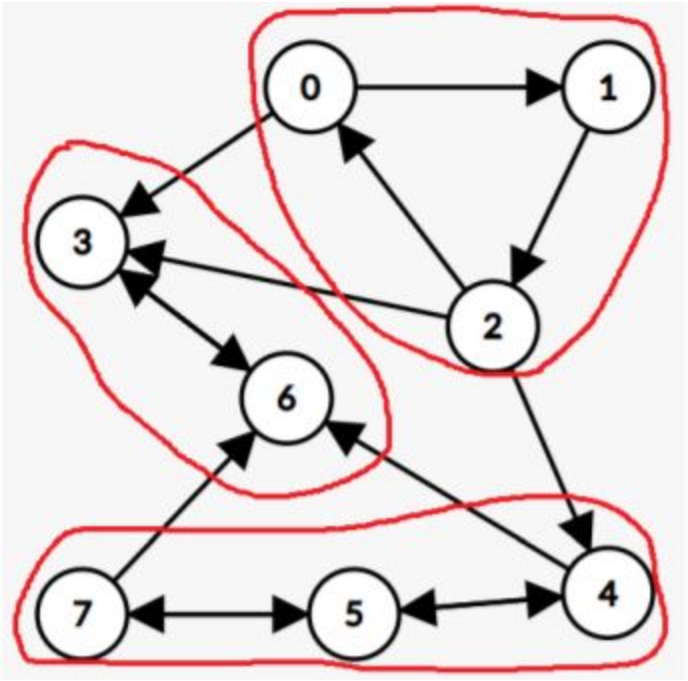
6) To check if a graph has a cycle we use DFS and recursion to solve this. To do this we will make a boolean function call is isCyclic:

1. Set a mark on v as visited, this is our current node
2. A while loop that will continue until next is null, this will be its 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 the loop
    a. Return false

This is one case shown in the slide, in this case we have a cycle of [0-5-4-6-0]

V	0	1	2	3	4	5	6
edgeTo[]	6	-	-	-	5	0	4
marked	T	F	F	F	T	T	T

7)



1. Set a mark on v as visited, this is our current node
2. A while loop that will continue until next is null, this will be its 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 the loop
    a. Return false
4. Remove that cycle from the stack and select a new node as the current and repeat the process again.

V	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