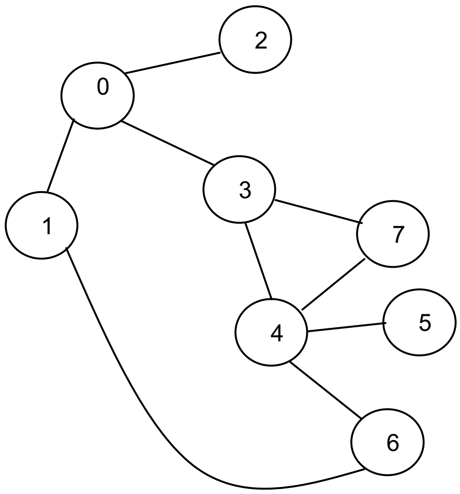
Mystery Algorithm!

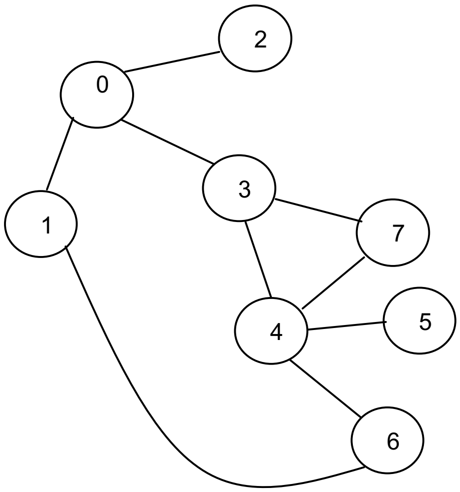



```
ALGORITHM MYSTERY( G = (V,E),  start_v )
    mark all vertices in V as unvisited
    mystery( start_v )


mystery( v )
    mark vertex  v  as visited
    PRINT  v
    for each vertex w adjacent to v do
        if w is not yet marked as visited
            mystery( w )
```

What is the output of MYSTERY when called with the graph above and start_v = 0? So that we all get the same results, when considering “each vertex w adjacent to v”, consider the adjacent vertices in numerical order from lowest to highest.

Depth-first Search Algorithm




```
// performs a depth-first search of G
// starting at vertex start_v
Algorithm DFS( G=(V,E), start_v )
    mark all vertices in V as unvisited
    dfs( start_v )


dfs( v )
    mark v as visited
    for each vertex w adjacent to v do
        if w is not yet marked as visited
            dfs( w )
```

Depth-first Search Tree

Depth-first search tree:

Tree edge:

Back edge:

Draw the depth-first search tree corresponding to the call DFS(G, 4) on the undirected graph below which is represented using adjacency lists. When considering "each vertex w adjacent to v", do so in the order in which they appear in v's adjacency list. What do you notice is different about your depth-first search tree?

Graph G – Adjacency Lists Representation

Vertex	Adjacency List
0	→ 3 → 2 → 4
1	→ 6 → 7
2	→ 0 → 3 → 5 → 4
3	→ 0 → 2
4	→ 2 → 0
5	→ 2
6	→ 7 → 1
7	→ 1 → 6

```
// performs a depth-first search of G
// starting at vertex start_v
Algorithm DFS( G=(V,E), start_v  )
    mark all vertices in V as unvisited
    dfs( start_v )


dfs( v )
    mark v as visited
    for each vertex w adjacent to v do
        if w is not yet marked as visited
            dfs( w )
```

- A graph is **connected** if there is a path between every pair of vertices. What happens when you start DFS(G, start_v) on a graph that is not connected? How can you use DFS to determine if a graph is connected or not?

- Can you come up with a real world application involving a graph for which it would be useful to know if the graph is connected. Explain what the vertices and edges represent in your application and explain why knowing if it is connected is useful.

--- DFS Implementation Using Adjacency Lists ---

```
public class UndirectedGraph {
   private Vertex[] vertices;

   private class Vertex {
       private Edge head;


   }
   private class Edge {
       private int dst;
       private Edge next;
       private Edge(int d, Edge n) {dst=d; next=n;}
   }
   public UndirectedGraph( int numVerts ) {…}
   public void addEdge( int v, int w ) {…}
   public void removeEdge( int v, int w ) {…}

   public boolean connected( ) {
       // mark all vertices as unvisited






       // invoke dfs starting at vertex 0


       // check if all vertices were visited





   }
```

```
public void dfs( int v ) {
    // mark v as visited



    // for each vertex w adjacent to v do
    //     if w is not yet marked as visited
    //          dfs(w)
















}
```

Running Time of Recursive DFS
When Graph is Represented using Adjacency Lists

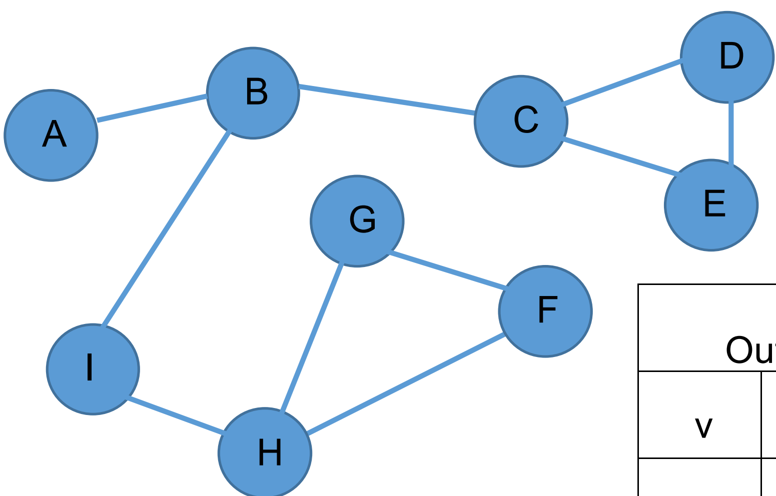
```
// performs a depth-first search of G
// starting at vertex start_v
Algorithm DFS( G=(V,E), start_v  )
    mark all vertices in V as unvisited
    dfs( start_v )


dfs( v )
    mark v as visited
    for each vertex w adjacent to v do
        if w is not yet marked as visited
            dfs( w )
```

Breadth-first Search

For the graph below, trace through the execution of the Breadth-first search algorithm starting at vertex B. Show the printed output in the chart. Then number the graph vertices in the order in which they are marked as visited by the algorithm. So that we all get the same answer, when you consider "each vertex w adjacent to v", consider the adjacent vertices in alphabetical order.

```
Algorithm BFS( G=(V,E) , start_v )
    mark each vertex in V as unvisited
    mark start_v as visited
    let Q be an empty queue
    Q.enqueue( start_v )
    while Q is not empty
        v ← Q.dequeue()
        for each vertex w in V adjacent to v do
            if w is unvisited
                mark w as visited
                Q.enqueue(w)
        print v and contents of Q
```



Output Printed by BFS Algorithm Above	
v	contents of Q (with the front item on left)

What do you notice about the order in which breadth-first search marks the vertices as visited? (Hint – think about how far they are from the start vertex.)

Suppose you want to modify bfs so that it assigns each vertex a number which is the number of edges it is from the start vertex. For example, start_v.distance should be set to 0 because it is the start vertex. Each vertex w adjacent to start_v should have w.distance set to 1 because it is one edge away from s, and so on... Mark modifications on the algorithm below that set the distance fields appropriately.

```
Algorithm BFS( G=(V,E) , start_v )
    mark each vertex in V as unvisited
    mark start_v as visited
    let Q be an empty queue

    Q.enqueue( start_v )

    while Q is not empty

        v ← Q.dequeue()

        for each vertex w in V adjacent to v do

            if w is unvisited

                mark w as visited

                Q.enqueue(w)
```