# Breadth First Search

https://cs.pomona.edu/classes/cs140/

# Outline

## Topics and Learning Objectives

- Discuss breadth first search for graphs

## Exercises

- Continued from previous lecture slides
- Compute distance with Breadth-first search

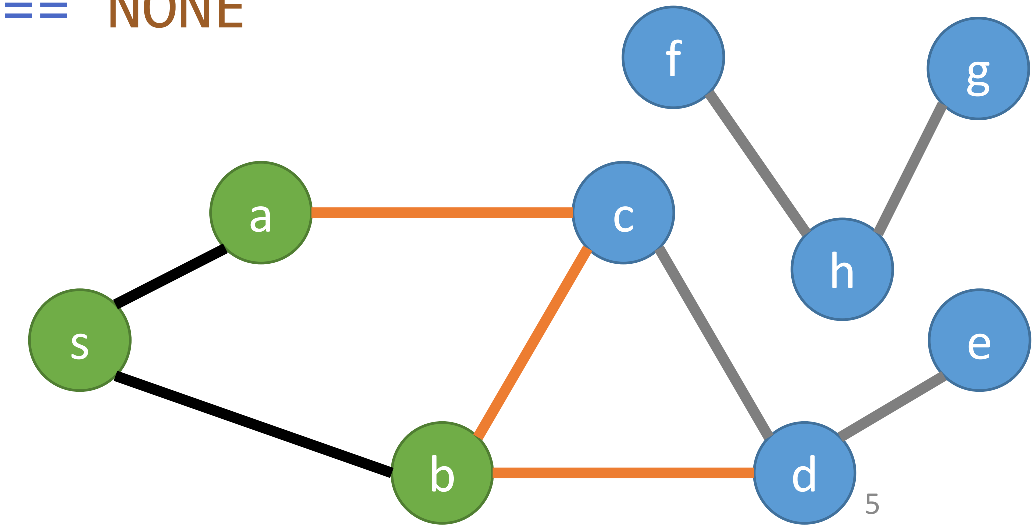
# Extra Resources

- Introduction to Algorithms, 3rd, Chapter 22
- Algorithms Illuminated Part 2: Chapter 8

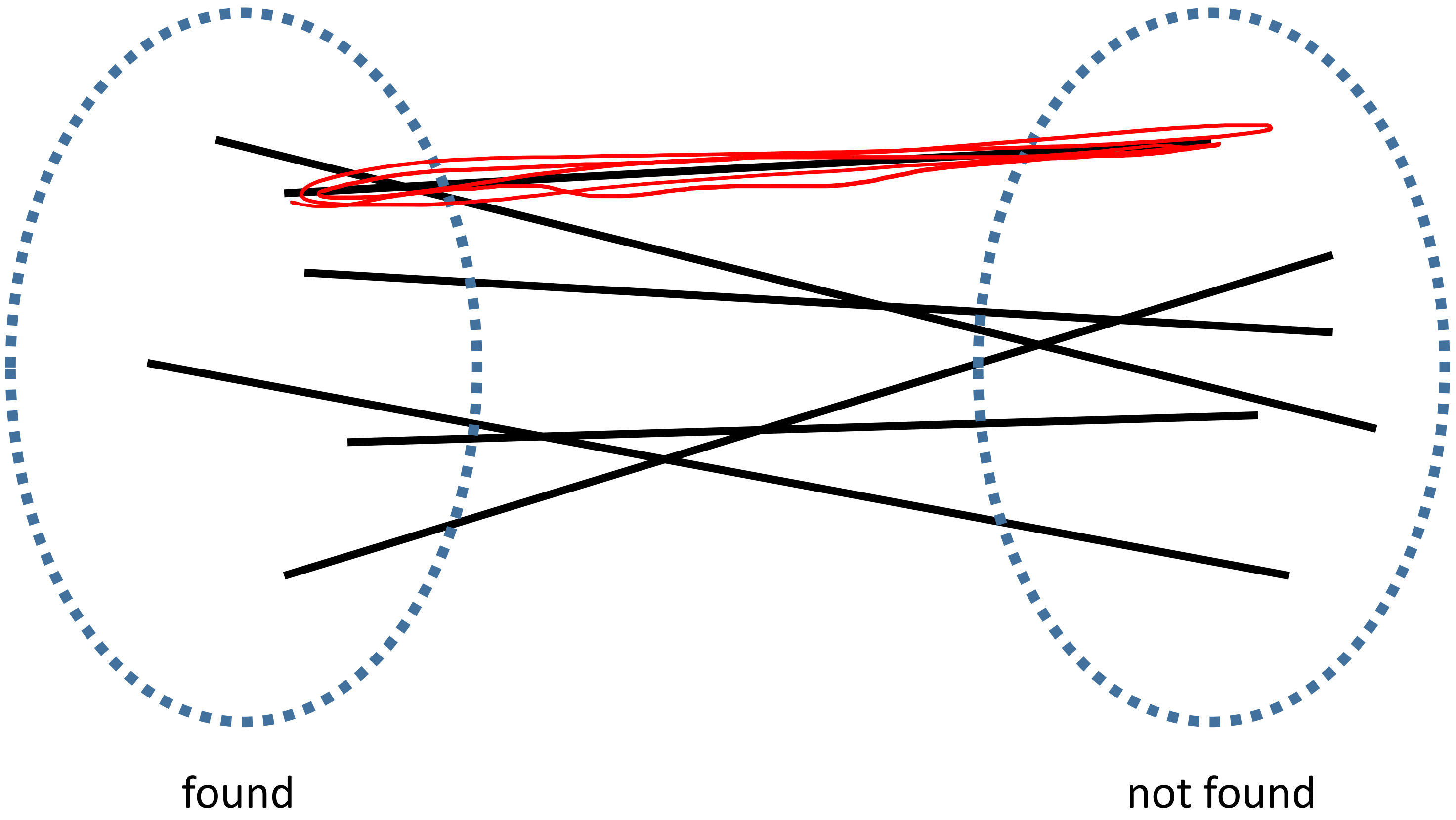
# General Algorithm

```
FUNCTION Connectivity(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    LOOP
        (vFound, vNotFound) = get_valid_edge(G.edges, found)
        IF vFound == NONE || vNotFound == NONE
            BREAK
        ELSE
            found[vNotFound] = TRUE
    RETURN found
```

Find an edge where one vertex has been found and the other vertex has not been found.



# How do we choose the next edge?



# Two common (and well studied) options

Breadth-First Search

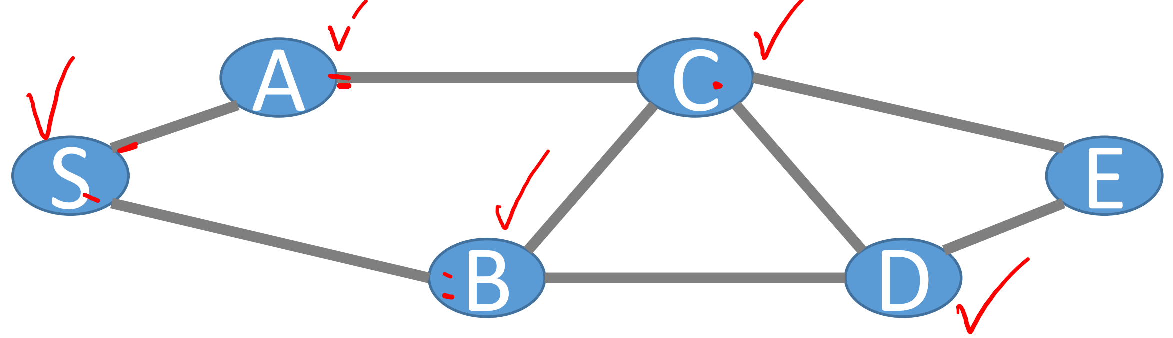
- Explore the graph in layers
- *"Cautious"* exploration
- Use a FIFO data structure (can you think of an example?)

Depth-First Search

- Explore recursively
- A more *"aggressive"* exploration (we backtrack if necessary)
- Use a LIFO data structure (or recursion)

```
FUNCTION BFS(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    visit_queue = [start_vertex]

    WHILE visit_queue.length != 0
        vFound = visit_queue.pop()
        FOR vOther IN G.edges[vFound]
            IF found[vOther] == FALSE
                found[vOther] = TRUE
                visit_queue.add(vOther)

    RETURN found
```

```
FUNCTION Connectivity(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    LOOP
        (vFound, vNotFound) =
            get_valid_edge(G.edges, found)
        IF vFound == NONE || vNotFound == NONE
            BREAK
        ELSE
            found[vNotFound] = TRUE
    RETURN found
```




# Exercise questions 2 and 3

```
FUNCTION BFS(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    visit_queue = [start_vertex]
                        = [S A B C D E

    WHILE visit_queue.length != 0
      → vFound = visit_queue.pop()
        FOR vOther IN G.edges[vFound]
            IF found[vOther] == FALSE
                found[vOther] = TRUE
                visit_queue.add(vOther)

    RETURN found
```

*Given a tie, visit edges are in alphabetical order*

vFound = S
vO = A
vO = B
vF = A
vO = C
vO = S
vF = B
vO = C
vO = D
vO = S

vF = C
vO = A
vO = B
vO = D
vO = E
vF = D

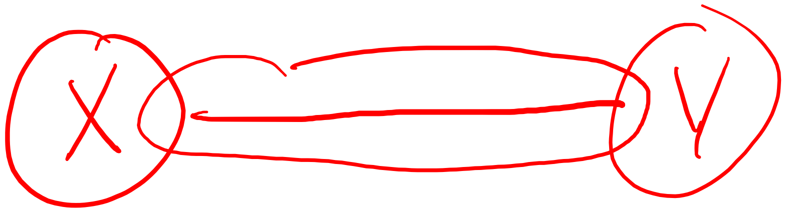
SABCDE

# Running Time

```
FUNCTION BFS(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    visit_queue = [start_vertex]

    WHILE visit_queue.length != 0
        vFound = visit_queue.pop()
        FOR vOther IN G.edges[vFound]
            IF found[vOther] == FALSE
                found[vOther] = TRUE
                visit_queue.add(vOther)

    RETURN found
```

x, y

What is the running time?




How many times to we consider each edge?

twice

$\cancel{O(m) \cdot O(n)}$

$O(n) + O(m) = O(n+m)$

# Running Time

What is the running time?

```
FUNCTION BFS(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    visit_queue = [start_vertex]

    WHILE visit_queue.length != 0
        vFound = visit_queue.pop()
        FOR vOther IN G.edges[vFound]
            IF found[vOther] == FALSE
                found[vOther] = TRUE
                visit_queue.add(vOther)

    RETURN found
```

How many times to we consider each edge?

$$T_{BFS}(n, m) = \underbrace{O(n_s + m_s)}$$

where $n_s$ and $m_s$ are the nodes and edges **findable/connected** from/to the start vertex

# Proof: BFS

**Claim**: BFS finds all nodes connected to the start node.

At the end of the BFS algorithm, v is marked found if there exists a path from s to v

- Note: this is just a special case of the general algorithm that we proved by contradiction

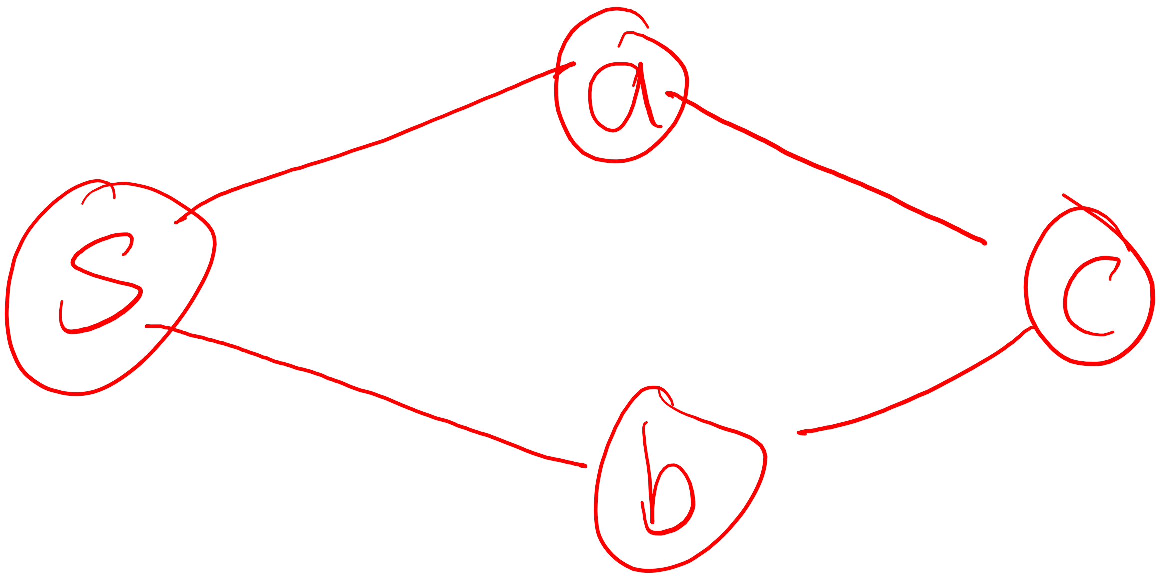
Practice for a loop invariant

Homework question

# Question

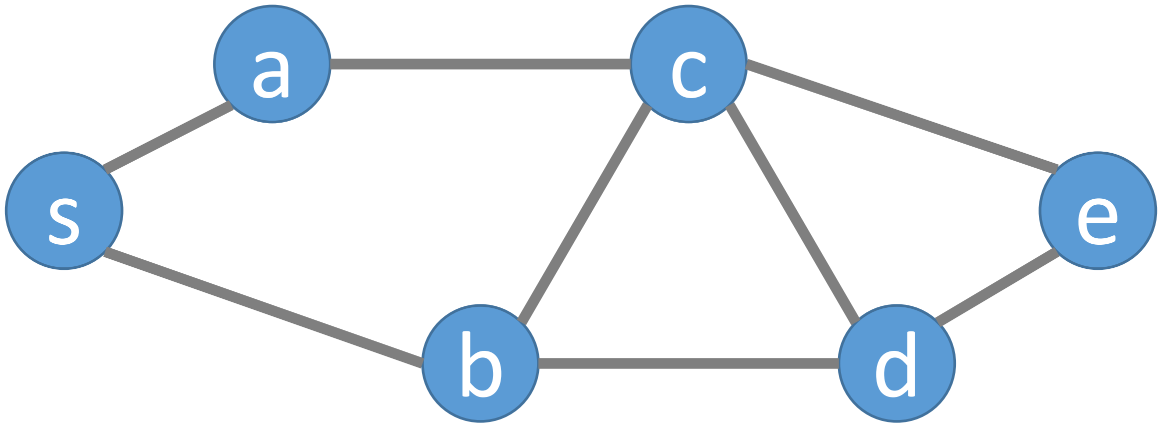
The Shortest Path Problem

- How can we determine the fewest number of hops between the start vertex and all other connected vertices?



BFS Exercise Question 1

How can we determine the fewest number of hops between the start vertex and all other connected vertices?




```
FUNCTION BFS(G, start_vertex)
    found = {v: FALSE FOR v IN G.vertices}
    found[start_vertex] = TRUE
    visit_queue = [start_vertex]

    WHILE visit_queue.length != 0
        vFound = visit_queue.pop()
        FOR vOther IN G.edges[vFound]
            IF found[vOther] == FALSE
                found[vOther] = TRUE
                visit_queue.add(vOther)

    RETURN found
```

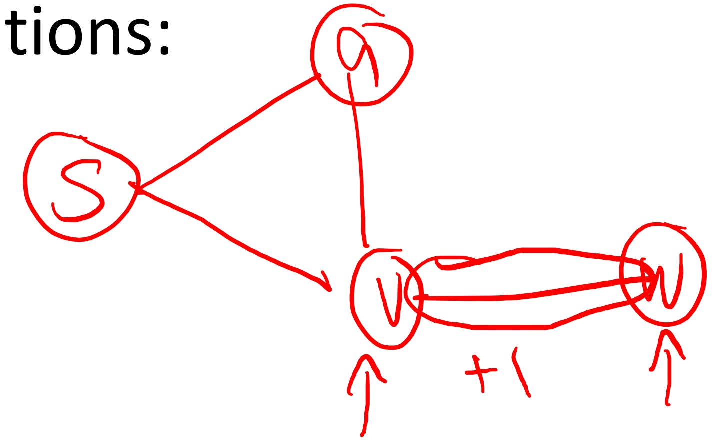
*Given a tie, visit edges are in alphabetical order*

# The Shortest Path Problem

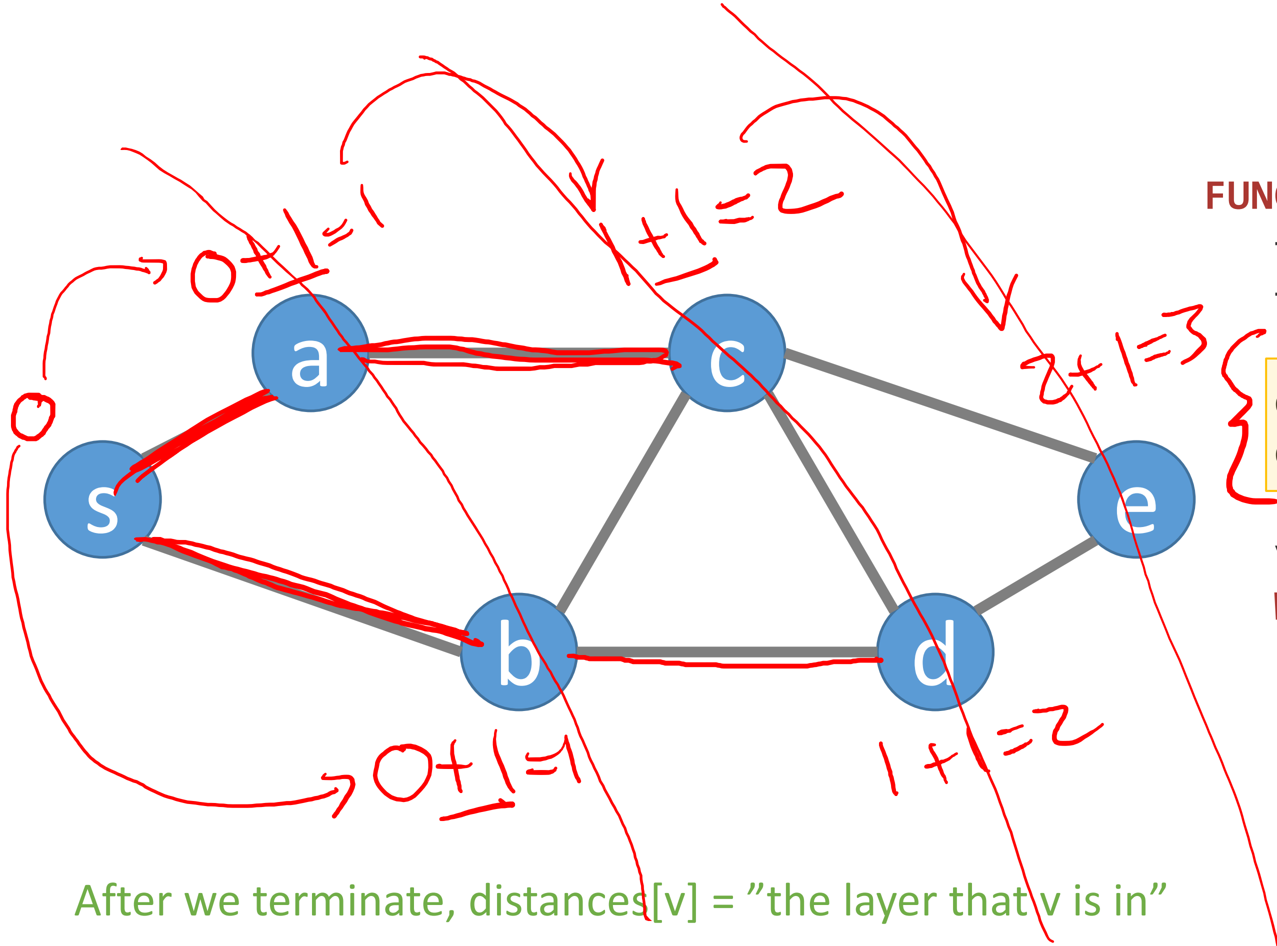
Determine the fewest number of hops between the start vertex and all other vertices

Same algorithm as before with the following additions:

- Initialize the distances[s] as 0
- Initialize all other distances to **infinity**
- When considering an edge (v, w)
  - If w is not found, then set dist(w) to dist(v) + 1



# The Shortest Path Problem

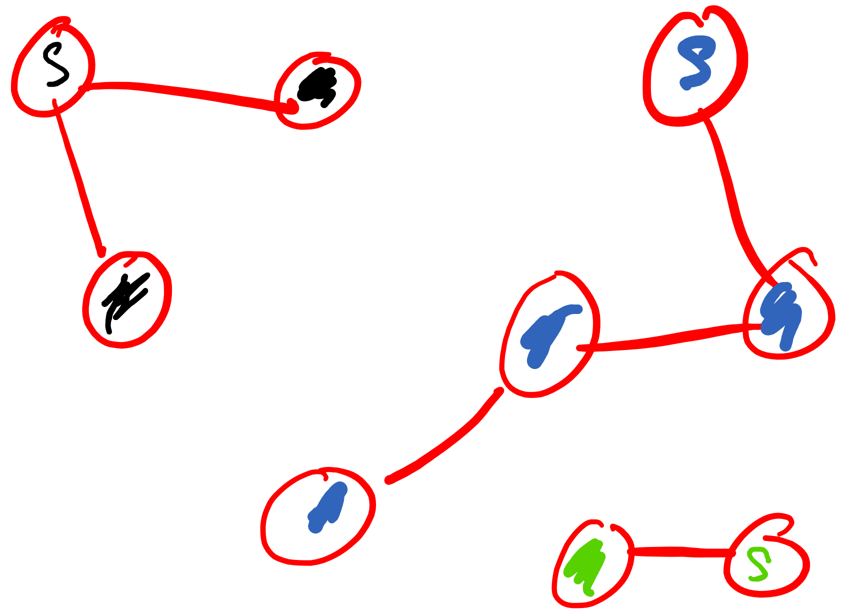



After we terminate, distances[v] = ”the layer that v is in”

```
FUNCTION DistanceBFS(G, start_vertex)
   found = {v: FALSE FOR v IN G.vertices}
   found[start_vertex] = TRUE

   distances = {v: INFINITY FOR v IN G.vertices}
   distances[start_vertex] = 0

   visit_queue = [start_vertex]
   WHILE visit_queue.length != 0
      vFound = visit_queue.pop()
      FOR vOther IN G.edges[vFound]
         IF found[vOther] == FALSE
            found[vOther] = TRUE
            visit_queue.add(vOther)
            distances[vOther] = distances[vFound] + 1

   RETURN distances
```

*Given a tie, visit edges are in alphabetical order*

# Connected Components



Let's only consider undirected graphs for now

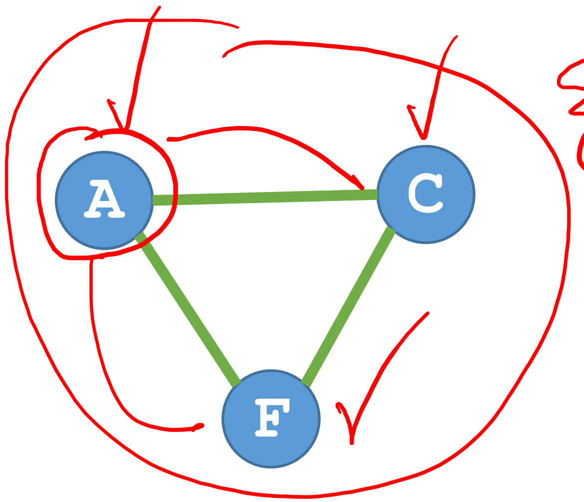
Let G = (V,E) be an undirected graph

Goal: compute all connected components in O(m + n)

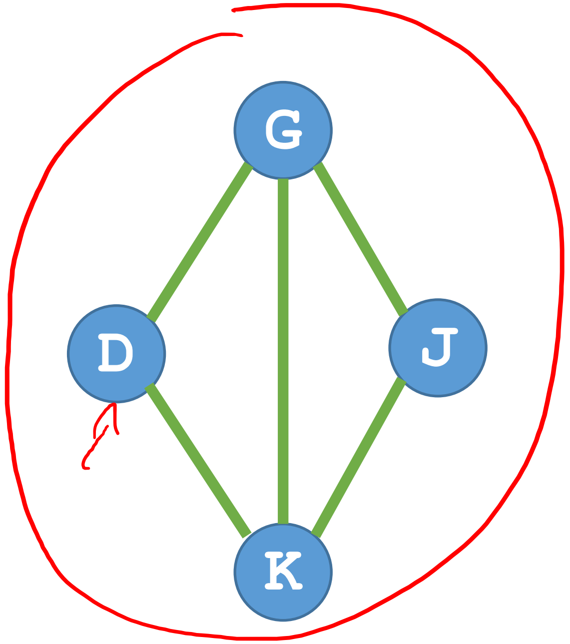
- A component is any group of vertices that can reach one another
- For example, if we are trying to see if a network has become disconnected

<u>Exercise question 2:</u>
How would you do this using our BFS procedure from before?

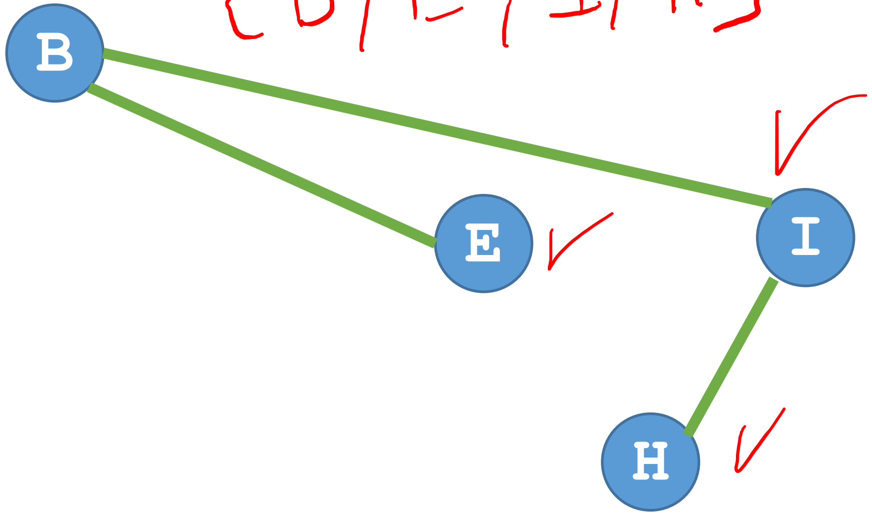
# BFS Exercise Question 2




{A, C, F}




{B, E, I, H}




```
FUNCTION FindComponents(G)
   components = []
   found = {v: FALSE FOR v IN G.vertices}
   FOR v IN G.vertices
      IF NOT found[v]
         newly_found = BFS(G, v)
         new_component = {
            w FOR w, w_is_found IN newly_found
            IF w_is_found
         }
         component.append(new_component)
         FOR w IN new_component:
            found[w] = TRUE
   RETURN components
```